UNIVERSITÉ D'AIX-MARSEILLE
ÉCOLE DOCTORAL DE MATHÉMATIQUE ET INFORMATIQUE, ED 184
UFR SCIENCES
Laboratoire d'informatique Fondamentale de Marseille, CNRS UMR 7279

THÈSE DE DOCTORAT

Discipline : Informatique

par

Worachet UTTHA

sous la direction de Clara BERTOLISSI, Jean-Marc TALBOT

Étude des politiques de sécurité pour les applications distribuées : le problème des dépendances transitives

Modélisation, Vérification et Mise en œuvre

Soutenue le 26/09/2016 devant le jury :

Horatiu Cirstea	Université de Lorraine	Rapporteur
Pascal Lafourcade	Université d'Auvergne	Rapporteur
Maribel Fernández	King's College London	Examinateur
Emmanuel Godard	LIF, Université d'Aix-Marseille	Examinateur
Silvio Ranise	Bruno Kessler Foundation	Examinateur
Clara Bertolissi	LIF, Université d'Aix-Marseille	Directeur de thèse
Jean-Marc Talbot	LIF, Université d'Aix-Marseille	Directeur de thèse

Résumé

Le contrôle d'accès est un ingrédient fondamental de la sécurité des systèmes d'information. Depuis les années 70, les travaux dans ce domaine ont apporté des solutions aux problèmes de confidentialités des données personnelles avec applications à différents environnements (systèmes d'exploitation, bases de données, etc.). Parmi les modèles de contrôle d'accès, nous nous intéressons au modèle basé sur les organisations (OrBAC) et nous en proposons une extension adaptée aux contextes distribués tels que les services web. Ce modèle étendu est capable en particulier de gérer les demandes d'accès transitifs. Ce cas peut se produire lorsqu'un service doit appeler un autre service qui peut avoir besoin d'en invoquer à son tour un ou plusieurs autres afin de satisfaire la demande initiale.

Nous appelons D-OrBAC (*Distributed Organisation Based Access Control*), l'extension du modèle OrBAC avec une notion de procuration représentée par un graphe de délégation. Ce graphe nous permet de représenter les accords entre les différentes organisations impliquées dans la chaîne d'invocations de services, et de garder la trace des autorisations transitives. Nous proposons également une technique d'analyse basée sur Datalog qui nous permet de simuler des scénarios d'exécution et de vérifier l'existence de situations non sécurisées.

Nous utilisons ensuite des techniques de réécriture afin d'assurer que la politique de sécurité spécifiée via notre modèle D-OrBAC respecte les propriétés importantes telles que la terminaison et la consistance. Finalement, nous implémentons pour un cas d'étude, le mécanisme d'évaluation des demandes d'accès selon la norme XACML sur la plateforme *WSO2 Identity Server*, afin de montrer que notre solution est capable de fournir à la fois la fonctionnalité désirée et la sécurité nécessaire pour le système.

Mots clés : *Contrôle d'accès, Vérification, OrBAC, SOA, XACML*

Abstract

The access control is a fundamental ingredient of computer security. Since the 70s, the research in this area has provided many solutions to the privacy issue of personal data with applications to different environments (operating systems, databases, etc.). Among many access control models, we are interested in the model based on organisations (OrBAC) and we propose an extension adapted to distributed environments such as web services. This extended model is able, in particular, to handle access transitive requests. This situation can occur when a service has to call another service that may need to invoke in turn one or more services to meet the initial demand.

We call D-OrBAC (Distributed Organisation Based Access Control), the extension of OrBAC model with a notion of delegation represented by a delegation graph. This graph allows us to represent agreements between the different organisations involved in the chain of service invocations, and to keep track of transitive authorisations. We also propose an analytical technique based on Datalog that allows us to simulate execution of scenarios and to check for the existence of unsafe situations.

Thereafter, we use rewriting techniques to ensure that the security policy specified via our D-OrBAC model complies with important properties such as termination and consistency. Finally, we implement for a case study, the mechanism of access request evaluations according to the XACML on the WSO2 Identity Server platform to show that our solution is able to provide both the desired functionality and the security for the system.

Keywords : *Access Control, Verification, OrBAC, SOA, XACML*

Remerciements

Je tiens tout d'abord à remercier chaleureusement ma famille (mon père Preecha, ma mère Wilai, mon frère Sitthichai et tous mes proches) qui est toujours présente à mes côtés. Je suis devenu celui que je suis à présent grâce à vous. Je suis très fier d'être un membre de la famille Uttha et Pimsuta. Vous m'avez toujours soutenu même si nous sommes à l'autre bout du monde.

Je remercie le gouvernement thaïlandais de m'avoir donné la bourse ODOS (One District One Scholarship), qui m'a donné l'opportunité de faire ma licence en France, ainsi que le Ministère thaïlandais des Sciences et Technologies, et Rajabhat Nakhon Pathom Université pour la bourse de poursuite d'études en Master et en Doctorat.

Je voudrais remercier Clara Bertolissi et Jean-Marc Talbot d'avoir accepté d'être mes directeurs de thèse. Ces années de travail sous votre direction m'ont permis de me développer tant intellectuellement, que professionnellement et me seront très utiles pour la suite de ma carrière.

Je remercie Horatiu Cirsrea et Pascal Lafourcade d'avoir accepté de relire mon manuscrit, ainsi que pour leurs retours constructifs. Je remercie aussi Maribel Fernàndez, Emmanuel Godard et Silvio Ranise d'avoir accepté de faire partie du jury.

Merci à Anthony pour la relecture de mon manuscrit et les corrections du français. Une pensée toute particulière pour tous mes amis, notamment les thaïlandais de Marseille : Patcharin, Sabaipohn, Saranyoo, Bunpot, Jumapohn, Thippawan, Orawan, Grisada, Sarayuth, Kanitta, Kanjana et Supanee. Merci pour votre présence, votre patience et votre soutien ; sans vous tous, ces 10 ans d'études en France aurait été très difficile.

Je remercie le LIF et son personnel pour leur accueil et en particulier l'équipe administratif, Nadine, Sylvie et Martine pour leur grand professionnalisme, et leur gentillesse. Je remercie aussi tous les doctorants : Florent, Mathieu, Élie, Makki, Antoine, Christina, Didier et Éloi avec qui j'ai eu le plaisir de partager le bureau et la vie doctorant.

Enfin, je remercie le centre de recherche FBK et l'équipe Security and Trust de m'avoir accueilli plusieurs fois dans leurs locaux, dans le cadre des échanges scientifiques de ma thèse.

Table des matières

Table des figures

Introduction

Contexte

Depuis les deux dernières décennies, documents, archives, données, enregistrements médicaux, etc. sont souvent numérisés et stockés sur des dépôts électroniques pour permettre l'accès à distance par différents utilisateurs. Un très grand nombre d'informations sensibles circule dans un système d'information de grande taille, que ce soit au sein de son organisation, ou partagées avec des organisations partenaires. C'est ainsi que les systèmes distribués tels que les *architectures orientées services (SOA)* sont devenus un des choix privilégiés pour réaliser le partage des données au travers de plusieurs domaines. Dans le paradigme SOA, des applications fournissent une interface standard et sont stockées comme des entités réutilisables pour la composition. SOA ne demande pas d'intégration de composants ou de codes dans son environnement de calcul, il ne nécessite que la spécification d'échange de données afin que le résultat final puisse être produit en exploitant les résultats des services coopérants.

Dans SOA, les composants ont en effet peu de (voire aucune) connaissance des définitions des autres composants séparés (Loosely coupled). Cela permet de rendre la plateforme de calcul indépendante, c'est-à-dire que chaque application peut être développée indépendamment, mais peut utiliser le résultat des autres applications via une interface pour réaliser son calcul. Par ailleurs, de nombreux processus opérationnels (business processes) deviennent dépendants des services fournis par d'autres entités en dehors de leurs propres domaines. Par exemple dans le cas d'une banque, le partage des données des clients avec d'autres banques ou institutions financières peut être nécessaire pour valider les demandes de prêts des clients. De façon similaire, pour des achats en ligne, la validation d'une commande peut nécessiter des informations supplémentaires du client qui seraient fournies par la banque ou le service de paiement en ligne (ex. Paypal) afin de vérifier la capacité de dépenses pour valider le paiement. De même dans le domaine médical, le partage des certains dossiers médicaux des patients entre les prestataires de services santé

(hôpital, laboratoire, pharmacie, etc.) est requis pour de nombreuses raisons.

Un large éventail de technologies est capable d'implémenter SOA, e.g. REST [1], Java RMI [2], ou des services web [3]. Ces derniers sont très fréquemment utilisés pour l'implémentation de SOA. Un service web est un programme informatique qui permet à des applications d'effectuer des échanges de données et de communiquer entre elles dans un système distribué à travers le réseau internet. Les services web deviennent la technologie préférée d'implémentation pour l'intégration et l'interaction entre différents systèmes dans les environnements internet et intranet. Ils fournissent un lien entre plusieurs applications (ou machines) même si celles-ci utilisent des technologies différentes en leur permettant d'envoyer et de recevoir des données à travers le protocole http, ces données étant exploitables partout. Les services web sont alors un moyen rapide de distribuer des informations entre demandeurs, fournisseurs et collaborateurs dans différentes plates-formes.

Un ingrédient important dans le développement de services web fiables est la sécurité, et en particulier, la protection des données partagées contre des utilisations indésirables. Le *contrôle d'accès* détermine quels utilisateurs sont autorisés à utiliser quelles données. Il est généralement décomposé en trois étapes différentes : *Identification, Authentification,* et *Autorisation.* L'identification est l'étape au cours de laquelle un utilisateur va prétendre être quelqu'un (e.g. son login) ; cette affirmation peut être vraie ou fausse et sera prouvée au cours de l'étape suivante. La deuxième étape, l'authentification, consiste en ce que l'utilisateur identifié prouve qu'il est bien celui qu'il prétend être (e.g. son mot de passe). Ces deux étapes effectuées avec succès se résument ainsi : l'utilisateur a prétendu puis prouvé être quelqu'un. C'est ainsi que parfois l'identification et l'authentification coexistent dans une même phase. Enfin la dernière étape du contrôle d'accès est l'autorisation. Cette étape a lieu après que l'utilisateur ait été à la fois identifié et authentifié ; c'est une étape qui détermine les ressources auxquelles l'utilisateur est autorisé à accéder. Toutes les autorisations sont définies dans un document appelé *politique de contrôle d'accès*, document dans lequel sont définies des règles qui précisent quels utilisateurs peuvent effectuer quelles actions sur quelles ressources. Un formalisme appelé *modèle de contrôle d'accès* est ensuite défini pour faciliter la compréhension de la politique, spécifier le comportement d'un système de manière exacte et non ambiguë, et pour abstraire une implémentation des mécanismes de contrôle d'accès.

Depuis l'apparition de la matrice d'accès de Lampson à la fin des années

1. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`
2. `http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/`
3. `www.w3.org/TR/ws-arch/`

60, plusieurs modèles de contrôle d'accès ont été proposés par exemple DAC, MAC, RBAC et ABAC. Chaque modèle a des avantages et des inconvénients différents selon le contexte et l'environnement où la politique de sécurité sera appliquée.

Problématique

Dans le cas de SOA, il est fréquent que l'administration des aspects sécurités soit effectuée de manière autonome par chaque service, qui peuvent appartenir à des organisations différentes. Il est aussi fréquent qu'un service doive en appeler un second, lequel peut ensuite avoir besoin d'invoquer d'autres services afin de satisfaire la demande initiale. Nous appelons cela « *problème d'accès transitif* ». Ce problème est complexe car les services web sont développés et gérés de façons indépendantes, chacun possédant sa propre politique de contrôle d'accès. Même si des utilisateurs ont le droit d'accéder au service initial, cela ne garantit pas qu'ils aient les autorisations pour accéder aux services invoqués ultérieurement, lesquels pouvant appartenir à d'autres organisations. Cela peut conduire à des erreurs d'autorisation indirectes. Concernant les mécanismes d'authentification et de contrôle d'accès, ils doivent prendre en compte l'accès des utilisateurs externes de manière efficace et évolutive, tout en considérant la confiance envers des organisations externes. Pour l'authentification, il existe une solution standardisée tel que le *Single-Sign On*[4] *(SSO)* qui permet aux utilisateurs de se connecter une fois et d'exploiter tous les services mis à disposition par un fournisseur de services. Cependant, la situation est beaucoup moins satisfaisante pour l'autorisation.

Une généralisation du modèle de contrôle d'accès basé sur les rôles (RBAC) tel que le modèle basé sur les organisations (OrBAC) est proposée dans [Kalam et al., 2003] pour augmenter l'expressivité du modèle RBAC, de sorte qu'il soit possible de gérer simultanément plusieurs politiques de sécurité associées à différentes organisations. Cependant, le concept des organisations de ce modèle est centralisé, c'est-à-dire qu'une organisation est divisée en plusieurs sous-organisations ; des accès aux ressources ne sont possibles qu'entre ces sous-organisations. Ainsi le système OrBAC ne peut pas identifier et authentifier le sujet pour déterminer ses permissions en dehors de son organisation. Cela n'est pas suffisant dans le cas de SOA afin de résoudre le problème d'accès transitif au travers différentes organisations puisque des utilisateurs peuvent ne pas être reconnus dans toutes les organisations.

Dans cette thèse, nous allons aborder plusieurs questions liées à la spécifi-

4. `http://www.opengroup.org/security/sso`

cation, à la validation, à la vérification et à la mise en œuvre des politiques de contrôle d'accès :

— Comment peut-t-on surmonter les limites du modèle Or-BAC afin qu'il puisse modéliser des permissions où plusieurs organisations sont impliquées ?

— Y a-t-il des techniques pour effectuer une analyse automatique pouvant détecter des scénarios d'exécutions qui peuvent générer des problèmes d'autorisation provoqués par la définition de la politique de contrôle d'accès ?

— Est-ce que notre modèle de contrôle d'accès est bien conçu et sans erreur ?

— Est-ce que notre solution est capable de fournir à la fois un système avec les fonctionnalités désirées et la sécurité nécessaire ?

Contribution

La contribution principale de ce travail est de fournir une solution efficace et adaptable à la mise en œuvre d'un contrôle d'accès dans un contexte distribué et dynamique tel que celui des services web. Cette thèse apporte plusieurs contributions :

1. La définition d'un nouveau modèle de contrôle d'accès appelé "*D-OrBAC : Distributed-Organisation Based Access Control*".
2. La validation du modèle D-OrBAC par les analyses des scénarios d'exécution au travers deux cas d'étude en utilisant Datalog comme langage de spécification du modèle.
3. La vérification de propriétés d'une politique de contrôle d'accès spécifié dans D-OrBAC (telles que la terminaison, la consistance et la totalité) en utilisant des techniques de réécriture.
4. La mise en œuvre de la politique sur un cas d'étude afin de montrer que notre solution respecte la sécurité nécessaire pour un système tout en préservant les fonctionnalités désirées.

Afin d'expliciter notre approche d'une manière plus concrète, nous étudions notre modèle au travers de deux cas d'études :

1. Un centre médical inspiré de [Fischer and Majumdar, 2008] qui se compose de quatre organisations : portail web, management de clinique, laboratoire d'analyse, historique des patients. Chaque organisation met à disposition certains services web afin d'échanger des informations avec ses différentes organisations partenaires.

2. Un centre de recherche qui a développé une solution informatique pour la gestion des autorisations de réservation d'hôtel, de transport, et autres dépenses des chercheurs en mission. Il est composé de quatre départements : secrétariat, administratif, comptabilité, informatique.

Notre contribution est répartie et présentée dans différents publications [Utha et al., 2014a,b, 2015a,b, 2016].

1. Définition du modèle D-OrBAC.

Nous proposons un nouveau modèle de contrôle d'accès appelé *"Distributed - Organisation Based Access Control" (D-OrBAC)* qui permet de spécifier une politique de sécurité pour protéger des ressources ou des données partagées par différentes organisations dans un système distribué. Plus précisément, dans le cas d'un appel transitif où plusieurs organisations sont impliquées, des services intermédiaires invoquent d'autres services au nom de l'utilisateur qui invoque le service initial. Afin de représenter les liens entre différentes organisations impliquées dans la chaîne d'invocation de services, nous introduisons des relations (binaires) représentant les accords entre organisations. Cela est formalisé par un graphe orienté acyclique appelé *graphe de délégation*. Nous définissons une extension du modèle OrBAC avec des délégations qui permet de modéliser les règles de permission dans le cas où plusieurs organisations sont impliquées. Le graphe de délégation nous permet de spécifier des accords entre les différentes organisations partenaires, il nous permet également de garder la trace des autorisations transitives, ce qui élargit la flexibilité du modèle Or-BAC original.

2. Validation du modèle D-OrBAC sur deux cas d'étude.

Une fois le modèle D-OrBAC défini, nous voulons valider notre modèle de contrôle d'accès afin d'assurer que la politique de sécurité spécifiée par notre modèle répond aux besoins réels, c'est-à-dire que la politique protège correctement les ressources et donne les accès aux bonnes personnes. Pour cela, nous proposons d'effectuer une simulation des scénarios d'exécutions avant le déploiement du système en utilisant Datalog comme langage de spécification de notre modèle de contrôle d'accès. Cette technique nous permet de tester les différentes exécutions de scénarios susceptibles de mener à une situation non sécurisée. Notre technique d'analyse possède deux avantages importants. Tout d'abord, il n'est pas nécessaire d'implémenter un prototype afin d'expérimenter le comportement du système parce que nous pouvons effectuer la simulation du système directement via le moteur de Datalog. Ensuite, sachant

qu'il est parfois difficile d'accéder aux environnements exécutant les différents services et qu'il se peut que cela rende le test du système composé plus difficile, nous avons porté notre choix sur une technique capable de prédire les résultats en utilisant la spécification abstraite du système sans encourir de coûts supplémentaires pour faire des tests.

3. Vérification d'une politique en utilisant la réécriture.

Afin d'assurer que la politique de sécurité spécifiée par le modèle D-OrBAC respecte les propriétés importantes de la politique, nous spécifions notre politique par un système de réécriture des termes. Par la suite, nous montrons les correspondances entre des propriétés de la politique et des propriétés du système de réécriture. Le système de réécriture nous permet de vérifier les propriétés de la politique telle que la terminaison et la consistance, ce qui garantit que la politique de sécurité est capable de répondre à toutes les demandes d'accès et que cette réponse est unique. Nous présentons également un outil CiME qui nous permet d'effectuer une vérification d'une politique de contrôle d'accès spécifiée par un système de réécriture.

4. Mise en œuvre de notre approche.

Afin de montrer que notre approche satisfait à la fois la sécurité requise pour un système et la fonctionnalité souhaitée, nous avons choisi deux cas d'étude (centre médical et centre de recherche) et nous avons mis en œuvre le mécanisme de contrôle d'accès dans le centre médical. Pour cela nous développons des services web en utilisant Java et les technologies standards (UDDI comme un annuaire de services, WSDL pour des définitions d'interface du service, SOAP pour des invocations de services et XML comme un format de communication). Ces services sont fournis par différentes organisations et protégés par notre politique de sécurité. Notre objectif est d'étendre les capacités et les aptitudes des outils standards existants afin de répondre au problème d'accès transitif dans l'environnement distribué. Pour cela, nous implémentons le mécanisme d'évaluation des demandes d'accès selon la norme XACML sur la plateforme WSO2 Identity Server, laquelle fait partie d'une open source middleware pour le développement des applications distribuées.

Organisation du manuscrit

Ce manuscrit se divise en quatre chapitres.

- Le chapitre 1 présente l'état de l'art des modèles de contrôle d'accès de base, le modèle OrBAC original et finalement la définition formelle de notre modèle de contrôle d'accès basé sur l'organisation, étendu avec le graphe de délégation. Nous appelons cela le modèle "*Distributed-Organisation Based Access Control (D-OrBAC)*".
- Le chapitre 2 présente une technique d'analyse automatique du comportement du système protégé par notre politique de sécurité avant son déploiement en utilisant Datalog. Nous validons le modèle D-OrBAC au travers deux cas d'étude : le centre médical et le centre de recherche.
- Dans le chapitre 3 nous utilisons des techniques de réécriture pour la vérification des propriétés de la politique telles que la terminaison et la confluence comme cela peut être vérifié de façon automatique avec l'outil CiME.
- Le chapitre 4 présente la mise en œuvre des politiques de sécurité en utilisant les outils existants. Nous montrons comment notre modèle formel D-OrBAC peut être traduit vers une architecture XACML. Par la suite, nous présentons une extension de XACML avec un module de délégation et l'implémentation du centre médical.

Chapitre 1

Modélisation du contrôle d'accès dans un cadre distribué

Sommaire

La sécurité d'un système d'information a pour objectif d'assurer :
(1) l'**intégrité**, c'est-à-dire de rendre impossible une modification inappropriée de l'information, les données attendues doivent par ailleurs être exactes et complètes ; (2) la **confidentialité**, *i.e.* seules les personnes qui en ont le droit peuvent accéder aux informations demandées, tout accès indésirable doit être empêché ; enfin (3) la **disponibilité** qui garantit que l'accès aux services ou aux ressources est toujours disponible. Dans la littérature, le contrôle d'accès vise à garantir les trois propriétés de sécurité mentionnées ci-dessus.

Cependant dans la réalité, la plupart du temps seules la confidentialité et l'intégrité peuvent être garanties par le contrôle d'accès, ce qui est moins évident pour la disponibilité. Pour résumer, le contrôle d'accès est un mécanisme qui garantit la confidentialité et l'intégrité. Parmi ces propriétés de sécurité, nous nous intéressons en particulier à assurer la confidentialité des informations en utilisant le contrôle d'accès grâce auquel le système autorise ou interdit aux utilisateurs d'effectuer des actions sur des ressources.

Une **politique de sécurité** est un document qui exprime clairement et de manière concise les mécanismes de protection à réaliser sans tenir compte de la façon dont ils seront accomplis ; de même, une **politique de contrôle d'accès** est un document dans lequel sont définies des règles qui précisent quels utilisateurs peuvent effectuer quelles actions sur quelles ressources. La spécification des politiques de sécurité demande une compréhension de tous les éléments qui peuvent influencer la décision de contrôle d'accès, raison pour laquelle son élaboration est toujours complexe. Afin de faciliter la spécification, il est nécessaire que le cadre formel fournisse à la fois l'expressivité pour capturer des exigences complexes et des techniques fiables pour vérifier les propriétés de la politique.

Afin de comprendre comment la conception de notre cadre a été influencée, nous présentons dans la section 1.1 différents modèles de contrôle d'accès de base (DAC, MAC et RBAC) ainsi que les avantages et désavantages de chaque modèle. Nous montrons ensuite le modèle de contrôle d'accès (OrBAC), qui exprime les règles de sécurité via des entités abstraites (des organisations, des rôles, des vues, des activités et des contextes). Ce modèle nous permet d'exprimer les politiques de contrôle d'accès au niveau organisationnel et il est indépendant de l'implémentation. Il est ainsi possible de spécifier l'ensemble des exigences de sécurité pour l'ensemble des organisations en respectant la fonctionnalité locale de chaque organisation composante.

Nous terminons par la présentation de notre approche qui est capable de surmonter les limites déjà présentées et de résoudre le problème de l'accès transitif à travers plusieurs domaines de contrôle d'accès.

1.1 État de l'art

1.1.1 Contrôle d'accès

Le principe du « **Contrôle d'accès** » consiste à accorder ou à refuser une demande provenant d'un sujet authentifié pour effectuer des actions sur des ressources. Les sujets peuvent par exemple être des utilisateurs, des processus ou des entités informatiques qui représentent les utilisateurs dans des sys-

tèmes et agissent en leurs noms. De même les ressources peuvent être des objets, des fichiers, des imprimantes ou des tables relationnelles dans des bases de données. Enfin les actions peuvent être des opérations possibles que nous pouvons effectuer dans des systèmes d'information ou des systèmes de gestion de base de données, par exemple écrire, lire, modifier, supprimer, etc. Dans certains systèmes, un accès complet est accordé à l'utilisateur qui s'est authentifié, mais la plupart des systèmes nécessitent un contrôle plus sophistiqué et complexe.

Définition 1.1: Contrôle d'accès

Le contrôle d'accès est un mécanisme grâce auquel un système autorise ou interdit les actions demandées par des sujets (entités actives) sur des ressources (entités passives).

L'**Authentification** est le processus visant à déterminer si une personne est celle qu'elle prétend être. L'authentification se fait généralement en utilisant un nom d'utilisateur et un mot de passe, mais cela peut inclure toute autre méthode permettant d'identifier une personne, comme l'utilisation des empreintes digitales, l'analyse rétinienne, la reconnaissance vocale ou d'autres méthodes biométriques. Dans les systèmes de sécurité, l'authentification est distincte de l'**Autorisation**, qui est un processus permettant de donner aux utilisateurs un accès aux ressources du système en fonction de leur identité, mais l'authentification ne dit rien sur les droits d'accès de l'utilisateur. Ainsi, le contrôle d'accès porte sur la façon dont l'organisation est structurée. Le développement d'un système de contrôle d'accès nécessite la définition d'un règlement selon lequel l'accès doit être contrôlé. Par ailleurs leur implémentation est basée sur trois niveaux principaux : politiques, modèles et mécanismes de contrôle d'accès [Samarati and de Vimercati, 2001].

— *Politique de contrôle d'accès* : elle définit les règles selon lesquelles le contrôle d'accès doit être régularisé. Les politiques sont une exigence de haut niveau qui précisent d'une part, la façon dont le contrôle d'accès est structuré, et d'autre part quel utilisateur peut effectuer telles actions sur telles ressources.

— *Modèle de Contrôle d'accès* : il fournit une représentation formelle des politiques de contrôle d'accès. La formalisation permet de vérifier les propriétés de sécurité fournies par le système de contrôle d'accès.

— *Mécanisme de contrôle d'accès* : il provient généralement du niveau bas de l'abstraction où il applique ces politiques de contrôle d'accès de haut niveau et traduit la demande d'un utilisateur sous forme d'une structure spécifique que le système a fournie.

Les trois niveaux sont indépendants, c'est ainsi que les politiques peuvent être analysées abstraitement sans référence au mécanisme d'application, mais seulement à travers le respect du modèle, qui à son tour, peut être pris en considération lorsque l'on montre la correction du modèle d'application.

1.1.2 Modèles de contrôle d'accès

Le *modèle de sécurité* est une spécification d'une politique de sécurité pour la confidentialité, l'intégrité ou la disponibilité qui n'explique pas le mécanisme particulier pour les atteindre. Il décrit uniquement les entités régies par la politique et énonce les règles qui constituent la politique. Plusieurs modèles ont été proposés pour encoder les politiques de contrôle d'accès, ils peuvent être classés en trois catégories principales :

Discretionary Access Control (DAC)

Dans *Discretionary Access Control* ou « modèle de contrôle d'accès discrétionnaire » [Gallagher, 1987] chaque objet ou ressource du système a un propriétaire (un sujet), lequel peut déterminer les privilèges d'accès à cet objet. Le sujet a un contrôle complet sur tous les objets qui lui appartiennent, il peut changer les permissions d'accès, transférer des objets authentifiés ou des accès à l'information à d'autres sujets. C'est pourquoi il est dit discrétionnaire.

Dans ce modèle, les autorisations sont attribuées directement à des sujets en fonction de leur identité ; l'inconvénient d'une telle approche est que, dans les grands systèmes, déterminer l'octroi de l'autorisation sur une ressource donnée à des utilisateurs individuels, est laborieux et difficile à gérer. La révocation de la permission est également complexe lorsque l'utilisateur quitte l'entreprise ou change de fonction, par exemple. L'information peut être copiée d'un objet à un autre, de sorte que l'accès à une copie est possible même si le propriétaire initial ne donne pas accès à l'originale. Puisque les politiques du DAC peuvent être facilement modifiées par le propriétaire, un programme malveillant s'exécutant en son nom pourra aussi changer ces mêmes politiques, ce qui constitue une faiblesse de ce système.

Mandatory Access Control (MAC)

Dans *Mandatory Access Control* ou « modèle de contrôle d'accès obligatoire » [Benantar, 2006], seul l'administrateur de sécurité peut gérer les autorisations. C'est lui qui définit la politique d'utilisation et d'accès. La politique indiquera qui a accès à quelles ressources, mais les utilisateurs ne pourront pas

déterminer qui peut accéder à leurs fichiers. Dans ce modèle, toutes les informations sont affectées à un niveau de sécurité, et chaque utilisateur est affecté à une habilitation de sécurité. Sujets et objets possèdent des habilitations et des étiquettes, respectivement, comme par exemple « confidentiel », « secret », et « très secret ». Il garantit que tous les utilisateurs n'ont accès qu'aux données pour lesquelles ils possèdent une habilitation égale ou supérieure à l'étiquette de l'objet. MAC est généralement approprié pour les systèmes sécurisés tels que des applications militaires à plusieurs niveaux ou des applications de données critiques. Les modèles MAC bien connus sont par exemple Bell-La Padula [LaPadula et al., 1973], Biba [Biba, 1977], Clark-Wilson [Clark and Wilson, 1987].

MAC préserve la confidentialité et l'intégrité des informations, empêche certains types d'attaques comme Trojan Horse et prévient l'altération non autorisée des objets. Mais ses inconvénients majeurs sont manque de flexibilité et la difficulté à mettre en œuvre et à programmer ce modèle.

Role Based Access Control (RBAC)

Role Based Access Control ou « modèle de contrôle d'accès basé sur les rôles » [David and Richard, 1992] a été proposé pour fournir un modèle et des outils qui permettent de gérer le contrôle d'accès dans un système complexe avec un très grand nombre d'utilisateurs et d'objets. RBAC est basé sur la structure organisationnelle de l'entreprise. Les propriétaires peuvent mieux gérer et maintenir leurs ressources, mais ils sont limités à modifier les politiques de sécurité globales de l'entreprise. RBAC facilite aux systèmes la gestion des utilisateurs, en leur attribuant des rôles plutôt que d'utiliser l'identité de chaque utilisateur. Des décisions d'accès sont basées sur les rôles d'un utilisateur dans l'organisation. Les droits d'accès sont regroupés par nom de rôle, et l'accès aux ressources est limité aux utilisateurs qui ont été autorisés à assumer le rôle associé. Avec RBAC, certains utilisateurs peuvent avoir plusieurs rôles pour satisfaire les exigences de leur travail.

L'avantage de RBAC est qu'il aide les administrateurs à mieux contrôler les utilisateurs sans avoir à les associer explicitement aux ressources spécifiques. Les rôles des utilisateurs sont révocables, sans nécessiter la gestion séparée des autorisations de contrôle d'accès pour chaque individu ; cela réduit les coûts administratifs et les coûts d'entretien.

Cependant, dans le modèle RBAC il n'est pas possible de spécifier une permission qui dépend d'un certain contexte car si une permission particulière est donnée à un certain rôle, tous les utilisateurs qui possèdent ce rôle hériteront de cette permission. Un autre inconvénient de RBAC est que seul l'administrateur est capable de spécifier des permissions.

Dans la section suivante nous présentons comment réduire les limitations de RBAC lorsqu'il est utilisé pour spécifier la politique de sécurité d'un système qui comprend plusieurs organisations.

1.1.3 ORganization Based Access Control (OrBAC)

Le modèle OrBAC [Kalam et al., 2003] a été proposé pour la première fois pour surmonter les limitations des modèles de contrôle d'accès existants (*i.e.* DAC, MAC, RBAC). Ce modèle introduit un niveau d'abstraction dans laquelle les sujets sont abstraits en rôles, des actions sont abstraites en activités et les objets sont abstraits en vues. Chaque politique de sécurité est définie par et pour une organisation et la spécification des politiques est paramétrée par l'organisation de sorte qu'elle est capable de gérer simultanément plusieurs politiques de sécurité associées aux différentes organisations.

L'entité centrale dans ce modèle est l'**Organisation** qui peut considérée comme un groupe d'activités ou un groupe de services. Par exemple dans le domaine médical, l'organisation pourra être l'hôpital, une clinique ou bien des services dans un hôpital tel que le service des urgences, le service de chirurgie ou le service de radiologie. L'entité **Sujet** dans ce modèle peut être soit un utilisateur, soit une organisation. L'entité **Rôle** est utilisée ensuite pour représenter la relation entre des organisations et des sujets. La relation **Empower(**org, r, s**)** a été introduite pour exprimer le fait que des sujets jouent des rôles dans une organisation. Cela signifie que l'organisation org habilite le sujet s à jouer le rôle r.

L'entité **Objet** représente des entités passives comme des dossiers médicaux, des fichiers, des imprimantes ou des matériels dans l'hôpital. Ils sont abstraits en **Vue** pour faciliter la mise à jour des politiques de sécurité et structurer des objets quand un nouvel objet est ajouté dans le système. Intuitivement une vue correspond à un ensemble d'objets qui satisfait une propriété commune et elle caractérise la manière dont les objets sont utilisés dans l'organisation. La relation **Use(**org, o, v**)** est nécessaire pour représenter le lien entre une organisation, un objet et une vue, cela signifie que l'organisation org utilise l'objet o dans la vue v.

L'entité **Action** correspond aux actions informatiques qui peuvent être opérées sur le système (*i.e.* lire, écrire, modifier, exécuter). L'entité **Activité** a été utilisée comme une abstraction des actions qui ont un objectif commun. Des organisations peuvent considérer qu'une même action réalise des activités différentes selon l'organisation qui l'utilise ; ainsi, la relation **Consider(**org, act, a**)** a été introduite pour associer les entités Organisation, Action et Activité. Cela signifie que l'organisation org considère l'action act comme faisant partie de l'activité a.

Après l'introduction des entités de base, il reste l'entité **Contexte** qui permet aux organisations de spécifier des autorisations de rôles pour effectuer des activités sur les vues dans une circonstance concrète, ce qui n'est pas réalisable dans RBAC. Dans le modèle RBAC, si une certaine permission est accordée à un rôle, alors tous les utilisateurs qui jouent ce rôle héritent de cette permission, alors que dans le modèle OrBAC, le *contexte* couvrira des circonstances concrètes ce qui est définie par des règles logiques. Seul l'utilisateur qui satisfera à ces règles pourra hériter de ce privilège. Chaque contexte peut être vu comme une relation ternaire entre les sujets, les objets et les actions, ce qui est défini dans l'organisation. Par conséquent ces entités sont définies par la relation **Define(**org, s, o, act, c**)** cela signifie que dans l'organisation org, le contexte c est vrai entre le sujet s, objet o et actions act.

Nous avons vu toutes les entités du modèle, nous pouvons donc maintenant spécifier des règles de sécurité dans le modèle OrBAC en utilisant la relation **Permission(**org, r, a, v, c**)** ce qui est une abstraction des permissions et correspond à une relation entre les organisations, les rôles, les activités et les contextes. Cette relation signifie que l'organisation org accorde au rôle r la permission de réaliser l'activité a sur la vue v dans le contexte c. Afin d'exprimer les actions concrètes qui peuvent être effectuées par des sujets sur les objets, la relation **Is_permitted(**s, act, o**)** a été introduite pour modéliser la permission concrète. Cela signifie que le sujet s est autorisé à effectuer l'action act sur l'objet o.

Même si OrBAC peut améliorer la gestion de la politique de sécurité et réduire sa complexité grâce aux abstractions des entités, il reste relativement limité car il n'est pas adapté aux systèmes distribués et interopérables. Nous avons vu que la règle de sécurité est sous forme
$Permission(org, r, v, a, c)$, ce qui ne permet pas de représenter des règles qui impliquent plusieurs organisations, par exemple dans le cas de la coopération entre plusieurs entreprises, un utilisateur appartenant à l'entreprise A souhaitant accéder aux ressources appartenant à ses partenaires (entreprises B, C,etc.). Par conséquence, OrBAC n'est pas très adapté pour modéliser un environnement dynamique et décentralisé où les organisations coopèrent en respectant l'autorité des autres sur ses propres utilisateurs et ses ressources.

Plusieurs extensions du modèle OrBAC ont été proposées récemment dans le but de définir des règles de sécurité intra ainsi que inter-organisations. Par exemple, le modèle d'administration pour Or-BAC (AdOr-BAC) [Cuppens and Miege, 2004] ; ou encore l'extension qui prend en compte les règles et les niveaux d'intégrité basés sur le modèle d'intégrité de Totel (MultiLevel-OrBAC) [Baina and Laarouchi, 2012]. Enfin l'outil MotOrBAC [Autrel et al., 2008] qui a été développé afin de faciliter la conception de politique de sécurité du mo-

dèle OrBAC à l'administrateur. Il permet de simuler des politiques de sécurité, détecter les conflits et aider le concepteur à les éliminer. Nous ne utilisons pas cet outil parce qu'il ne supporte que le système multi-organisationnel centralisé sans l'accès transitif dans lequel les différentes organisations provenant de différents environnements peuvent être impliquées. Par contre, nous empruntons la façon dont il définit la politique de sécurité en Prolog pour valider notre modèle.

1.2 Cas d'étude

Afin d'expliciter notre approche d'une manière plus concrète, nous utilisons, tout au long de notre travail, le cas d'étude sur le centre médical inspiré par le travail de [Fischer and Majumdar, 2008] et le cas d'étude sur le centre de recherche que nous avons présenté à ESSoS2015 [Uttha et al., 2015b].

1.2.1 Centre médical

Dans ce cas d'étude, nous considérons un centre médical composé de quatre organisations où chaque organisation fournit un ou plusieurs services [voir figure 1.1] :

— Clinical Management (CM) gère la planification des rendez-vous des patients et capture les actions réalisées par les médecins et les infirmières.

— Laboratory (LA) suit la trace les tests à effectuer et leurs résultats.

— Patient Records (PR) gère les données historiques sur la santé de chaque patient.

— Web Portal (WP) offre un accès web aux trois autres services. Il ne stocke pas localement des données confidentielles. Lorsqu'un utilisateur demande l'accès à un service, le portail effectue une invocation aux autres services en utilisant les attributs de cet utilisateur.

Chaque organisation possède ses propres politiques de contrôle d'accès qui sont gérées de manière indépendante afin de protéger ses ressources.

Scénario. Un médecin veut consulter un dossier médical d'un patient, il se connecte sur le portail web et demande l'accès à ce dossier via le service $PatientMedData$. Afin de satisfaire la demande initiale du médecin, ce service doit invoquer le service $CareOrders$ (mis à disposition par l'organisation $Clinical\ Management$) qui est en charge d'enregistrer toutes les actions réalisées par des médecins dans ce centre médical. Le service $CareOrders$, à son tour,

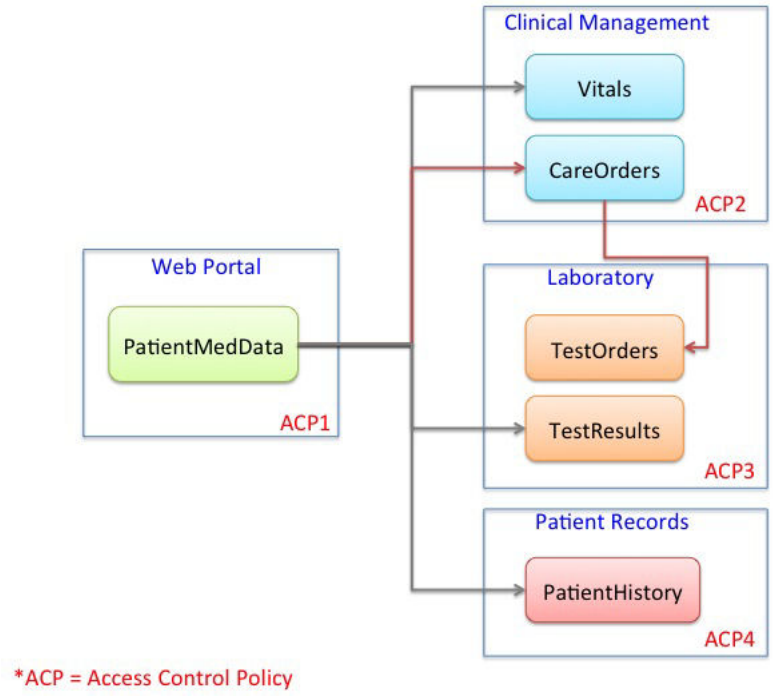


FIGURE 1.1 – Topologie du cas d'étude et un exemple d'une demande d'autorisation avec la dépendance transitive (chemin rouge).

est obligé d'appeler le service *TestOrder* (mis à disposition par l'organisation *Laboratory*) pour récupérer les détails des analyses de soins qui ont été ordonnées par les médecins. Quand toutes les informations sont rassemblées, le rapport médical de ce patient est livré au médecin.

1.2.2 Centre de recherche

Nous avons créé un second cas d'étude basé sur un centre de recherche, qui a développé une solution informatique pour la gestion des autorisations de réservation d'hôtel, de transport, et autres dépenses associées lorsque ses chercheurs partent en mission.

Le centre de recherche [Uttha et al., 2015b] est composé de 4 départements [Figure 1.2] :

— *Secretary's Office* (SEC) reçoit la demande des chercheurs et crée l'estimation du budget ainsi que la demande d'autorisation

— *Administrative Department* (ADD) valide et autorise la demande

— *Accounting Department* (ACD) gère le budget du centre de recherche

— *IT Department* (ITD) gère les données historiques des missions des chercheurs

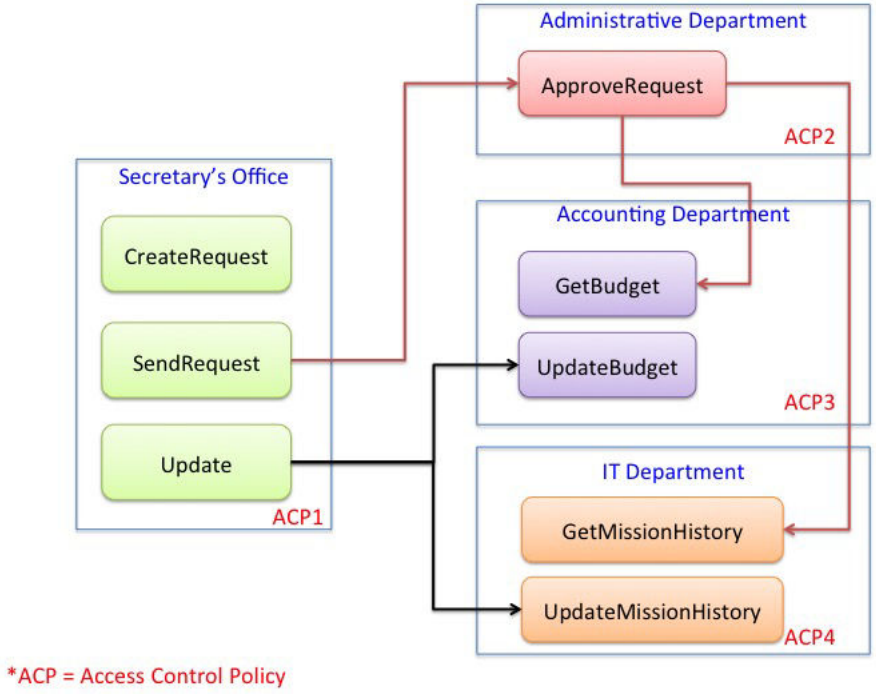


FIGURE 1.2 – Topologie du centre de recherche.

Scénario. Un chercheur (Sujet) doit partir en mission pour participer à une conférence afin de présenter son travail, il sollicite la secrétaire pour la réservation de l'hôtel et du voyage ainsi que pour le paiement des frais d'inscription. La secrétaire (Initiateur) crée alors un prévisionnel du budget et génère la requête d'autorisation, qu'elle transmet ensuite au chef d'équipe ou au responsable concerné en fonction de la nature du projet auquel le chercheur appartient. Avant de valider ou d'autoriser la demande, le chef d'équipe (Approuveur) doit vérifier si ce chercheur n'a pas encore dépassé son quota de déplacement en demandant au service informatique l'historique des missions, puis vérifier si les dépenses demandées ne sont pas trop élevées et respectent le plafond autorisé en sollicitant le service comptabilité. Une fois la décision prise (validation ou refus), la réponse est retournée à la secrétaire pour la

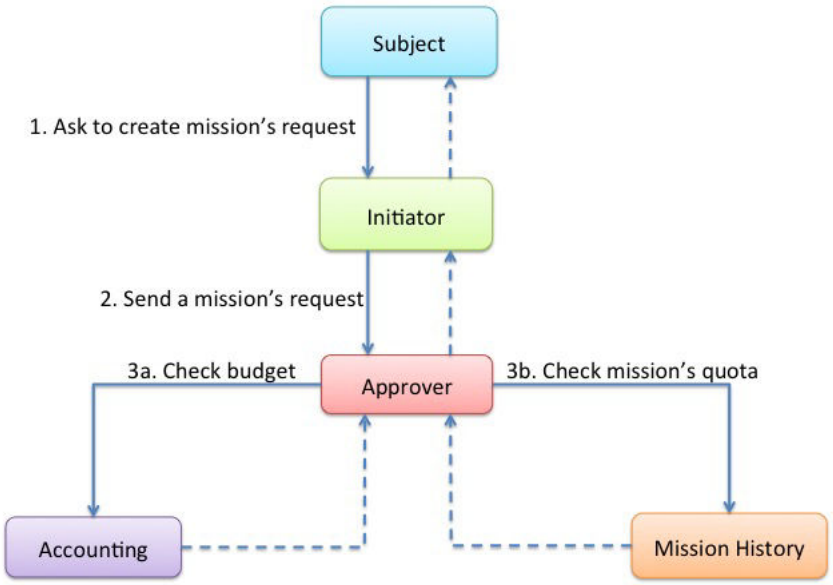


FIGURE 1.3 – Scénario d'un appel transitif dans le cas du centre de recherche.

mise à jour des données et la notification au chercheur. Le scénario d'un appel transitif est présenté à la figure 1.3.

Nous allons utiliser ces deux cas d'étude tout au long de ce travail pour montrer que notre modèle de contrôle d'accès est adapté à ceux types de l'application distribuée.

1.3 Notre modèle : Distributed-Organization Based Acces Control : D-OrBAC

Nous avons vu dans la section 1.1.3 que le modèle OrBAC peut traiter la spécification des politiques de sécurité pour des organisations et des sous-organisations, mais une seule organisation peut être impliquée dans chaque règle de sécurité, ce qui convient aux systèmes centralisés. OrBAC n'est pas adapté à l'environnement distribué où un sujet peut être reconnu dans une organisation locale mais peut ne pas être reconnu dans une autre organisation à distance. Nous formalisons donc dans cette section comment surmonter ces limitations en étendant le modèle OrBAC original avec une notion de

délégations pouvant rendre le modèle convenable aux systèmes distribués. Nous présentons notre extension du modèle OrBAC en utilisant un langage formel basé sur la logique du premier ordre. Nous appelons cela le modèle "*Distributed-Organisation Based Access Control (D-OrBAC)*".

1.3.1 Entités du modèle D-OrBAC

Les entités principales.

Les entités principales de notre approche sont dénotées uniquement par des constantes dans plusieurs domaines, y compris les organisations, sujets, catégories, ressources et actions.

1. **Organisations** : une organisation (dénotées $org_0, org_1, org_2, \ldots$) peut être considérée comme un groupe d'activités ou un groupe de services. Par exemple dans le domaine du centre médical, l'organisation peut être la gestion clinique, l'administratif, le laboratoire ou bien les services des urgences, de radiologie, de chirurgie, etc.
2. **Sujets et Catégories** : L'entité sujet (dénotée $s_0, s_1, s_2, \ldots$) est une entité active (*i.e.* un utilisateur ou une organisation). Les sujets qui satisfont aux mêmes conditions, appartiennent au même groupe appelé « Catégorie » (dénotée $cat_0, cat_1, cat_2, \ldots$). Ils possèdent ainsi les mêmes permissions. L'entité catégorie est utilisée pour structurer le lien entre les sujets et les organisations.
3. **Ressources et Actions** : Des ressources (dénotés $r_0, r_1, r_2, \ldots$) sont des entités passives telles que des fichiers de données, un dossier médical d'un patient, une machine, une imprimante, etc. Des actions (dénotées $act_0, act_1, act_2, \ldots$) sont des opérations informatiques telles que « lire », « écrire », « envoyer », « imprimer », etc.

Pour simplifier la notation, nous ne considérons pas l'abstraction des ressources en vues et des actions en activités comme dans le modèle original de OrBAC, mais cela peut être facilement représenté en ajoutant deux entités supplémentaires et les relations correspondantes.

L'abstraction des sujets.

Dans le cas où chaque organisation possède sa propre politique de contrôle d'accès qui est gérée indépendamment, différents modèles de contrôle d'accès pourraient devoir cohabiter et interagir ; par exemple dans le cas de l'interopérabilité entre plusieurs organisations, le modèle RBAC classique a ses limites.

Afin d'augmenter la flexibilité et satisfaire l'interopérabilité entre des organisations, nous avons structuré dans notre approche les sujets en une entité plus générale appelée « **Catégorie** » selon la définition du modèle *Category-Based Access Control* (CBAC) [Bertolissi and Fernández, 2014]. Ceci est différent du modèle OrBAC original où les sujets sont structurés en rôles comme dans le modèle RBAC : un sujet est pré-assigné à un rôle ou à un ensemble de rôles, lesquels sont statistiquement associés à des permissions.

Définition 1.2: Catégorie [Bertolissi and Fernández, 2014]

Une catégorie est l'une des classes ou groupes distincts auxquels les sujets peuvent être affectés. Les catégories sont associées aux utilisateurs basés sur des attributs qu'ils possèdent, et les permissions sont attribuées à chaque catégorie. Le rôle peut être considéré comme un des attributs d'un utilisateur.

Dans le modèle CBAC, on vérifie si les attributs de l'utilisateur satisfont les conditions nécessaires pour accéder aux ressources auxquels il souhaite, au moment où il en demande l'accès. C'est ce qui constitue son avantage : les permissions sont acquises dynamiquement grâce aux attributs. C'est le contraire dans RBAC, qui ne peut pas traiter les changements dynamiques des attributs tels que l'heure et l'emplacement. De plus, les permissions y sont calculées à partir des rôles qui sont pré-associés aux utilisateurs. Cela nécessite de bien concevoir les rôles avant de les appliquer dans RBAC.

Les prédicats.

Nous avons présenté les entités principales y compris la nouvelle entité « *catégorie* » de notre modèle dans la section précédente, nous présentons maintenant les prédicats du modèle D-OrBAC.

Définition 1.3: Empower

Le prédicat **Empower(org, cat, s)** représente le lien entre les organisations, les catégories et les sujets. Cela signifie que dans l'organisation org, le sujet s est associé à la catégorie cat.

Notons que le prédicat « *Empower* » du modèle D-OrBAC représente une extension du prédicat « *Empower* » du modèle OrBAC original où il représente la relation entre des sujets et des rôles dans des organisations.

En conformité avec le modèle entité-relation et l'affectation sujet-catégorie, des attributs peuvent être associés aux entités et aux relations. Nous pouvons donc modéliser des attributs et leur valeur par l'ensemble de prédicats binaires sous forme de **attribute_name(entity, value)**. Puisque dans notre modèle, seuls les attributs de sujets sont considérés et cet ensemble d'attributs est en relation avec l'organisation, $entity$ sera donc un sujet et $value$ représentera la valeur de $attribute\_name$ pour l'entité $entity$ dans son organisation. Afin d'éviter l'ambiguïté sur la signification des attributs dans les différentes organisation, nous ajoutons l'acronyme de chaque organisation devant ces prédicats binaires.

Nous pouvons maintenant modéliser l'affectation sujet-catégorie basée sur les attributs de sujet comme ci-dessous :

$$\begin{aligned}\forall org, \forall cat, \forall s, \exists [val_1 \in d_1, val_2 \in d_2, \cdots, val_n \in d_n] \\ Empower(org, s, cat) \leftarrow org\_attr_1(s, val_1) * \\ org\_attr_2(s, val_2) * \cdots * org\_attr_n(s, val_n) \quad (1.1)\end{aligned}$$

Où d_i est un ensemble des valeurs possibles pour chaque attribut org_attr_i et * peut être une disjonction ($\vee$), une conjonction ($\wedge$) ou une combinaison des deux.

Exemple 1.1. *Dans l'organisation CM, un sujet s est affecté à la catégorie CM_doctor si s possède l'attribut diplôme égal à médecine ou l'attribut spécialité égal à docteur ou dentist. Cette affectation est représentée comme ci-dessous :*

$$\begin{aligned}Empower(CM, s, CM\_Doctor) \leftarrow CM\_diploma(s, medecine) \vee \\ CM\_speciality(s, docteur) \vee CM\_speciality(s, dentist)\end{aligned}$$

Définition 1.4: Permission

Le prédicat **Permission(org, cat, act, r)** spécifie l'abstraction les permissions en reliant la catégorie cat, l'action act et la ressource r dans l'organisation org.

Pour la simplicité, nous ne traitons que les permissions en considérant que les mêmes raisonnements peuvent s'appliquer aux interdictions, aux obligations ou aux recommandations. La relation $permission$ permet à l'administrateur du système de définir une politique de sécurité afin d'en réglementer les accès. Cela représente l'autorisation donnée à l'utilisateur appartenant à la catégorie cat d'effectuer l'action act sur la ressource r dans l'organisation org.

Exemple 1.2. *Dans l'hôpital H_1, les permissions sont définies comme la suivante :*

1. $Permission(H_1, H_1\_Doctor, read, patientRecords)$
2. $Permission(H_1, H_1\_Accounting, create, invoices)$

La signification : Dans l'hôpital H_1,

1. *l'utilisateur appartenant à la catégorie H_1_Doctor peut lire les dossiers médicaux des patients*
2. *l'utilisateur appartenant à la catégorie $H_1\_Accounting$ peut créer des factures*

Le prédicat *Permission* nous permet de modéliser les permissions abstraites décrivant le fait que les catégories ont autorisé à effectuer telles actions sur telles ressources. Cependant, le contrôle d'accès doit pouvoir décrire les actions concrètes que réalisent les sujets sur les ressources. Le prédicat *Is_permitted* ou *Est_permis* en français a donc été introduit dans notre modèle pour représenter les permissions concrètes.

Définition 1.5: Is_permitted

Le prédicat **Is_permitted(s, act, r)** nous permet de décrire la demande d'accès et dériver des permissions concrètes entre le sujet s, l'action act et la ressource r.

Ce prédicat possède deux utilités : d'une part, du point de vue d'un administrateur de sécurité du système, il lui permet de définir la dérivation des permissions concrètes à partir des permissions abstraites. D'autre part, du point de vue des utilisateurs, le prédicat *Is_permitted* représente une requête d'accès correspondant à la question « Est-ce que l'utilisateur s peut réaliser l'action act sur la ressource r ? ».

Dans le modèle OrBAC original, chaque règle de sécurité ne peut s'appliquer qu'à une seule organisation, ce qui rend impossible le traitement des demandes d'accès où plusieurs organisations sont impliquées. Afin de surmonter cette limite, nous introduisons dans le modèle D-OrBAC la notion de délégations qui littéralement signifie « *l'opération par laquelle une personne (le délégant) donne le contrôle, l'autorité, un privilège, un travail, etc., à une autre personne (le délégué)* »[1]. La définition standard de la délégation dans le monde informatique correspond au fait qu'un utilisateur délègue son privilège ou son attribut à quelqu'un autre et le délégué peut aussi déléguer ce privilège à une

1. Merriam-Webster Dictionary

troisième personne. La délégation se fait au niveau des utilisateurs et c'est toujours le même privilège qu'ils se partagent, mais dans notre modèle, la délégation se fait au niveau des organisations et elle n'est visible que pour le déléguant et le délégué. Nous n'autorisons les organisations à déléguer que la catégorie, et elles ne peuvent pas la transmettre à quelqu'un d'autre.

Définition 1.6: Délégation

Le prédicat **Delegation(org1, cat1, org2, cat2)** nous permet de structurer le lien entre les catégories de deux organisations différentes. Il signifie que l'organisation $org1$ délègue la catégorie $cat1$ à la catégorie $cat2$ de l'organisation $org2$

Une politique de délégation est définie pour spécifier les correspondances des catégories appartenant à différents domaines d'autorisation. Ceci est formalisé par un graphe de la délégation sous forme d'un graphe orienté acyclique (DAG) [plus de détails dans la section 1.4.1]. Le graphe de délégation est très important pour déterminer des demandes d'accès dans le cas de l'invocation transitive de services. C'est ainsi que les organisations doivent se mettre d'accord sur la définition du graphe de délégations pour qu'elles puissent coopérer entre elles correctement.

Exemple 1.3. *Prenons un exemple sur le centre médical, il y a une délégation entre* $ClinicalManagement$ *et* $Laboratory$ *comme la suivante :*

— $Delegation(LA, LA\_Clinicain, CM, CM\_Doctor)$

Cela signifie que $Laboratory$ *délègue la catégorie* $LA\_Clinicain$ *à la catégorie* CM_Doctor *de* $ClinicalManagement$.

Les prédicats utilisés dans notre approche pour modéliser les associations entre des entités sont récapitulés dans la table 1.1

1.3.2 Règles de sécurité

Dans le modèle D-OrBAC, nous avons pris en compte la requête d'accès local, c'est-à-dire que les utilisateurs demandent l'accès aux ressources appartenant à la même organisation. Nous traitons également la requête d'accès transitive qui concerne la demande l'accès aux ressources appartenant à l'organisation différente de celle de l'utilisateur. En utilisant les entités et les prédicats présentés dans la section précédente, nous pouvons maintenant définir les politiques de sécurité pour les appliquer aux différentes organisations dans les deux cas.

Prédicat	Domaine	Description
$Permission$	$Org \times$ $Category \times$ $Action \times$ $Resource$	si org est une organisation, c une catégorie, act une action, et r une ressource, alors $Permission(org, c, act, r)$ signifie que dans l'organisation org, la catégorie c est autorisée à effectuer l'action act sur la ressource r. Ex. : *Permission(France, FrMajor18, watch, FilmX)* signifie que en France, un adulte est autorisé à regarder une vidéo pour adultes
$Is\_permitted$	$Subject \times$ $Action \times$ $Resource$	si s est un sujet, act une action et r une ressource, alors $Is\_permitted(s, act, r)$ signifie que s a la permission d'effectuer l'action act sur la ressource r Ex. : *Is_permitted(Bob, watch, FilmX)* signifie que Bob a la permission de regarder une vidéo pour adultes
$Empower$	$Org \times Subject \times$ $Category \times$ $Condition$	si org est une organisation, s un sujet, c une catégorie and $cond$ une condition, alors $Empower(org, s, c, cond)$ signifie que org habilite le sujet s à la catégorie c si $cond$ est satisfaite. Ex. : *Empower(France, Bob, FrMajor18, cond)* signifie que la France habilite Bob en tant qu'adulte si $cond$ est satisfaite où $cond \leftarrow (Country(Bob), =, France) \wedge (Age(Bob), \geq, 18)$
$Delegate$	$Organization \times$ $Category \times$ $Organization \times$ $Category$	si $org1, org2$ sont des organisations et $c1, c2$ sont des catégories, alors $Delegate(org1, c1, org2, c2)$ signifie que l'on associe la catégorie $c1$ de l'organisation $org1$ à la catégorie $c2$ de l'organisation $org2$. Ex. : *Delegate(France, FrMajor18, Japan, JpMajor21)* signifie que l'on associe la catégorie $FrMajor18$ de la France à la catégorie $JpMajor21$ du Japon.

TABLE 1.1 – Prédicats de base du modèle D-OrBAC

Dans le cas local, il suffit de regarder dans son organisation à quelle catégorie l'utilisateur appartient et vérifier si sa catégorie est autorisée à effectuer l'action qu'il demande sur la ressource donnée.

Définition 1.7: Politique de sécurité (local)

Dans une organisation org, un sujet s a une permission de réaliser une action act sur une ressource r si :

1. s est associé à une catégorie cat dans une organisation org et
2. l'organisation org accorde une permission à la catégorie cat d'effectuer l'action act sur la ressource r.

Si et seulement si les deux conditions ci-dessus sont satisfaites, alors la demande d'accès d'un sujet s à effectuer une action act sur l'objet o sera acceptée. Sinon la demande d'accès sera rejetée. La dérivation des permissions est modélisée par la règle ci-dessous :

$$\forall org, \forall cat, \forall s, \forall r, \forall act,$$
$$Is\_permitted(s, act, r) \leftarrow$$
$$Permission(org, cat, act, o) \wedge Empower(org, s, cat) \tag{1.2}$$

Exemple 1.4. *L'organisation CM (Clinical Management) autorise la catégorie CM_Doctor à effectuer l'action call sur le service $CareOrders\_service$. L'organisation CM assigne le sujet Bob à la catégorie CM_Doctor, dans ce contexte Bob a le droit d'appeler le service $CareOrders\_service$. La dérivation de cette permission est la suivante :*

$$Is\_permitted(Bob, call, CareOrders\_service) \leftarrow$$
$$Permission(CM, CM\_Doctor, call, CareOrders\_service)$$
$$\wedge Empower(CM, Bob, CM\_Doctor)$$

Nous commençons par vérifier l'organisation de Bob et de $CareOrders\_service$, puis récupérons la catégorie à laquelle Bob appartient dans cette organisation. Quand nous avons toutes les entités nécessaires, nous évaluons les prédicats $Permission$ et $Empower$. Si la condition est satisfaite alors Bob est autorisé à appeler le service $CareOrders\_service$.

Afin de modéliser la politique de sécurité dans le cas d'un appel transitif où plusieurs organisations ont été impliquées, nous utilisons le prédicat

$Delegation$ pour représenter l'accord de la délégation entre deux organisations ou plus. Enfin, nous déterminons la permission en utilisant la nouvelle catégorie qui a été déléguée par son organisation partenaire.

Définition 1.8: Politique de sécurité (transitive)

Dans un système distribué où chaque organisation possède un domaine d'autorisation différent, le sujet s de l'organisation $org2$ a la permission d'effectuer l'action act sur la ressource r appartenant à l'organisation $org1$ si :

1. s est associé à la catégorie $cat2$ dans l'organisation $org2$,
2. l'organisation $org1$ délègue la catégorie $cat1$ à la catégorie $cat2$ de l'organisation $org2$ et
3. l'organisation $org1$ accorde à la catégorie $cat1$, la permission d'effectuer l'action act sur la ressource r.

La dérivation des permissions dans le cas d'un service invoquant un autre service hors de son organisation, est modélisée par la règle suivante :

$$\begin{aligned}\forall org1, \forall org2, \forall cat1, \forall cat2, \forall s, \forall r, \forall act,\\ Is\_permitted(s, act, r) \leftarrow\\ Empower(org2, s, cat2) \wedge\\ Permission(org1, cat1, act, r) \wedge\\ Delegate(org1, cat1, org2, cat2) \quad (1.3)\end{aligned}$$

Exemple 1.5. *L'organisation CM assigne le sujet Bob à la catégorie* CM_Doctor*, l'organisation LA autorise la catégorie* $LA\_Clinician$ *à effectuer l'action call sur le service* $TestOrders\_service$*. Si l'organisation LA délègue la catégorie* $LA\_clinician$ *à la catégorie* CM_doctor *de l'organisation CM, alors Bob a le droit d'appeler le service* $TestOrders\_service$*. La représentation de cette permission est la suivante :*

$$\begin{aligned}Is\_permitted(Bob, call, TestOrders\_service) \leftarrow\\ Empower(CM, Bob, CM\_Doctor) \wedge\\ Permission(LA, LA\_Clinician, call, TestOrders\_service) \wedge\\ Delegate(LA, LA\_Clinician, CM, CM\_Doctor)\end{aligned}$$

Nous commençons par vérifier l'organisation de Bob et de $TestOrders\_service$*, puis récupérons la catégorie à laquelle Bob appartient dans cette organisation. Quand nous avons toutes les entités nécessaires, nous évaluons les prédicats* $Permission$ *et* $Empower$*. Comme l'organisation de Bob est différente de celle du service* $TestOrders\_service$*, nous devons vérifier s'il y a une délégation entre* $ClinicalManagement$ *et* $Laboratory$*. S'il y en a une et que la condition d'accès est satisfaite, alors Bob est autorisé à appeler le service* $TestOrders\_service$*.*

Notons que dans le cas où un même utilisateur est reconnu dans plusieurs organisations, le choix de catégorie que nous utilisons pour évaluer la requête, est déterminé par l'organisation de la ressource dans le cas d'un appel local. Par contre, dans le cas d'un appel transitif, nous récupérons toutes les catégories auxquelles un utilisateur donné est assigné, puis nous éliminons toutes les catégories qui ne sont pas partenaires (pas de la délégation) avec l'organisation de la ressource.

1.4 Délégation dans le modèle D-OrBAC

1.4.1 Graphe de délégation

Dans un contexte du système distribué (*i.e.* le service web) où chacun a conçu, mis en œuvre et géré indépendamment son système, plusieurs problèmes cruciaux ont été identifiés ; par exemple la disponibilité des contenus, l'intégrité des données et l'interopérabilité entre différents fournisseurs de services. Un autre problème crucial dans ce contexte à résoudre est la présence des domaines multiples ; chaque service ayant sa propre politique de contrôle d'accès, les utilisateurs peuvent ne pas être reconnus dans tous les domaines. Pour résoudre ce problème, nous avons proposé de déléguer les catégories d'un domaine à l'autre. Pour cela, nous avons défini un graphe de délégation.

Définition 1.9: Graphe orienté acyclique : DAG

Un graphe orienté $G(V, E)$ est une paire ordonnée (V,E) telle que :

- V est un ensemble fini de sommets.
- E est un ensemble de paires ordonnées.
- Si $e \in E$, alors $e = (X_i, X_j)$ et $\{X_i, X_j\} \subseteq V$ où X_i est le sommet initial, et X_j le sommet terminal.

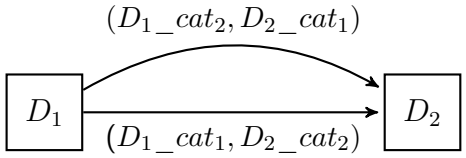
— Un couple (X_i, X_j) est appelé un arc, et représenté graphiquement par $X_i \rightarrow X_j$.
— Un graphe orienté G est acyclique si G ne contient pas le cycle.

Définition 1.10: Graphe de délégation

Un graphe de délégation $D(V, E)$ est un DAG, qui permet de décrire qui (*i.e.* le délégant) a le droit de déléguer quoi (*i.e.* une catégorie ou un attribut) à qui (*i.e.* le délégué).

— V est un ensemble fini de sommets où chaque sommet représente un domaine, possédant la catégorie initiale, pouvant déléguer ses catégories aux autres.
— E est un ensemble d'arcs, chaque arc représente une catégorie qui a été déléguée du délégant au délégué.
— La direction de l'arc indique le sens de délégation, chaque arc est étiqueté par une paire de catégories appartenant à deux domaines partenaires.
— Il est possible également d'avoir plus d'un arc partant du même sommet départ, se dirigeant vers le même sommet cible ; ils doivent cependant être étiquetés par différentes étiquettes.

Exemple 1.6. *Graphe de délégations pour l'organisation D_1 et D_2 représenté comme ci-dessous :*



$D_1 \xrightarrow{(D_1\_cat_1, D_2\_cat_2)} D_2$ *signifie que si un utilisateur du domaine D_1 appartenant à la catégorie $D_1\_cat_1$ invoque un service appartenant au domaine D_2, on lui délègue la catégorie $D_2\_cat_2$.*

Dans le cas d'un appel local où l'utilisateur et la ressource appartiennent à la même organisation, nous n'avons pas besoin d'utiliser le graphe de délégation pour déterminer l'autorisation que l'utilisateur a demandé car il n'y a pas de délégation. Par contre, l'utilité du graphe de délégations est très importante dans le cas d'un appel transitif.

Exemple 1.7. *Prenons un scénario de notre cas d'étude sur la clinique [voir figure 1.1], le service PatientMedData du domaine Web portal invoque le service Care Orders appartenant au domaine Clinical Management. Afin de compléter sa tâche, le service Care Orders doit appeler le service Test Order appartenant au domaine Laboratory de la part du service PatientMedData. Par conséquent, pour satisfaire la requête initiale, nous avons besoin des délégations entre Web portal et Clinical Management et entre Clinical Management et Laboratory.*

On peut trouver un sous-graphe de délégation (les arcs en rouge) qui permet de résoudre le problème d'appel transitif pour cette chaîne d'invocation dans le graphe de délégation illustré dans la figure 1.4. Quand le service PatientMedData appelle le service Care Orders, le graphe de délégation doit être vérifié s'il existe un accord entre le domaine de l'utilisateur et le domaine du service appelé pour la délégation, c'est-à-dire s'il existe un arc partant du sommet CM au sommet WP.

Dans notre cas, il y a eu un arc $CM \xrightarrow{(WP\_Doctor, CM\_Docteur)} WP$, cela signifie que, quand un utilisateur appartenant à la catégorie WP_Doctor du domaine Web portal invoque un service dans le domaine Clinical Management, on lui délègue la catégorie $CM\_Docteur$. Si cet utilisateur est autorisé à appeler le service Care Orders, il invoquera le service Test Order et le module d'autorisation vérifiera la délégation entre Clinical Management et Laboratory, puis il prendra une décision d'accès. Sinon la requête se terminera au service Care Orders.

Dans notre travail, nous nous sommes restreints à l'association simple des catégories (une-à-une) en raison de la difficulté à obtenir un consensus de l'association des catégories (ou attributs) entre deux domaines ou plus. Ce problème est considéré difficile car des organisations peuvent avoir des catégories de noms différents mais avec une même signification ou bien avoir des catégories avec le même nom mais pas la même signification. Dans les travaux sur l'association des attributs d'un domaine à un autre, ce qui est fait généralement et particulièrement dans [V. C. Hu, 2014; Long et al., 2010], c'est une association un-à-un, c'est-à-dire un attribut vers un attribut. Mais dans notre cas, l'association est plus générale puisque nous associons une catégorie à une autre catégorie, c'est à dire l'association d'une combinaison d'attributs vers une autre combinaison.

On pourrait imaginer un graphe de délégation plus générique qui prenne en compte la délégation de catégories multiples, c'est-à-dire qu'à la place d'associer une catégorie d'un domaine à une catégorie d'un autre domaine, nous pouvons imaginer une association d'un tuple de catégories avec un autre tuple de catégories. Nous pouvons également considérer des opérateurs logiques pour la combinaison des catégories dans le tuple et créer ainsi un langage

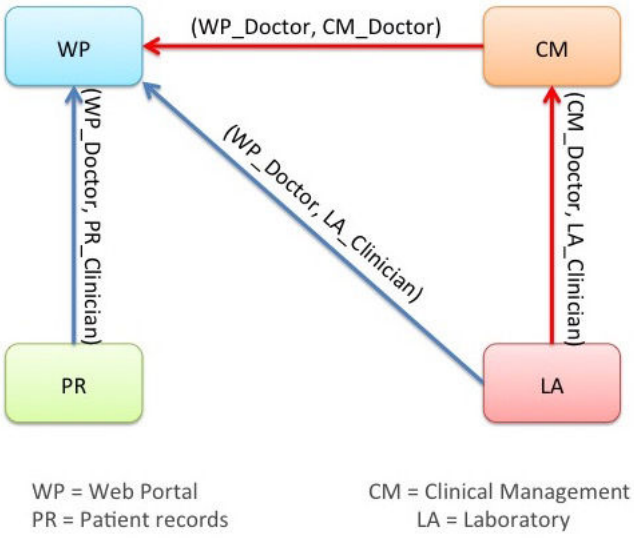


FIGURE 1.4 – Graphe de délégation.

assez expressif pour spécifier une large gamme de délégation.

Exemple 1.8. *Dans l'hôpital H_1, la personne qui appartient à la catégorie $H_1\_infirmer$ et H_1_senior, se voit déléguer la catégorie correspondante à $H_2\_cadre\_infirmer$ quand elle arrive à l'hôpital H_2. La représentation de cette délégation est comme ci-dessous :*

$$H_2 \xrightarrow{[(H_1\_infirmer, H_1\_senior), (H_2\_cadre\_infirmer)]} H_1$$

Exemple 1.9. *L'utilisateur de l'hôpital H_1 qui appartient à la catégorie $H_1\_medecin\_intern$ ou $H_1\_dentist$, se voit déléguer correspondante à $H_2\_medecin$ dans l'hôpital H_2. La représentation de cette délégation est suivant :*

$$H_2 \xrightarrow{[(H_1\_medecin\_intern \vee H_1\_dentist), (H_2\_medecin)]} H_1$$

Discussion

Le graphe de délégation est statiquement définit au préalable selon un accord entre les différents participants. Il sera consulté par le module de délégation au moment où l'application de contrôle d'accès aura détecté qu'un sujet est en train d'invoquer un service d'une autre organisation afin de décider si la requête d'accès est acceptée. Nous ne voulons pas *a priori* avoir de cycle

dans le graphe de délégation car un utilisateur pourrait alors obtenir plus de privilège que nécessaire grâce au cycle. De plus, un cycle pourrait changer la sémantique de l'autorisation du système.
Cependant, si nous sommes confronté à un problème de cycle, nous pouvons le traiter par différentes méthodes en voyant les choses en deux façons, soit la façon « design-time » qui correspond à notre formalisation, soit la façon « run-time » qui correspond à l'exécution de requête de la part du système.

1. Même si nous avons un cycle dans le graphe de délégation à *design-time*, nous ne l'avons pas forcément à *run-time*, cela dépend des requêtes d'accès qui sont exécutées.

2. Si jamais nous avons un cycle à *run-time*, nous pouvons spécifier un arbre de délégation pour chaque chemin d'exécution, ainsi le cycle est rompu [voir la figure 1.5 A où chaque chemin apparait d'une couleur différente]. Il ne posera donc plus de problème lors de l'exécution puisque la boucle ne peut se produire qu'avec la fusion d'au moins deux arbres de délégation.

3. Il est possible que le chemin d'exécution revienne dans le même domaine, mais ce serait alors pour appeler un autre service ou une autre méthode. Pour éliminer le cycle, nous pouvons définir de façon plus fine les domaines en les découpant en services. Dans ce cas, les sommets du graphe représenteront des services tandis que les arcs représenteront les délégations de catégories d'un domaine pour accéder à un service concerné dans l'autre domaine [voir la figure 1.5 B]. Cela peut permettre d'éliminer une boucle dans le cas où différents services d'un même domaine soient appelés. Il n'est pas vraisemblable que le programme se rappelle lui-même ou rappelle le même service et produise ainsi une boucle dans le chemin d'exécution ; si tel était le cas, c'est que le programme lui-même n'aurait pas été bien défini.

Dans le monde réel, les entreprises ou les organisations sont souvent hiérarchisées (par exemple, la hiérarchie militaire, l'ancienneté au travail ou la hiérarchie de poste, etc.). Ainsi dans notre cas de la clinique [figure 1.1], il semble être logique de supposer qu'il y ait une hiérarchie entre les domaines [voir la figure 1.6]. Dans le modèle D-OrBAC, nous n'autorisons la délégation que depuis la racine (hiérarchie) vers le bas ; c'est-à-dire qu'un domaine ne pourra déléguer des attributs qu'à ceux situés en dessous. Nous pouvons utiliser ce principe de hiérarchie pour garantir l'absence de cycles dans notre graphe de délégation. Dans ce cas, il n'y a pas de délégation dans les deux sens, et cela nous évitera une boucle dans la chaîne de délégation.

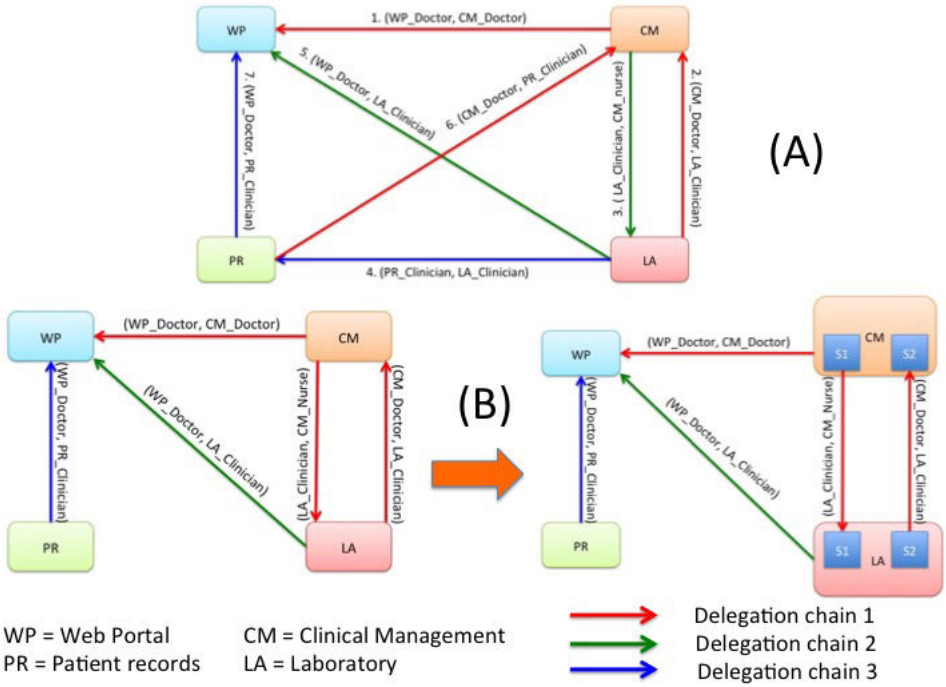


FIGURE 1.5 – Cycles dans le graphe de délégation.

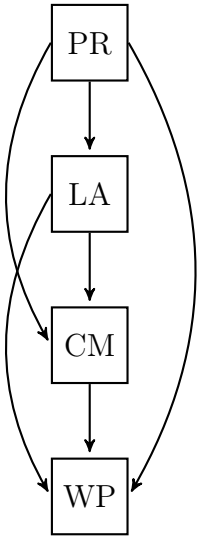


FIGURE 1.6 – Hiérarchie des domaines.

1.4.2 Complexité de recherche d'une délégation

La vérification de l'existence d'une délégation entre un domaine et un autre correspond au problème de recherche d'un chemin de longueur 1 dans un

graphe orienté. Il existe un algorithme générique très connu pour chercher le chemin de longueur quelconque dans un graphe [Sedgewick and Wayne, 2011]. C'est l'algorithme de parcours en profondeur (Depth First Search : DFS). Le principe de cet algorithme est :

— nous commençons par un sommet quelconque s, puis listons tous ses voisins pour pouvoir explorer à fond les chemins un par un.

— si le graphe contient n sommets et p arêtes, alors la complexité de DFS sera en $O(n+p)$ dans le cas où le graphe est représenté sous forme d'une liste d'adjacence.

Dans notre cas où le graphe de délégation est un DAG et les arcs étiquetés par les paires de catégories, si nous voulons parcourir le chemin de longueur 1(qui correspond à la délégation d'une étape), nous n'avons pas besoin de parcourir récursivement tous les voisins des voisins. En effet, une fois que nous obtenons la liste des voisins et comme nous connaissons les sommets de départ et d'arrivé, nous vérifions simplement dans cette liste s'il existe un arc partant du sommet s au sommet correspondant à l'organisation partenaire. Notons que même s'il existe un arc reliant deux organisations partenaires, si l'étiquette ne correspond pas à la délégation de catégories, nous considérons qu'il n'existe pas d'arc. La complexité de cet algorithme sera en $O(p)$ où p est le nombre d'arcs.

Dans le cas où nous avons une chaine de délégation de longueur l pour un appel transitif, les organisations délèguent successivement leur catégorie aux autres pour pouvoir satisfaire la requête ; nous pouvons utiliser l'algorithme DSF pour parcourir la chaine et vérifier les voisins des voisins en profondeur. La complexité sera en $O(n + p)$ où n est le nombre de sommets et p est le nombre d'arcs.

1.5 Travaux liés

Dans [Chadwick et al., 2009], les auteurs ont présenté la notion de délégation dynamique d'autorité d'un utilisateur à un autre, c'est-à-dire qu'un utilisateur avec un privilège (le déléguant) octroie son privilège à un autre utilisateur (le délégué). Ils utilisent un graphe orienté acyclique pour représenter la hiérarchie de délégation quand un utilisateur délègue un attribut (le privilège) à un autre utilisateur, qui ensuite le délègue à un autre récursivement. Le sommet racine représente un utilisateur ayant initialement cet attribut. Les sommets intermédiaires représentent des délégués qui par la suite agissent comme des délégants et sous-délèguent leurs attributs (ou permissions) aux

autres. Les arêtes dans le graphe représentent les attributs ou permissions qui ont été délégués.

Nous avons adapté la représentation de la délégation montrée par Chadwick et al. pour modéliser notre chaîne de délégation à un niveau plus général (des attributs sont délégués au niveau des domaines alors que dans le cas de Chadwick, la délégation est faite entre des utilisateurs). Dans leur contexte, un utilisateur peut recevoir des attributs par différents distributeurs d'attributs. Il peut déléguer son attribut à autre utilisateur et lui aussi peut ensuite sous-déléguer le même attribut à quelqu'un d'autre dans la même organisation (c'est toujours le même attribut qui se partage). C'est ainsi que les arêtes successeurs doivent toujours avoir les mêmes ou moins d'attributs et de permissions que l'union des arêtes précédentes. Leur graphe est acyclique, car un déléguant n'a pas besoin de déléguer des attributs à lui-même ou à son prédécesseur.

Contrairement à notre contexte, la délégation est faite au niveau inter-organisations ; chaque organisation décide quels attributs peuvent être délégués à un utilisateur provenant de son organisation partenaire. L'utilisateur de chaque organisation possède des attributs différents qui ne sont peut-être pas pris en compte dans les autres organisations ou bien ces attributs n'y auront pas les mêmes définitions. Les délégations d'attributs sont accordées par deux organisations différentes et elles ne délèguent pas tout le temps le même attribut aux autres organisations. De plus, les organisations ne connaissent pas les délégations que leurs partenaires ont octroyé aux autres. Il est donc possible que les nœuds intermédiaires puissent obtenir plus d'attributs ou de permissions que leurs prédécesseurs. Il est aussi possible d'avoir un cycle dans notre graphe de délégation.

Afin de compléter la politique d'association des attributs entre différents domaines, notre travail est inspiré par [Fischer and Majumdar, 2008]. Leur solution est de définir le schéma de rôles globaux. Il regroupe de manière automatique les rôles locaux de chaque service en rôles globaux afin qu'un utilisateur donné ait toutes les autorisations d'accès nécessaires pour récupérer les données de chaque service car il considère que si un utilisateur possède l'un des rôles requis pour le service initial, il aura le rôle requis pour chaque service appelé dans la chaine d'exécution. Pour trouver la compatibilité globale de rôle, il doit garantir que la sémantique de l'autorisation d'une application ne sera pas modifiée par l'association global. Il est donc interdit d'associer deux rôles locaux du même domaine au même rôle global. De plus, l'association global de rôle ne devrait pas donner aux utilisateurs des droits d'accès supérieurs à ceux nécessaires pour atteindre leurs objectifs.

Leur approche peut bien fonctionner dans le cas où nous connaissons bien

la topologie du système et la définition des rôles locaux de chaque composant pour pouvoir associer correctement les rôles. Il est nécessaire de la connaitre car nous devons vérifier si nous donnons des droits supérieurs à ceux nécessaires à un utilisateur pour chaque service appelé. Nous pourrions utiliser leur technique pour établir notre graphe de délégation si nous avions la vision globale du système et connaissions tous les chemins d'exécution possible, mais pour le moment nous travaillons uniquement sur l'association par paires de domaines. En effet, dans notre contexte, chaque composant de notre système est indépendant. Il connait uniquement quel service doit être invoqué dans son domaine partenaire mais il ne sait pas si le service appelé doit en invoquer d'autres ou non. Les invocations de services dans les autres domaines sont donc transparentes. Chaque association de rôle a été fait selon un accord de deux composants et est invisible par les autres. Ces raisons font que nous ne pouvons pas trouver un rôle global capable de regrouper les rôles locaux de chaque partenaire pour qu'il puisse satisfaire à toutes les invocations et atteindre le dernier service de la chaine sans connaitre la topologie globale du système et tout le chemin d'exécution possible.

Dans [Y. Deswarte, 2009], les auteurs ont proposé une extension de l'OrBAC afin de spécifier des règles de sécurité pour les services Web inter-organisationnels. Ils modélisent des permissions, interdictions et obligations en automates temporisés pour vérifier des propriétés telles que l'accessibilité et la correction. Ils considèrent le cas d'un service demandé à distance à partir d'une organisation différente. Dans ce cas, afin d'autoriser un utilisateur d'un domaine différent à accéder à un service, un rôle particulier sera associé à un utilisateur virtuel, puis une règle spécifique définie comme une circonstance (la relation « contexte » dans OrBAC) sera appliquée. Cependant, ils ne traitent pas le problème de l'accès transitif pour les services dépendants, ni l'utilisation d'attributs de demandeurs pour calculer leurs droits d'accès.

Dans [Baina and Laarouchi, 2012], les auteurs ont proposé une autre extension du modèle OrBAC. Ils se sont concentrés sur le problème de l'intégrité des données dans un système d'information complexe. Ils étendent le modèle OrBAC original par le modèle d'intégrité de Totel [Totel et al., 1998] en ajoutant différents niveaux d'intégrité sur les sujets et les ressources, puis ils ont défini les règles de flux d'information pour éviter la propagation d'erreur parvenant du bas niveau d'intégrité au niveau plus élevé. Enfin, la décision d'accès est basée sur les niveaux d'intégrité et le flux d'information. Afin de prendre en compte le niveau d'intégrité dans le modèle OrBAC de base, les auteurs ont travaillé sur la notion de $Context$ pour accroitre son expressivité en ajoutant les différentes conditions d'accès basées sur le niveau d'intégrité (comparaison le niveau d'intégrité du sujet avec celui de la ressource).

Chapitre 2

Validation du modèle D-OrBAC

Sommaire

Dans le chapitre précédent, nous avons présenté notre extension du modèle de contrôle d'accès basé sur les organisations (D-OrBAC), et l'avons adapté pour la collaboration de différentes organisations dans un environnement distribué où des services peuvent échanger des données entre eux. Puis, nous avons spécifié cette extension en utilisant des prédicats logiques du premier ordre. Avant de passer à l'implémentation et au déploiement dans les prochains chapitres, nous présentons ici la validation du modèle. Nous utilisons une technique formelle pour effectuer l'analyse automatique qui nous permet d'identifier des erreurs (*e.g.* une mauvaise définition de politique de contrôle d'accès dans laquelle l'utilisateur obtient plus de privilèges que nécessaire) en regardant le comportement du système via la simulation des différents scénarios.

L'approche que nous avons choisi pour effectuer une analyse automatique, ne nécessite pas d'implémenter un prototype pour surveiller le fonctionnement du système, car il est possible d'en effectuer la simulation grâce à la conception abstraite. Par conséquent, cela nous aide à réduire les coûts de correction des erreurs de l'implémentation quand le système est déjà mis en

place et en particulier, cela nous permet d'éviter les difficultés des tests des applications dans le cas du système distribué. Parfois, le code de certains services intégrés est long et n'est pas facilement accessible, ou cela peut coûter cher pour les invoquer. De plus, il n'est pas facile d'accéder à l'environnement dans lequel les différents services sont exécutés, il se peut que cela rende le test du système composé plus difficile. Il est en effet souhaitable d'être capable de prédire les résultats en utilisant la spécification abstraite du système sans encourir de coûts supplémentaires pour faire des tests.

Afin d'expérimenter les comportements du système avant son déploiement, nous traduisons les prédicats logiques de notre modèle en Datalog (un sous-ensemble de Prolog), qui est un langage de programmation logique adapté au calcul symbolique et à la manipulation de prédicats et de relations. Ensuite, nous utilisons un interpréteur de Datalog pour effectuer la simulation et l'analyse automatique de notre modèle. Il nous aide à identifier les erreurs d'évaluation des requêtes d'accès et à assurer que l'évaluation se termine toujours, ceci étant une des propriétés importantes de la politique de contrôle d'accès.

2.1 Préliminaires

2.1.1 Prolog / Datalog

Prolog (PROgramming in LOGic) [Bratko, 2001] est un langage de programmation logique basé sur le calcul des prédicats du premier ordre. C'est un langage de programmation adapté au calcul symbolique et à la manipulation de prédicats et de relations. Il n'y a ni instruction, ni affectation, ni boucles explicite. On y trouve uniquement des clauses qui expriment des faits, des règles et des questions. Le premier interpréteur de Prolog a été implémenté à l'Université de Marseille en 1972 par Alain Colmerauer. Depuis, de nombreux interpréteurs ont été écrits avec différentes syntaxes par exemple SWI-Prolog, C-Prolog et Sixtus-Prolog.

Les langages de programmation traditionnels sont généralement procéduraux. C'est à dire que le programme contient une série d'instructions à exécuter les unes après les autres. Ce n'est pas le cas de Prolog qui est un langage de programmation déclarative. Cela signifie que, lors de l'implémentation de la solution d'un problème, au lieu de spécifier comment atteindre un certain objectif dans une certaine situation, nous définissons ce que la situation (des règles et des faits) et l'objectif (requête) doivent être et laissons l'interpréteur Prolog nous fournir la solution (par la dérivation). En Prolog il n'y a que deux composants pour tous les programmes : des règles et des faits. Le système

Prolog lit dans le programme et mémorise tout simplement. L'utilisateur entre ensuite une série de questions (des requêtes), le système répond en utilisant les faits et les règles dont il dispose. Grâce à sa concision et sa simplicité, il est devenu très populaire dans plusieurs domaines : la logique formelle associée à la programmation, la modélisation de raisonnement, la programmation de bases de données, etc.

Quant à Datalog, c'est un langage de requête non procédural basé sur le langage de programmation logique (Prolog). Il simplifie l'écriture de requêtes et rend l'optimisation des requêtes plus facile. Il est souvent utilisé comme un langage de requête pour bases de données déductives. Des programmes en Prolog/Datalog sont équivalents à des bases de données de faits et de règles. Le backtracking automatique est intégré dans un prouveur de théorème. Il est aisé de modifier des programmes ou des bases de données durant l'exécution. Pour aller plus loin sur Datalog, cf. [Abiteboul et al., 1995; Ramakrishnan and Gehrke, 2003].

Caractéristiques de base de Datalog

Un programme Datalog est un ensemble fini de **clauses**. Une clause est soit un **fait** de la forme « tête. » soit une **règle** de la forme « tête :-corps. »

Les **faits** décrivent les propriétés des objets ou des relations entre les objets. Un fait est un prédicat logique suivi par un point final « . » par exemple :

— *role(Alice, Nurse)*. signifie que Alice joue le rôle d'infirmière.

— *role(Bob, Doctor)*. signifie que Bob joue le rôle de docteur.

Les **règles** permettent de dériver une nouvelle propriété ou une nouvelle relation d'un ensemble de celles qui existent déjà. Une règle se compose d'une tête (un prédicat) et d'un corps (une séquence de prédicats séparés par des virgules). La tête et le corps sont séparés par le signe « : - » et, comme pour chaque expression Datalog, une règle doit être terminée par un point final « . » . Par exemple :

— *son(Bob, David) :- father(David, Bob), male(Bob)*.
signifie que Bob est un fils de David si David est le père de Bob et Bob est masculin

— *son(Bob, Sylvie) :- mother(Sylvie, Bob), male(Bob)*.
signifie que Bob est un fils de Sylvie si Sylvie est la mère de Bob et Bob est masculin

La signification intuitive d'une règle est que le prédicat exprimé dans la tête de la règle est vrai, si nous pouvons montrer que toutes les expressions dans le corps de la règle sont vraies.

Une **requête** est une demande de preuves ou de récupération d'informations depuis la base de données. La requête a la même structure que le corps d'une règle, c'est à dire, qu'elle est une séquence de prédicats séparée par des virgules et terminée par un point final. Pour présenter des requêtes nous écrivons souvent « ?- » . Par exemple :

— *?- male(Bob).*
Cette requête correspond à la question « Est-ce que Bob est masculin ? » Datalog répond « oui » si il peut le prouver (si le fait est dans la base de données), ou « non » si il ne peut pas (si le fait en est absent).

2.1.2 Datalog et les politiques de sécurité

Un langage de programmation logique (*i.e.* Prolog ou Datalog) est bien adapté et très utilisé [Lin, 1999; Wahsheh and Alves-Foss, 2008] pour exprimer et vérifier une politique de sécurité, grâce à sa nature déclarative basée sur la logique du premier ordre qui peut garantir des propriétés de politiques de sécurité telles que la sûreté et la complétude (*i.e.* la demande d'accès est accordée uniquement aux personnes qui en ont le droit). En outre, l'évaluation en Prolog et Datalog utilise le *backtracking*, l'algorithme de calcul qui peut déterminer tous les faits ou règles qui satisfont l'unification (trouver toutes les solutions d'un problème).

Quant à Datalog, sous-ensemble de Prolog, il est initialement utilisé pour effectuer des requêtes sur les bases de données relationnelles déductives. Cependant, il a aussi été beaucoup utilisé dans la sécurité d'information pour spécifier le contrôle d'accès. Par exemple, [Dougherty et al., 2006] ont exprimé une politique de contrôle d'accès dans un environnement dynamique en Datalog, ils ont également effectué la vérification de l'atteignabilité au but (goal), et la vérification de la permissivité d'une politique par rapport à une autre dans certains contextes.
[Fournet et al., 2007] ont utilisé Datalog afin de spécifier des politiques d'autorisation dans un environnement distribué. Ils ont intégré des prédicats logiques dans le calcul des processus (pi-calcul) pour vérifier si un système particulier implémente correctement la politique.

Grâce à Datalog, qui est un langage de programmation déclarative, nous n'avons pas besoin de spécifier comment atteindre un certain objectif dans une certaine situation, il suffit de définir des faits et des règles pour exprimer la situation et laisser un moteur de Datalog dériver le résultat pour notre requête (objectif). Un autre avantage à utiliser Datalog comme langage de spécification des politiques de sécurité est qu'il permet d'analyser automatiquement des exécutions de scénarios de notre système via l'invocation du moteur de

Datalog disponible.

2.2 Analyse des scénarios d'exécution au travers du centre médical

Jusqu'ici, nous avons présenté le modèle de contrôle d'accès D-OrBAC qui est une extension du modèle Or-BAC, pour résoudre le problème d'accès dans le cas d'un appel transitif, en particulier dans l'environnement des services web. Nous avons spécifié notre politique de contrôle d'accès en utilisant un fragment de la logique du premier ordre que nous pouvons traduire facilement en Datalog. Une entité peut être vue comme un terme, une relation comme un fait et un prédicat comme une règle. Seules les constantes, les variables et les prédicats sont suffisants pour spécifier notre politique ; nous ne considérons pas les symboles de fonctions. L'utilisation de Datalog nous assure la réponse d'une demande d'autorisation de notre politique.

Nous utilisons l'exemple du centre médical afin d'expliquer comment nous avons spécifié notre modèle en Datalog pour simuler le système.

Considérons un centre médical composé de 4 organisations : *Web portal(wp), Clinical Management(cm), Laboratory(la), Patient Records(pr)*. Les utilisateurs de chaque organisation peuvent avoir différents attributs. Les attributs de l'utilisateur sont représentés par les faits suivants :

- $org(s, cm)$ un sujet s appartient à l'organisation $ClinicalManagement$
- $role(s, doctor)$ le rôle d'un sujet s est égal à $docteur$
- $diploma(s, medecine)$ un sujet s a fait $medecine$
- $speciality(s, dentist)$ la spécialité d'un sujet s est $dentiste$
- $experience(s, 5)$ l'expérience de travail d'un sujet s est égale à 5 ans

Puisque chaque organisation est libre d'affecter n'importe quels sujets à n'importe quelles catégories selon sa politique de contrôle d'accès, nous avons introduit la règle *cat(org, sujet, catégorie)* pour représenter une affection sujet-catégorie pour chaque organisation en interne afin que la relation *Empower* reste générique pour toutes les organisations. Chaque organisation est libre de personnaliser la règle *cat* pour ses propres besoins.
La relation *Empower* est représentée en Datalog par cette règle :

```
empower(Org,U,Category):- org(U,Org), cat(Org,U,Category).
```

Exemple 2.1 (L'affectation sujet-catégorie)**.** *Dans l'organisation $WebPortal$, l'affectation peut être faite simplement en considérant le rôle d'un sujet.*

```
cat(wp,U,wp_nurse) :- role(U, nurse).
cat(wp,U,wp_doctor) :- role(U, doctor).
```

Cela signifie que dans l'organisation $WebPortal$ un sujet U est affecté à la catégorie wp_nurse si son rôle est « infirmière ». Si son rôle est égal à « docteur », le sujet U sera associé à la catégorie wp_doctor.

De même dans l'organisation $ClinicalManagement$, l'affectation peut dépendre d'autres attributs tels que le diplôme, l'expérience de travail ou la spécialité.

```
cat(cm,U,cm_senior_doctor) :- cat(cm,U,cm_doctor),
experience(U,Exp), Exp >= 5.
cat(cm,U,cm_doctor) :- diploma(U, medicine).
cat(cm,U,cm_doctor) :- speciality(U, physician).
cat(cm,U,cm_doctor) :- speciality(U, dentist).
```

Cela signifie que pour être assigné à la catégorie cm_senior_doctor dans l'organisation $ClinicalManagement$, le sujet U doit appartenir à la catégorie cm_doctor et son expérience de travail doit être de plus de 5 ans. En outre pour être associé à la catégorie cm_doctor, le sujet U doit faire des études en médecine, ou bien sa spécialité doit être « médecin » ou « dentiste ».

Dans le cas d'un environnement distribué comme le service web, nous devons déterminer si, afin d'accomplir une tâche, un service aura besoin ou non d'invoquer un autre service. Si cela est requis, on dit que ce service est dépendant d'un autre.

Définition 2.1: Dépendances

Un service est **dépendant**, s'il a besoin d'invoquer un autre service pour compléter le calcul, sinon il est **indépendant**. Afin de représenter cette relation, nous introduisons le fait *depends_on(service1, service2)*. Si $service1 \neq service2$, cela signifie que le $service1$ a besoin du $service2$ pour satisfaire la requête initiale, sinon le $service1$ est indépendant.

Exemple 2.2. *Dans l'organisation Clinical Management, nous pouvons définir des dépendances du service en Datalog comme ci-dessous :*

```
depends_on(careOrders_service, testOrders_service).
depends_on(vitals_services, vitals_service).
```

Cela représente une dépendance du services $careOrders\_service$ qui dépend du service $testOrders\_service$, alors que le service vitals dépend de lui-même, on dit qu'il est indépendant.

Après avoir définit les dépendances des services, nous devons déterminer à quelle organisation le service appartient. En comparant l'organisation du service avec celle de la ressource, cela nous aide à identifier si ce service a fait un appel au travers d'organisations ou non. Il nous permet de savoir si on a besoin d'une délégation de catégories pour évaluer cette requête. Nous pouvons identifier à quelle organisation un service appartient à l'aide du fait *belong_ to(service,organisation).*

Exemple 2.3. *Les appartenances des services sont spécifiées en Datalog comme ci-dessous :*

```
belong_to(vitals_service,cm).
belong_to(careOrders_service, cm).
belong_to(testOrders_service, la).
belong_to(patientHistory_service, pr).
```

Cela signifie que :

- *le service vitals et le service careOrders appartiennent à l'organisation $ClinicalManagement$,*
- *le service testOrders appartient à l'organisation Laboratory,*
- *et le service patientHistory appartient à l'organisation $PatientRecord$.*

Nous avons vu comment spécifier en Datalog les attributs, ainsi que l'association sujet-catégorie ou encore les dépendances et les appartenances des services. Nous allons maintenant montrer comment il est possible de spécifier la politique de sécurité. Rappelons que la relation $Permission$ est la relation abstraite entre une catégorie, une action et un objet. L'organisation dans laquelle une permission est valide est aussi indiquée dans cette relation.

$$Permission(Organisation, Categorie, Action, Objet) \tag{2.1}$$

Cette relation signifie que l'organisation donne la permission à une catégorie de réaliser une action sur un objet. Nous pouvons représenter cette relation en Datalog comme ci-dessous :

```
(1) permission(cm,cm_doctor,read, careOrders_service).
(2) permission(la,la_clinician,read,testOrders_service).
(3) permission(pr,pr_clinician,read, patientHistory_service).
```

— Le fait (1) signifie que l'organisation $ClinicalManagement$ donne la permission à la catégorie cm_doctor de consulter le service $careOrders\_service$.

— Le fait (2) signifie que l'organisation $Laboratory$ donne la permission à la catégorie $la\_clinician$ de consulter le service $testOrders\_service$.

— Le fait (3) signifie que l'organisation $PatientRecords$ donne la permission à la catégorie $pr\_clinician$ de consulter le service $patientHistory\_service$.

L'objectif de notre modèle est de rédiger la politique de sécurité à l'aide de permissions abstraites. Des permissions concrètes sont alors dérivées des permissions abstraites ; on peut définir la règle de dérivation (règle 1.2) déjà présentée dans la section 1.3.2 comme ci-dessous :

```
is_permitted(U, A, R):-
empower(Org, U, C),
permission(Org, C, A, R).
```

Cela signifie que si dans l'organisation Org le sujet U est associé à la catégorie C et l'organisation Org donne la permission à la catégorie C de réaliser l'action A sur l'objet R alors le sujet U a le droit de réaliser l'action A sur l'objet R.

Afin de spécifier la règle de dérivation des permissions abstraites dans l'environnement distribué (règle 1.3) où il y aura peut-être un appel transitif, nous avons besoin d'introduire une fonction auxiliaire qui parcourt une chaine d'invocations, et qui est donc récursive. Nous avons deux cas principaux à considérer :

1. Le $service1$ invoque le $service2$ qui est indépendant. Il nécessite une étape de délégations pour déterminer la requête. Par exemple, le service $PatientMedData$ de l'organisation $WebPortal$ appelle le service $Vital$ appartenant à l'organisation *Clinical Management*. Dans ce cas là, la délégation de catégorie entre *Web Portal* et *Clinical Management* doit être considérée.
2. La longueur d'une chaine d'invocations est supérieure à 2, nous avons besoin au moins de 2 étapes de délégations pour déterminer la requête. Prenons cette situation, le $service1$ invoque le $service2$ qui à son tour invoque le $service3$ qui est indépendant, les trois services sont de différentes organisations.

Nous devons examiner plus en détail le second cas parce que le $service2$ invoque le $service3$ en faveur du $service1$ et non au nom de lui-même, ce qui est différent du cas précédent.

Exemple 2.4. *Le service PatientMedData de l'organisation Web Portal invoque le service Care Order situé dans l'organisation Clinical Management. Afin de*

terminer son travail, Care Order a besoin d'appeler le service Test Order de l'organisation Laboratory au nom du service PatientMedData.

Par conséquent, pour satisfaire la requête initiale nous avons besoin de 2 étapes de délégations. La première délégation est entre le service Care Order et Clinical Management, la seconde est entre Clinical Management et Laboratory.

Nous pouvons spécifier la règle de dérivation des permissions abstraites impliquant plusieurs organisations à l'aide de la fonction auxiliaire *is_permitted_aux(Organisation, Category, Action, Resource1, Resource2)* comme illustrée dans la figure 2.1.

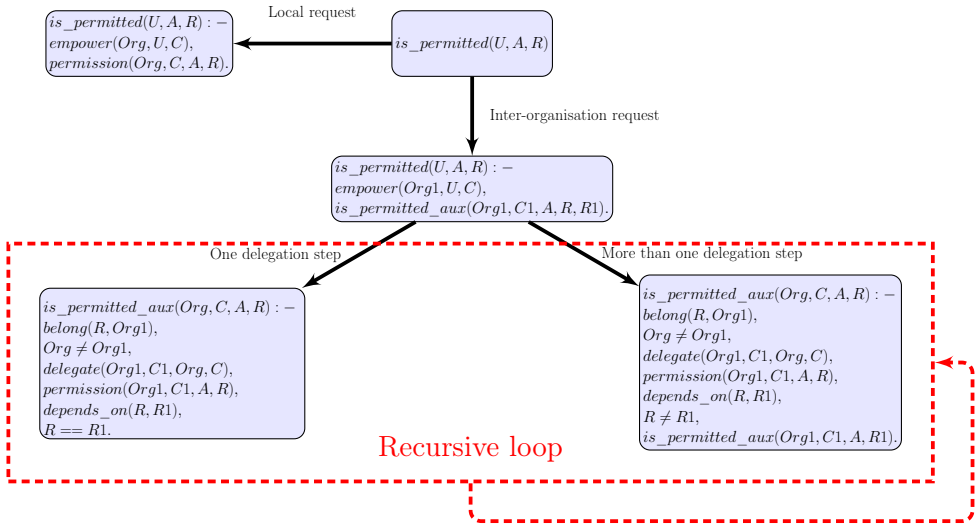


FIGURE 2.1 – Dérivation des permissions abstraites.

Nous montrons plus bas, un exemple d'une évaluation de requête d'accès lorsque Bob de l'organisation *Web Portal* essaie d'accéder au service *Care Order* appartenant à l'organisation *Clinical Management*. Premièrement, le système détermine l'organisation de Bob et récupère tous ses attributs afin de déterminer sa catégorie. Ensuite, il vérifie si l'organisation qui possède la ressource demandée est égale à l'organisation du sujet. Si l'organisation de Bob est différente de celle de la ressource, une délégation est alors nécessaire pour déterminer sa permission. Si des dépendances transitives sont présentes, comme dans cet exemple où Care Orders service dépend de Test Orders Service, alors, la chaîne de délégation est aussi calculée. La requête renvoie *True*, si les droits d'accès sont correctement propagés à travers la chaîne. Elle retourne *False* si un refus d'accès se trouve quelque part dans le chemin de services concernés. L'évaluation de requête en Datalog est comme ci-dessous :

```
>>> request = is_permitted('bob','read','careOrders_service')
New fact : org('bob','wp')
New fact : role('bob','doctor')
New fact : cat('wp','bob','wp_doctor')
New fact : empower('wp','bob','wp_doctor')
New fact : depends_on('careOrders_service','testOrders_service')
New fact : belong('careOrders_service','cm')
New fact : !=('wp','cm') is True
New fact : delegate('cm','cm_doctor','wp','wp_doctor')
New fact : permission('cm','cm_doctor','read','careOrders_service')
New fact : depends_on('testOrders_service','testOrders_service')
New fact : ==('testOrders_service','testOrders_service')
New fact : belong('testOrders_service','la')
New fact : delegate('la','la_clinician','cm','cm_doctor')
New fact : permission('la','la_clinician','read','testOrders_service')
New fact : is_permitted_aux('wp','wp_doctor','read',
'careOrders_service','testOrders_service')
New fact : is_permitted('bob','read','careOrders_service')
New fact : _pyD_query1()
True
```

Dans le cas où l'accès est accepté, le programme nous montre la trace de raisonnement de la requête en affichant les faits qui sont satisfaits jusqu'à ce que le programme se termine. Dans le cas où l'accès est refusé ou s'il existe une boucle infinie (nous détaillons plus tard la détection des boucles infinies), le programme nous montre jusqu'au dernier fait satisfaisant avant de retourner « False » à la demande d'accès.

Proposition 2.1

L'évaluation d'une requête d'accès dans notre programme Datalog pour le centre médical correspond à une dérivation de la relation 1.2 dans la définition 1.7 (resp. 1.3 dans la définition 1.8) dans le cas de la demande d'accès au sein de l'organisation (resp. inter-organisationnelle) définie dans le modèle D-OrBAC.
Datalog répond $True$ si et seulement si la demande d'accès est *accordée*.
Datalog répond $False$ si la demande d'accès est *refusée* ou si une récursion infinie est présente dans le programme.

Preuve. Par construction, puisque tous les prédicats du modèle D-OrBAC ont été spécifiés dans notre programme Datalog, celui-ci simule exactement le modèle de contrôle d'accès défini dans le chapitre 1.

Si l'évaluation de la requête $is\_permitted(U, A, R)$ est égale à $True$, alors la dérivation de la permission dans le modèle OrBAC étendu renverra la réponse *autorisé* ou *vice-versa*.
Si l'évaluation de la requête $is\_permitted(U, A, R)$ est égal à $False$, alors soit la dérivation de la permission dans le modèle D-OrBAC renvoie *refusé*, soit une récursion infinie est présente dans la liste de dépendances des services. Par conséquent, l'analyse des traces d'exécutions du programme Datalog est nécessaire afin de déterminer la signification de la réponse $False$ obtenue par l'évaluation de requête.

Si une récursion infinie est présente, elle doit être forcement celle engendrée par la règle :

```
is_permitted_aux( Org, C, A, R):- belong(R, Org1), Org \== Org1,
delegate(Org1, C1, Org, C),permission(Org1, C1, A, R),
depends_on(R,R1), R \== R1, is_permitted_aux(Org1, C1, A, R1).
```

qui parcourt la liste de dépendances des services. Avec l'aide de l'algorithme d'évaluation tabulaire [Chen et al., 1994; Chen and Warren, 1996] qui sauvegarde les calculs (et leurs résultats) de chaque nouvel appel dans un tableau, on peut éliminer le calcul redondant et produire une réponse pour une requête demandée. Grâce au moteur pyDatalog [Carbonnelle, 2014](qui utilise un algorithme d'évaluation tabulaire) que nous avons utilisé pour nos expérimentations, même si une récursion infinie est présente dans le programme Datalog, le moteur est capable de la détecter et de produire une réponse pour l'évaluation de la requête.

Les correspondances des relations du modèle D-OrBAC et des règles de Datalog sont récapitulées dans la table 2.1.

□

2.3 Analyse des scénarios d'exécution au travers du centre de recherche

Afin de démontrer que notre technique d'analyse et de validation peut s'appliquer dans la vie réelle et selon un scénario plus élaboré, reprenons le cas d'étude sur le centre de recherche que nous avons présenté dans la section 1.2.2.

Dans le centre de recherche, chaque département a assigné différemment ses utilisateurs à des catégories selon sa propre politique de contrôle d'accès. Par exemple, dans le cas du département *Secretary's Office*, les utilisateurs

Relation d'OrBAC étendu	**Ex. Datalog correspondant**
Empower(Org, Subject, Category)	empower(wp,bob, wp_doctor)
Delegate(Org1, Category1, Org2, Category2)	delegate(cm, cm_doctor, wp, wp_doctor).
Permission(Org, Category, Action, Object)	permission(cm,cm_doctor, read,careOrders_service).
Depend_on(Service1, Service2)	depends_on(careOrders_service, testOrders_service).
Belong(Service, Org)	belong(careOrders_service, cm).
Is_permitted(Subject, Action,Object)	is_permitted(Bob, read, careOrders_service).

TABLE 2.1 – Correspondances entre relations du modèle OrBAC étendu et règles de Datalog

sont assignés aux catégories par rapport à leur nature de travail. Dans le département *Administrative*, les utilisateurs sont associés aux différentes catégories en fonction de leurs postes dans le département, contrairement aux départements *Accounting* et *Information Technology*, où les catégories sont assignées aux utilisateurs d'après leurs rôles. Nous pouvons représenter l'affectation sujet-catégorie en Datalog comme ci-dessous :

```
cat(sec,U,sec_administrativeSecretary) :- task(U, administrative)
cat(adm,U,adm_director) :- position(U, director)
cat(acc,U,acc_budgetManager):- role(U, budgetManager)
cat(itd,U,itd_systemAdmin):- role(U,systemAdmin)
```

Afin de permettre aux départements dans ce centre de recherche de coopérer entre eux et de satisfaire la procédure d'autorisation des missions, ces départements doivent se mettre d'accord et déléguer les catégories nécessaires à leurs partenaires afin que ceux-ci puissent récupérer des données sensibles leur appartenant, tout en limitant les accès aux autres informations. Les délégations sont représentées par le graphe de délégation dans la figure 2.2.

Concernant les dépendances de services, nous avons vu dans le cas de la clinique que chaque service ne dépend que d'un service à la fois, c'est-à-dire que le $service1$ n'est dépendant que du $service2$ qui ne dépend que du $service3$, etc. pour réaliser la tâche initiale. Dans le second cas, un service peut dépendre de plusieurs autres services pour réaliser son travail et ces der-

FIGURE 2.2 – Graphe de délégation du centre de recherche.

niers pourront à leur tour dépendre de plusieurs services. Cela peut créer de nombreuses chaînes d'invocation transitive tant en largeur qu'en profondeur ; par exemple, quand le $service1$ dépend du $service2$ et du $service3$ pour finir la tâche et que le $service2$ et le $sevice3$ dépendent à leur tour d'autres services, et ainsi de suite. De plus, les actions que nous pouvons réaliser sur nos ressources ne sont plus toujours les mêmes pour toutes les organisations. C'est-à-dire que chaque organisation est libre de définir les actions sur ses propres ressources.

Nous avons donc modifié le fait $depends\_on(service1, service2)$ qui représente initialement quel service dépend de tel autre : nous souhaitons que le fait $depends\_on$ puisse représenter les dépendances multiples en indiquant que le service dépendant doit effectuer telles actions sur telles ressources. Le fait $depends\_on$ est maintenant défini comme suit :

```
depends_on(service,
action_1, service_1,
action_2, service_2,
...,
action_n, service_n).
```

où n est le nombre de dépendances maximal nécessaire. Si le service est indépendant, $action_1$ à $action_n$ seront égales à $noAction$ et $service_1$ à $service_n$ seront égaux à $noDepend$.

Au lieu de vérifier si le $service1$ est égal au $service2$ ou non, nous devons maintenant vérifier si le $service_i$ est égal à $noDepend$ ou non. Si les $service_1$ à

$service_n$ sont tous égaux à $noDepend$, alors le $service$ est indépendant ; dans le cas contraire, il est dépendant des $service_i$.

Une autre possibilité consiste à utiliser la liste en Prolog pour représenter les dépendances multiples. Afin de déterminer si le service donné est indépendant, nous vérifions simplement si la liste de dépendance est vide ou non. Puisque nous nous sommes intéressés à Datalog depuis le début, nous nous continuons à modéliser différemment les dépendances multiples.

Exemple 2.5 (Dépendances multiples)**.** *Dans le centre de recherche [Figure 1.2], le nombre de dépendances maximal nécessaire est égal à 2. Prenons le cas où le service SendRequest dépend du service ApproverRequest, et le service ApproveRequest dépend cette fois de deux services : GetBudget et GetMissionHistory qui sont indépendants. Nous pouvons modéliser ces dépendances multiples par les faits suivants :*

```
%% Centre de recherche %%
depends_on(SendRequest, consult, ApproveRequest, noAction, noDepend).
depends_on(approveRequest, read, getBudget, consult, getMissionHistory).
depends_on(getBudget, noAction, noDepend,  noAction, noDepend).
depends_on(getMissionHistory, noAction, noDepend,  noAction, noDepend).
```

Cela signifie que le service approuveRequest doit effectuer l'action read sur le service getBudget et l'action consult sur le service getMissionHistory.

Nous avons également amélioré l'algorithme de dérivation des permissions concrètes dans le cas d'un appel transitif avec les dépendances multiples des services. Nous avons définit des fonctions auxiliaires qui calculent récursivement des chaines d'invocation pour chaque dépendance de services. La dérivation est représentée dans la figure 2.3. Même si nous calculons différemment la délégation mais que le modèle de contrôle d'accès initial est inchangé, alors la correspondance entre programme Datalog et modèle est toujours garantie.

Proposition 2.2

La correspondance entre le programme Datalog spécifiant le centre de recherche et le modèle D-OrBAC est toujours garantie.

Preuve. Le mécanisme de l'évaluation d'une requête d'accès dans notre programme Datalog spécifiant le centre de recherche est identique à celui du centre médical (cf. proposition 2.1). La seule point qui est changé est le mécanisme de détermination des délégations de catégories qui ne modifie pas la sémantique du modèle de contrôle d'accès. Ainsi la correspondance entre notre programme Datalog et le modèle de contrôle d'accès est toujours garantie.

□

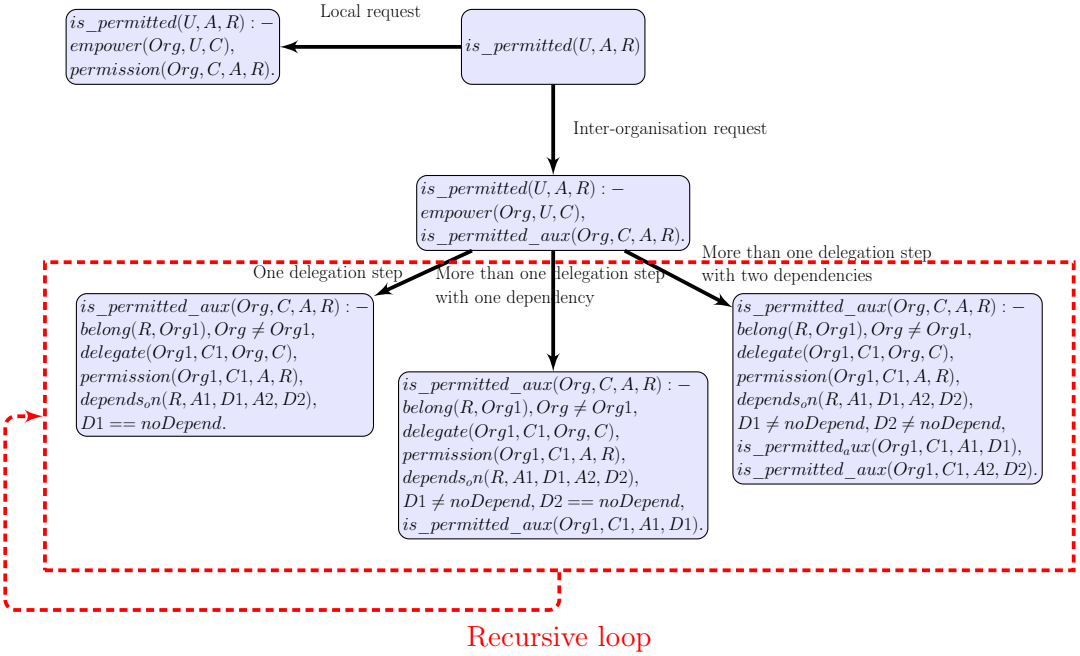


FIGURE 2.3 – Dérivation des permissions abstraites en présence de la délégation multiple.

Nous montrons plus bas, un exemple d'une évaluation de requête d'accès lorsque Alice de l'organisation *Secretary's Office* envoie la demande d'autorisation de partir en mission d'un chercheur au responsable concerné. Le service *SendRequest* tente d'accéder au service *ApproveRequest* appartenant à l'organisation *Administrative Department*. Le système détermine l'organisation de *Alice* et récupère tous ses attributs afin de déterminer sa catégorie. Ensuite, il vérifie si l'organisation qui possède la ressource demandée est égale à l'organisation du sujet. Si l'organisation d'Alice est différente de celle de la ressource, une délégation est alors nécessaire pour déterminer sa permission. Des dépendances multiples sont présentées comme expliqué dans l'exemple 2.5. Des chaines de délégations sont donc calculées. La requête renvoie *True*, si les droits d'accès sont correctement propagés à travers les chaines. Elle retourne *False* si un refus d'accès se trouve quelque part dans le chemin de services concernés. L'évaluation de requête en Datalog est comme ci-dessous :

```
New fact : org('alice','sec')
New fact : task('alice','administrative')
New fact : cat('sec','alice','sec_administrativeSecretary')
New fact : empower('sec','alice','sec_administrativeSecretary')
New fact : belong('approveRequest','adm')
New fact : !=('sec','adm') is True
```

```
New fact : delegate('adm','adm_director','sec',
'sec_administrativeSecretary')
New fact : permission('adm','adm_director','approve','approveRequest')
New fact : depends_on('approveRequest','read','getBudget','consult',
'getMissionHistory','noAction','noDepend','noAction','noDepend',
'noAction','noDepend')
New fact : !=('getBudget','noDepend') is True
New fact : !=('getMissionHistory','noDepend') is True
New fact : belong('getBudget','acc')
New fact : !=('adm','acc') is True
New fact : delegate('acc','acc_budgetManager','adm','adm_director')
New fact : permission('acc','acc_budgetManager','read','getBudget')
New fact : depends_on('getBudget','noAction','noDepend',
'noAction','noDepend',
'noAction','noDepend','noAction','noDepend','noAction','noDepend')
New fact : ==('noDepend','noDepend')
New fact : is_permitted_aux('adm','adm_director','read','getBudget')
New fact : belong('getMissionHistory','itd')
New fact : !=('adm','itd') is True
New fact : delegate('itd','itd_director','adm','adm_director')
New fact : permission('itd','itd_director','consult','getMissionHistory')
New fact : depends_on('getMissionHistory','noAction','noDepend',
'noAction','noDepend','noAction','noDepend','noAction','noDepend',
'noAction','noDepend')
New fact : is_permitted_aux('adm','adm_director','consult',
'getMissionHistory')
New fact : is_permitted_aux('sec','sec_administrativeSecretary',
'approve','approveRequest')
New fact : is_permitted('alice','approve','approveRequest')
New fact : _pyD_query1()
True
```

2.4 Terminaison et complexité

Remarquons que dans le modèle D-OrBAC, les requêtes d'autorisation sont associées aux requêtes de Datalog. Par conséquent (Proposition 2.1), nous pouvons étudier les propriétés du modèle par l'analyse du comportement du programme lié à l'évaluation des requêtes d'autorisation, en utilisant les résultats bien connus sur Datalog présentés par [Ceri et al., 1989]. Une des propriétés importantes des requêtes d'autorisation est leur finitude, c'est-à-dire toute requête d'autorisation admet un nombre fini de réponses. Cela garantit la sureté du programme Datalog spécifiant notre modèle de contrôle d'accès.

Du point de vue du modèle, elle assure que celui-ci peut toujours trouver une réponse pour toutes les requêtes. Le résultat présenté dans [Ceri et al., 1989] montre que toutes les relations définies dans un programme Datalog sont finies si les conditions suivantes sont satisfaites :

— des clauses unitaires (*i.e.* des clauses composées d'un seul littéral) ne contiennent aucune variable.

— chaque variable qui apparait dans la tête d'une règle apparait également dans un littéral positif dans le corps de la règle.

— chaque variable apparaissant dans un littéral négatif dans le corps d'une règle apparaît aussi dans certains littéraux positifs dans le corps de la règle.

Proposition 2.3

L'évaluation des requêtes d'autorisation de notre politique de contrôle d'accès décrite via le modèle D-OrBAC admet un nombre fini de réponses.

Preuve. Dans notre programme Datalog, les faits ne contiennent aucune variable. Il ne contient aucun littéral négatif ni symbole de fonction. Toutes les variables dans la tête de la règle apparaissent dans un littéral positif dans le corps de la règle. Par construction, le programme Datalog spécifiant notre modèle satisfait toutes les conditions présentées précédemment. Comme nous avons une équivalence entre le programme Datalog et notre modèle, nous pouvons en déduire que l'évaluation des requêtes d'autorisation de notre politique de contrôle d'accès décrite via le modèle OrBAC étendu admet un nombre fini de réponses. □

Le mécanisme d'évaluation tabulaire [Chen et al., 1994; Chen and Warren, 1996] est bien connu pour assurer la terminaison et la complétude de réponse à des requêtes Datalog, contrairement à la procédure de résolution SLD adoptée dans les moteurs de Prolog, qui ne garantit pas la terminaison. Dans le cas de l'évaluation tabulaire, celle-ci conserve les appels et leurs réponses associées dans un tableau. Elle utilise ces informations stockées au lieu de recalculer la réponse aux règles pour résoudre les appels répétés. Ainsi, les réponses redondantes d'un appel sont éliminées.

Proposition 2.4

L'évaluation des requêtes d'autorisation de notre politique de contrôle d'accès décrite via le modèle D-OrBAC se termine toujours.

Preuve. Nous avons utilisé le moteur pyDatalog [Carbonnelle, 2014], qui utilise un algorithme d'évaluation tabulaire, pour nos expérimentations. Ainsi, la terminaison de la procédure de réponse à des requêtes d'autorisation est garantie. Par cette observation, nous pouvons en déduire que l'évaluation des requêtes d'autorisation de notre programme Datalog se termine toujours. Pour cette raison, l'évaluation des requêtes d'autorisation de notre politique de contrôle d'accès décrite via le modèle OrBAC étendu se termine toujours. □

Concernant la complexité, [Vardi, 1982] a défini trois types de caractérisations de complexité pour répondre à des requêtes Datalog.

— la *complexité de données* est la complexité de l'évaluation lorsqu'un programme Datalog (des règles) sont fixes, alors que des bases de données d'entrées (des faits) et un atome clos (le but) sont une entrée.

— la *complexité d'expression* est la complexité de l'évaluation lorsque les faits sont fixes, alors que les règles et le but sont une entrée.

— la *complexité combinée* est la complexité de l'évaluation lorsque des règles, des faits et le but sont une entrée.

Regardons notre programme Datalog ; les règles sont toutes fixes car la topologie du système (y compris les délégations) est fixe. Il n'est pas possible de modifier les politiques d'autorisation au cours du fonctionnement du système, excepté les affectations sujets-catégories et sujets-attributs qui peuvent être changées selon le profil de l'utilisateur. Cela signifie que dans notre contexte, la caractérisation de la complexité à considérer est la complexité des données.

Théorème 2.1

L'évaluation des requêtes d'autorisation de notre politique de contrôle d'accès décrite via le modèle D-OrBAC est en temps polynomial.

Preuve. Puisque notre programme Datalog respecte la caractérisation de la complexité des données introduite par [Vardi, 1982], et puisque [Dantsin et al., 2001] ont montré que la complexité des données du programme Datalog est en P, nous pouvons en déduire que l'évaluation des requêtes est en temps polynomial. □

2.5 Discussion

Nous avons montré que Datalog est utilisable pour simuler nos cas d'étude afin de surveiller le fonctionnement du système et l'évaluation d'une requête d'autorisation sans avoir besoin d'implémenter le prototype. Cette simulation nous montre que notre système peut évaluer une requête d'accès en respectant la politique de sécurité dans l'environnement centralisé et distribué, et que le programme Datalog se termine toujours et en temps polynomial.

La terminaison du programme garantit que le système peut toujours nous donner une réponse pour une demande d'accès. Pourtant, elle ne peut pas garantir que la réponse donnée soit une bonne réponse sans vérifier les traces d'exécution du programme, puisqu'il se peut que le système ait répondu « non » à la requête d'accès dans le cas où la spécification de la politique serait mal définie et la récursion infinie présente dans le programme. Cependant, grâce à la programmation tabulaire implémentée dans Datalog, l'évaluation de la requête se termine et retourne « non satisfait » à la requête demandée. Même si les traces d'exécution du programme Datalog peuvent nous aider à identifier si la réponse donnée est correcte ou non, cela reste complexe à comprendre et à identifier l'endroit où des règles sont mal définies pour des administrateur de politique de contrôle d'accès qui ne sont pas familiers avec la programmation logique.

Nous avons besoin de prouver que notre modèle de contrôle d'accès basé sur l'organisation avec la délégation des catégories est bien défini et peut toujours donner une réponse à la demande d'accès, et cette réponse est unique. Ainsi, il ne peut pas répondre « autorisé » et « refusé » pour la même demande. Nous présentons l'utilisation du système de réécriture dans le prochain chapitre pour prouver ces propriétés ainsi que la terminaison de l'évaluation de requête comme nous l'avons fait avec Datalog sans perdre l'efficacité de calcul. L'autre raison pour laquelle nous choisissions le système de réécriture pour vérifier les propriétés de sécurité, est son expressivité au niveau de la représentation de structure de donnés tel que une liste ou un arbre, que nous pouvons utiliser pour représenter des dépendances multiples de service.

Chapitre 3

Définition et Vérification de la politique en utilisant la réécriture

Sommaire

La validation a pour but de vérifier que le système répondra aux besoins réels de l'utilisateur, tandis que la vérification vise à déterminer si le système est bien conçu et sans erreur. Nous avons déjà montré dans le chapitre précédent que le modèle D-OrBAC est valide en utilisant Datalog pour simuler des exécutions de scénarios possibles, et pour prédire les événements qui peuvent survenir et générer des problèmes d'autorisation. Dans le présent chapitre nous présentons la vérification des propriétés d'une politique de sécurité associée. Nous avons pour but d'assurer d'une part que la spécification est consistante et complète, et d'autre part que le système (modélisé auparavant par le modèle qui a été validé par notre technique formelle) répond aux propriétés de spécification.

Nous allons vérifier la qualité de la spécification de politique de contrôle d'accès, par exemple la consistance et la terminaison, en utilisant un système de réécriture, technique qui est déjà en partie étudiée dans mon travail de

Master [Uttha, 2012]. En effet, les langages basés sur la réécriture assurent une sémantique claire et sans ambiguïté. En outre, la nature déclarative de ce type de spécification de politique améliore la modularité, ce qui est un aspect crucial lorsque l'on considère des politiques de sécurité distribuées, développées indépendamment par différents départements, organisations ou institutions. En utilisant l'approche de réécriture, les politiques sont représentées comme des ensembles de règles de réécriture, dont l'évaluation produit des décisions d'autorisation, tandis que les demandes d'accès sont représentées comme des termes algébriques. Ainsi, l'évaluation de la demande est effectuée par la réduction des termes en une forme normale.

Dans ce chapitre, nous présentons en préambule les systèmes de réécriture. Par la suite, nous vérifions si chaque évaluation d'une requête est finie (terminaison) et peut renvoyer une décision (totalité). Puis, nous assurons que la réponse d'une requête d'autorisation est seule et unique (consistance). Enfin, nous présentons la vérification des politiques de sécurité en utilisant un outil CiME (une boîte à outil dédiée à la manipulation et l'analyse des programme de réécriture).

Définition 3.1: Propriétés de la politique

Afin qu'une politique de sécurité soit acceptable, il faut qu'elle satisfasse les propriétés suivantes :

Propriété 1 (Totalité) Toute demande d'accès d'un utilisateur u pour effectuer une action a sur une ressource r reçoit toujours une réponse, soit « autorisée » soit « refusée ». Cela peut assurer que pour toute requête d'accès, il existe toujours une décision d'accès correspondante.

Propriété 2 (Consistance) Pour tout utilisateur u et une requête req, il n'est pas possible de répondre « autorisé » et « refusé » pour une requête $access(u, req)$. La politique de sécurité est consistante si au plus un résultat est possible pour une demande d'autorisation, sinon on dit qu'il y a un conflit pour la spécification de la politique.

Propriété 3 (Terminaison) La politique de sécurité est terminante si pour toute requête req, toutes les dérivations de req sont finies. Cela assure que toute évaluation d'une requête est finie.

3.1 Préliminaires : Systèmes de réécriture de termes

3.1.1 Systèmes de réécriture de termes

La *réécriture* [Baader and Nipkow, 1998] est une méthode très puissante pour traiter des calculs avec des équations. Les équations en question, appelées *règles de réécriture*, sont utilisées pour remplacer des égaux par d'autres égaux, mais uniquement dans une seule direction. La théorie de la réécriture tourne autour de la notion de *forme normale* (une expression qui ne peut plus être réécrite). Le calcul consiste à réécrire en une forme normale ; quand la forme normale est unique, elle est prise comme la valeur de l'expression initiale. Si la réécriture mène toujours à la même forme normale, alors l'ensemble des règles est dit *convergent* et la réécriture peut être utilisée pour décider la validité des identités dans la théorie équationnelle. La réécriture a la puissance de calcul des machines de Turing.

Nous pouvons voir les systèmes de réécriture de termes comme un langage de programmation ou de spécification. Ils sont utilisés dans diverses applications telles que la démonstration automatique de théorème, l'optimisation du programme et récemment le sécurité informatique. Nous rappelons brièvement ici la définition des systèmes de réécriture de termes, nous renvoyons à [Baader and Nipkow, 1998] pour plus d'informations.

Définition 3.2: Signature

Une $signature$ $\mathcal{F}$ est un ensemble des symboles de fonction, où chaque $f \in \mathcal{F}$ est associé à un entier non négatif n, appelé l'$arité$ de f. Pour chaque $n \geq 0$, nous notons l'ensemble de tous les éléments n-aire de $\mathcal{F}$ par $\mathcal{F}^{(n)}$. Les éléments de $\mathcal{F}^{(0)}$ sont appelés des $symboles\ de\ constants$.

Définition 3.3: Terme

Étant donné la signature $\mathcal{F}$ et l'ensemble des $variables$ $\mathcal{X}$ (disjoint de $\mathcal{F}$). L'ensemble $T(\mathcal{F}, \mathcal{X})$ de tous les $\mathcal{F}$-termes sur $\mathcal{X}$ est défini comme ci-dessous :

— $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ (chaque variable est un terme)

— Pour tout $n \geq 0$, tout $f \in \mathcal{F}^{(n)}$ et tout $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$, nous

avons $f(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})$ (des symboles de fonctions appliqués à des termes sont aussi des termes)

L'ensemble des *termes* $T(\mathcal{F}, \mathcal{X})$ obtenus à partir de $\mathcal{F}$ et $\mathcal{X}$ peut être représenté par l'ensemble des arbres finis, où des nœuds sont des symboles de fonction f et leurs fils sont des arguments de la fonction. Nous obtenons ainsi que le nombre de sous-arbres est égal à l'arité de f.

Les *Positions* $\mathcal{P}os(t)$ sont des chaînes de nombres entiers positifs représentant un chemin menant de la racine à un nœud dans l'arbre. Le *sous-terme* de t à la position p noté par $t|_p$ et le résultat du remplacement $t|_p$ par u à la position p dans t est noté par $t[u]_p$. Cette notation est aussi utilisée pour indiquer que u est un sous-terme de t.

$\mathcal{V}(t)$ est noté pour l'ensemble des variables apparaissant dans t. Un terme est *linéaire* si les variables de $\mathcal{V}(t)$ apparaissent au plus une seule fois dans t. Un terme est *clos* si $\mathcal{V}(t) = \emptyset$.

Les substitutions sont écrites comme $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ où t_i est supposé être différent de la variable x_i. Nous utilisons les lettres grecques pour les substitutions et la notation post-fixe pour leur application.

Définition 3.4: Paire critique

Étant données :

- $l_1 \to r_1$ et $l_2 \to r_2$ sont deux règles de réécriture qui ne partagent pas des variables $\mathcal{V}(l_1) \cap \mathcal{V}(l_2) = \emptyset$,
- $p \in \mathcal{P}os(l_1)$ est un position tel que $l_1|_p$ n'est pas une variable, et
- σ est un unificateur le plus général (MGU : Most General Unifier) de $l_1|_p$ et l_2

Donc

$$\sigma(l_1) \to \sigma(r_1) \text{ et } \sigma(l_1) \to \sigma(l_1)[\sigma(r_2)]_p$$

La paire $\langle \sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p \rangle$ est appelée une **paire critique**

Définition 3.5: Système de réécriture des termes

Étant donnée une signature $\mathcal{F}$, un **système de réécriture des termes** sur $\mathcal{F}$ est l'ensemble des règles de réécriture $R = \{l_i \to r_i\}_{i \in I}$, où $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$, et $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$ et l_i est un ensemble fini. Considérons un terme t **réécrit** en terme u à la position p par la règle $l \to r$ et la

substitution σ, noté $t \to_p^{l\to r} u$, ou simplement $t \to_R u$, si $t|_p = \sigma(l)$ et $u = t[\sigma(r)]_p$; un tel terme t est appelé **réductible**. Les termes irréductibles sont dits en **forme normale**.

Nous notons $\to_R^+$ (resp. $\to_R^*$) pour la transitivité (resp. transitivité et réflexivité) la fermeture de la relation de réécriture $\to_R$. Le sous-indice R sera omis lorsqu'il est évident d'après le contexte.

Exemple 3.1 (Listes des nombres naturels [Barker and Fernández, 2006])**.** *Considérons la signature pour les listes des nombres naturels avec des symboles de fonction* z *(avec arité* 0*) et* s *(avec arité* 1*) pour construire les nombres,* nil *(avec arité* 0*) noté pour la liste vide,* cons *(avec arité* 2*) et* append*(avec arité* 2*) pour construire et concaténer les listes, respectivement, et enfin* $\in$ *(avec arité* 2*) pour tester l'appartenance d'un nombre entier naturel dans une liste. La liste contenant les numéros* 0 *et* 1 *est écrite comme* $\mathsf{cons}(\mathsf{z}, \mathsf{cons}(\mathsf{s}(\mathsf{z}), \mathsf{nil}))$*, ou simplement* $[\mathsf{z}, \mathsf{s}(\mathsf{z})]$*.*

Nous pouvons spécifier la concaténation des listes avec les règles de réécriture suivantes :

— $append(nil, x) \to x$ *et*

— $append(cons(y, x), z) \to cons(y, append(x, z))$*.*

Alors nous avons une séquence de réduction :

— $append(cons(z, nil), cons(s(z), nil)) \to^* cons(z, cons(s(z), nil))$*.*

Nous pouvons accéder au premier élément de la liste ou au reste de la liste après le premier élément avec l'aide des règles de réécriture suivantes :

— $head(cons(x, l)) \to x$

— $head(nil) \to nil$

— $tail(cons(x, l)) \to l$

— $tail(nil) \to nil$

Par exemple, nous pouvons avoir une séquence de réduction :

— $head(cons(z, cons(s(z), nil))) \to z$

— $tail(cons(z, cons(s(z), nil))) \to cons(s(z), nil)$

Les opérateurs booléens comme la disjonction, la conjonction, et la conditionnelle, peuvent être spécifiés en utilisant une signature qui inclut les constantes true and false. Par exemple :

— la conjonction est définie par les règles $\mathsf{and}(\mathsf{true}, x) \to x$, et $\mathsf{and}(\mathsf{false}, x) \to \mathsf{false}$

— la notation t_1 and $\ldots$ and t_n est une simplification pour le terme $\mathsf{and}(\ldots \mathsf{and}(\mathsf{and}(t_1, t_2), t_3) \ldots)$

— la conditionnelle *if* b *then* s *else* t est une simplification pour le terme $\mathsf{if\text{-}then\text{-}else}(b, s, t)$, avec des règles de réécriture : $\mathsf{if\text{-}then\text{-}else}(\mathsf{true}, x, y) \to x$ et $\mathsf{if\text{-}then\text{-}else}(\mathsf{false}, x, y) \to y$.

Nous pouvons définir l'opérateur d'appartenance "$\in$" comme suit : $\in (x, \mathsf{nil}) \to \mathsf{false}$, $\in (x, \mathsf{cons}(h, l)) \to$ *if* $x = h$ *then* true *else* $\in (x, l)$, où nous supposons que "=" est un test d'égalité syntaxique défini par les règles de réécriture standards. Nous écrivons souvent $\in$ comme un opérateur infixe.

Un système de réécriture R est :

— *confluent* si pour tous les termes t, u, v : $t \to^* u$ et $t \to^* v$ impliquent $u \to^* w$ et $v \to^* w$, pour un certain w ;

— *terminant* ou *fortement normalisant* si toutes les séquences de réduction sont finies ;

— *linéaire-gauche* si tous les côtés gauches des règles de R sont linéaires ;

— *non-chevauchant* s'il n'y a pas de paires critiques ;

— orthogonal s'il est linéaire-gauche et non-chevauchant ;

— *non-dupliquant* si pour tout $l \to r \in R$ et $x \in \mathcal{V}(t)$, le nombre d'occurrences de x dans r est inférieur ou égal au nombre d'occurrences de x dans l.

3.1.2 Propriétés des systèmes de réécriture

Terminaison.

Définition 3.6: Notion de terminaison

Soit R un système de réécriture

— L'élément s est *faiblement normalisant* dans R ssi s possède au moins une forme normale.

— L'élément s est *fortement normalisant* dans R ssi il n'existe aucune suite infinie $s = s_0 \to s_1 \to \ldots$ ssi chaque séquence de réductions à partir de s est finie.

— Le système R est *faiblement normalisant* ssi tout élément est faiblement normalisant dans R.

— Le système R *termine* ssi tout élément est fortement normalisant dans R.

Exemple 3.2.

— *le système formé de la règle unique* $x \to x + x$ *ne satisfait pas la propriété de terminaison, car il existe une chaine infinie* $x \to x + x \to x + x + x \to x + x + x + x \to \ldots$;

— *le système formé de l'unique règle* $x + y \to x$ *satisfait la propriété de terminaison, car la longueur de la règle est diminue à chaque étape de réécriture* $x + y + x + y + x + y \to x + x + y + x + y \to x + x + x + y \to \ldots$;

La preuve de la terminaison d'une relation de réécriture peut se faire de plusieurs manières. L'une des méthodes consiste à montrer que la relation de réécriture est incluse dans un ordre strict bien-fondé « $\succ$ » tel que $t \to t'$ implique $t \succ t'$, comme montré par Manna et Ness [Manna and Ness, 1970]. La terminaison d'un système de réécriture peut être également prouvée grâce à l'utilisation d'un ordre de simplification [Dershowitz, 1982] ; l'ordre dans lequel tout terme est syntaxiquement plus simple qu'un autre qui est plus petit que le suivant.

Théorème 3.1: Ordre de simplification [Dershowitz, 1982]

Étant donné une signature $\mathcal{F}$ avec un ensemble fini de symboles, un système de réécriture $\mathcal{R}$ sur $T(\mathcal{F}, \mathcal{X})$ termine s'il y a un ordre de simplification « $>$ » tel que $l > r$ pour toute règle $l \to r \in \mathcal{R}$

Confluence.

Définition 3.7: Notion de confluence

Soit R un système de réécriture.

— Deux termes s et t sont *joignables* ssi il existe v vérifiant $s \to^* v$ et $t \to^* v$.

— R est *Church-Rosser* [Voir le figure 3.1] ssi pour tout s, t tel que $s \longleftrightarrow^* t$, s et t sont joignables.

— R est *confluent* [Voir le figure 3.1] ssi pour tout s, t, u tel que $s \to^* t$ et $s \to^* u$, t et u sont joignables.

— R est *localement confluent* [Voir le figure 3.1] ssi pour tout s, t, u tel

que $s \to t$ et $s \to u$, t et u sont joignables.

— R est *fortement confluent* ssi pour tout s, t, u tel que $s \to t$ et $s \to u$, il existe v vérifiant $t \to^* v$ et $u \to^= v$.

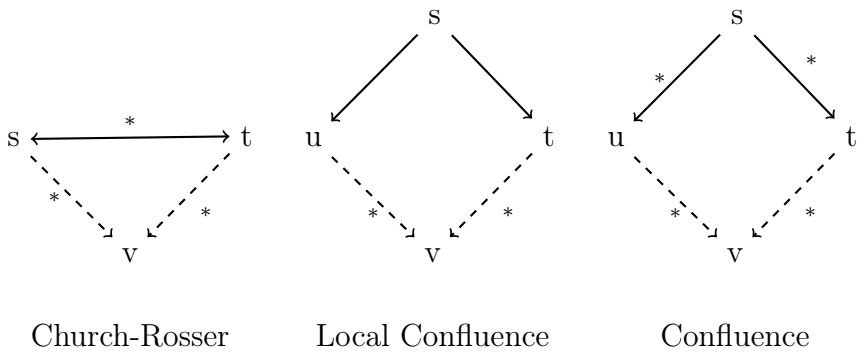


FIGURE 3.1 – Propriété Church-Rosser, Confluence locale et Confluence.

Théorème 3.2: Confluence [Newman, 1942]

Si le système R est *terminant* et *localement confluent* , alors il est *confluent.*

Si toutes les parties gauches des règles dans R sont linéaires et que les règles ne se chevauchent pas alors R est orthogonal. L'orthogonalité est une propriété de systèmes de réécriture décrit où les règles de réduction du système sont toutes linéaires, c'est-à-dire que chaque variable ne se produit qu'une seule fois sur le côté gauche de chaque règle de réduction, et il n'y a pas de chevauchement entre elles. L'orthogonalité est aussi une condition suffisante pour la confluence [Huet, 1980; Klop et al., 1993].

3.2 Définition d'une politique en réécriture et vérification de ses propriétés

Une politique de contrôle d'accès doit satisfaire certains critères pour être fiable. Par exemple, elle ne doit pas spécifier qu'un utilisateur est autorisé et rejeté pour le même privilège d'accès sur la même ressource (la politique doit être cohérente), ou encore la politique doit être capable de prendre une décision pour toutes les demandes d'accès (la totalité), etc.

La spécification des politiques de contrôle d'accès via des systèmes de réécriture de termes nous donne la possibilité de démontrer la satisfaction des propriétés importantes de la politique, comme la consistance et la totalité, en utilisant des propriétés du système de réécriture telles que la confluence et la terminaison.

Définition 3.8: Politique de sécurité

Une politique de sécurité $\mathcal{P}$ est un quadruplet $(\mathcal{F}, \mathcal{D}, \mathcal{R}, \mathcal{Q})$ où :

1. $\mathcal{F}$ est une signature
2. $\mathcal{D}$ est un ensemble non vide de termes clos appelés décisions : $\mathcal{D} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$
3. $\mathcal{R}$ est un ensemble fini de règles d'écriture sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$
4. $\mathcal{Q}$ est un ensemble de termes clos de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $\mathcal{Q} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ appelés requêtes d'accès

Nous explicitons ces éléments de la définition 3.8 ci-dessous :

— la spécification de la politiques et de son environnement est décrite en utilisant des termes construits à partir de la signature $\mathcal{F}$

— l'ensemble des décisions produites par la politique est noté $\mathcal{D}$. Dans notre cas, nous considérons comme des décisions possibles les constantes $grant$, $deny$

— le système de réécriture $\mathcal{R}$ contient les règles de réécriture qui définissent la politique de sécurité et son comportement

— Les requêtes d'accès $\mathcal{Q}$ sont des termes clos qui expriment des demandes d'accès de la part d'un utilisateur souhaitant effectuer une action sur une ressource

Exemple 3.3 (Politique de contrôle d'accès du centre médical)**.** *Nous pouvons modéliser une politique de contrôle d'accès pour un centre médical présenté dans la section 1.2.1 avec un système de réécriture $\mathcal{R}_{medical}$ comme ci-après :*

— *La signature $\mathcal{F}$ de la politique contient, entre autres, les symboles suivantes :*

depends_on	*: Service*	*→ Liste de services*
belong_to	*: Service*	*→ Organisation*
delegate	*: Organisation × Catégorie × Organisation*	*→ Catégorie*
organization	*: Sujet*	*→ Organisation*
empower	*: Organisation × Sujet*	*→ Catégorie*
permission	*: Organisation × Sujet × Action*	*→ Liste de services*
authorize	*: Organisation × Catégorie × Action × Ressource*	*→ Décision*
is_permitted	*: Sujet × Action × Ressource*	*→ Décision*

— *l'ensemble de décisions est* $\mathcal{D} = \{grant, deny\}$

— *l'ensemble de requêtes d'accès est constitué de termes clos de la forme* $is\_permitted(Sujet, Action, Ressource)$. *Par exemple,* $is\_permitted(bob, consult, careOrders\_service)$

— *le système de réécriture* $\mathcal{R}_{medical}$ *est l'ensemble des règles que nous pouvons regrouper selon leur nature et leur utilité (voir le système* $\mathcal{R}_{medical}$ *complet dans l'annexe A.1.2) comme ci-après :*

1. *des règles liées au modèle D-OrBAC, par exemple :*

```
delegate(cm,cm_doctor,la) -> la_clinician;
belong_to(careOrders_service) -> cm;
```

2. *des règles décrivant le comportement de la politique de sécurité, par exemple :*

```
permission(cm,cm_doctor,read)
    -> cons(vitals_service,cons(careOrders_service,nil));
```

3. *des règles représentant les attributs liés des utilisateurs, par exemple*

```
diploma(david, medicine) -> true
specialist(david, physician) -> true
experience(david) -> s(s(s(s(s(s(s(s(0)))))))))
```

4. *des règles utilitaires (manipuler des listes, faire des calculs sur des termes, etc.)*

```
ge(x,0) -> true
ge(0,s(y)) -> false
```

```
ge(s(x),s(y)) -> ge(x,y)

head(cons(x, l)) -> x
head(nil) -> nil
tail(cons(x, l)) -> l
tail(nil) -> nil
```

Théorème 3.3

Le système de réécriture $\mathcal{R}_{medical}$ spécifiant la politique de sécurité du centre médical selon le modèle D-OrBAC se *termine (est fortement normalisant).*

Preuve. Rappelons qu'un système de réécriture $\mathcal{R}$ est terminant si et seulement si tout élément dans $\mathcal{R}$ est fortement normalisant, c'est-à-dire qu'il n'existe aucune suite de réduction infinie en partant de n'importe quel élément.

Par inspection des règles de $\mathcal{R}_{medical}$ (voir annexe A.1.2), nous pouvons constater que :

— soit ce sont des règles qui ne posent pas de problème parce qu'elles n'engendrent pas la récursion

— soit ce sont des règles qui contiennent une récursion dans la partie droite de la règle telles que

```
eq(s(x),s(y)) -> eq(x,y)
le(s(x),s(y)) -> le(x,y)
ge(s(x),s(y)) -> ge(x,y)
is_member(x, cons(y,l)) -> ifeq(equal(x,y),x,l)
ifeq(true,x,l) -> true
ifeq(false, x, l) -> is_member(x,l)
append(l1,l2) -> ifappend(l1,l2,l1)
ifappend(l1,l2,nil) -> l2
ifappend(l1,l2,cons(x,l)) -> cons(x,append(l,l2))
```

Cependant, nous pouvons remarquer que la récursion dans la partie droite de la règle est appelée avec des arguments plus petits (en respectant l'ordre bien fondé [théorème 3.1]) que la partie gauche de la règle. Cela assure la terminaison de ces récursions.

— soit ce sont des règles liées à une évaluation d'une requête d'accès qui sont plus complexes et dont nous allons détailler leurs terminaisons par la suite.

La dérivation des termes liés à l'évaluation d'une requête d'accès (en particulier, une requête d'accès transitif) peut générer des récursions parce que nous sommes obligés de parcourir la liste de dépendances de services afin de vérifier si nous avons le droit d'accès à ces services ou non, et de vérifier si les services appartenant à cette liste dépendent d'autres services ou non, avant de prendre une décision pour cette demande d'accès. Nous pouvons obtenir une liste de dépendances de services par la réduction du terme $depends\_on(R)$ où R est un service initial auquel un utilisateur souhaite avoir accès. Cette liste contient des services que R doit invoquer afin de compléter son calcul. La forme normale du terme $depends\_on(R)$ est la suivante :

$$depends\_on(R) \rightarrow [R_1, R_2, R_3, \cdots, R_n]$$

où chacune de ces ressources à son tour peut avoir une liste de dépendances non vide, ou la liste vide si cette ressource est indépendante.

L'évaluation d'une requête d'accès transitif en système de réécriture commence par dérivation du terme $Is\_permitted\_global(ORG, C, A, R, L)$ où ORG est l'organisation d'un utilisateur U, C est la catégorie à laquelle U appartient, A est l'action que U souhaite effectuer sur la ressource R et L est la liste de décisions. Afin de montrer les différentes séquences de réduction possibles, nous les illustrons sous forme d'un arbre de réduction (voir figure 3.2) où $\rightarrow$ représente une réduction, $\rightarrow^*$ représente plusieurs étapes de réduction, les nœuds sont des termes dans $T(\mathcal{F}, \mathcal{X})$ et les fils de chaque nœud correspondent à des réductions du terme précédent.

Pour chaque itération, nous avons deux possibilités : soit le service R est indépendant, soit il dépend d'un autre service. Dans le premier cas, la réduction sera toujours terminée car les termes $authorize(\ldots)$, $belong\_to(\ldots)$, $append(\ldots)$ et $delegate(\ldots)$ ne présentent pas de récursions pour calculer leurs formes normales. Dans le second cas, la réduction va continuer de parcourir les chaines d'invocation transitive produites par chaque élément de la liste des dépendances jusqu'à ce qu'elle trouve un service indépendant. Par inspection de la topologie du centre médical (figure 1.1) et du système de réécriture $\mathcal{R}_{medical}$ dans l'annexe A.1.2, nous pouvons constater que les listes de dépendances de services sont finies et qu'elles ne forment pas de chemin d'exécution infini. C'est ainsi que la chaine d'invocation transitive dans $\mathcal{R}_{medical}$ est finie et ne contient pas de boucle. Par conséquent, cette récursion se termine toujours.

En outre, nous avons utilisé un prouveur automatique de la terminaison (AProVE) [Giesl et al., 2014] pour faire la preuve de la terminaison du système de réécriture $\mathcal{R}_{medical}$. AProVE nous a confirmé que $\mathcal{R}_{medical}$ est terminant. Par conséquent, nous pouvons conclure que le système de réécriture $\mathcal{R}_{medical}$ spécifiant la politique de sécurité du centre médical selon le modèle D-OrBAC se termine toujours. □

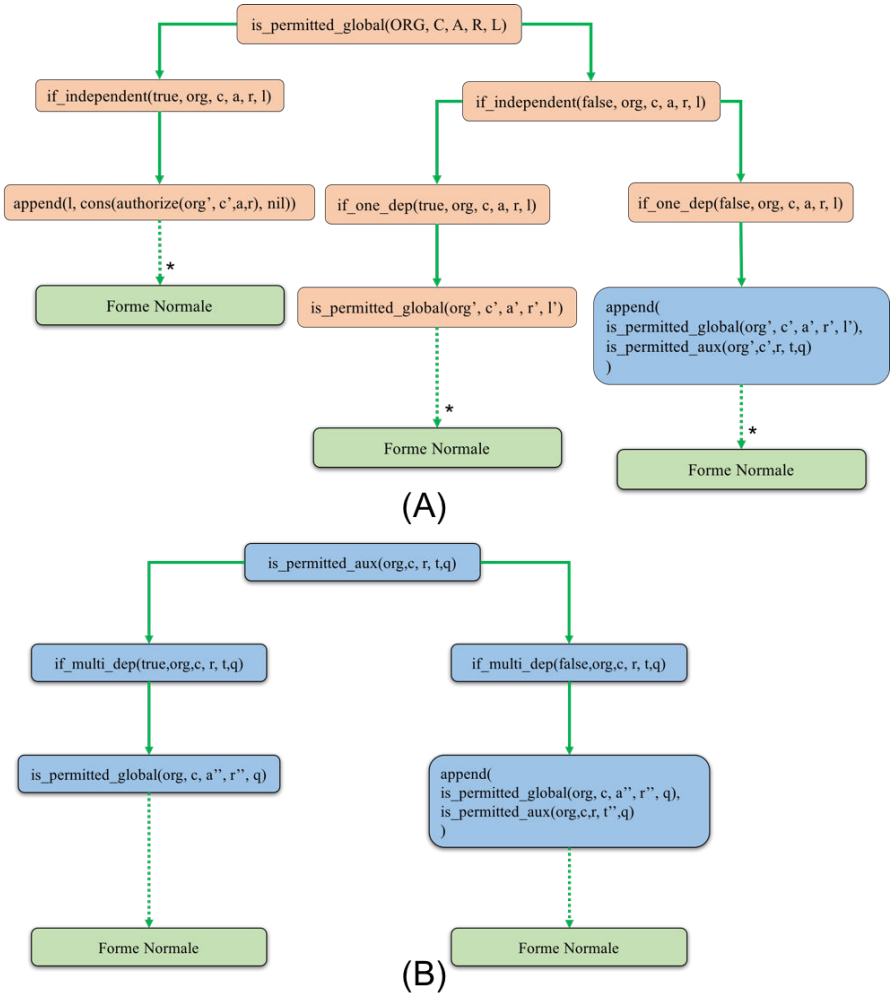


FIGURE 3.2 – (A) Arbre de réduction d'une requête d'accès transitif. (B) La suite de la réduction du terme $is\_permitted\_aux(org, c, r, t, q)$ dans (A)

Théorème 3.4

Le système de réécriture $\mathcal{R}_{medical}$ spécifiant la politique de sécurité du centre médical selon le modèle D-OrBAC est *localement confluent*.

Preuve. En analysant le système de réécriture $\mathcal{R}_{medical}$ dans l'annexe A.1.2, nous pouvons constater que notre système de réécriture $\mathcal{R}_{medical}$ ne contient aucune paire critique. Par conséquent, le système $\mathcal{R}_{medical}$ est localement confluent □

Proposition 3.1

Le système de réécriture $\mathcal{R}_{medical}$ spécifiant la politique de sécurité du centre médical selon le modèle D-OrBAC est *confluent*

Preuve. Nous avons montré que notre système de réécriture $\mathcal{R}_{medical}$ est terminant (résultat du théorème 3.3) et localement confluent (résultat du théorème 3.4). Par théorème 3.2, nous pouvons conclure que notre système de réécriture $\mathcal{R}_{medical}$ spécifiant notre politique de sécurité selon le modèle D-OrBAC est confluent. □

Proposition 3.2

Si le système de réécriture $\mathcal{R}_{medical}$ spécifiant la politique de sécurité du centre médical selon le modèle D-OrBAC est terminant et confluent, alors notre politique est totale et consistante.

Preuve. Une observation importante sur les propriétés de la réécriture présentées précédemment est que, si le système de réécriture définit d'une politique que se termine et est confluente, nous pouvons déduire que chaque demande d'accès a un résultat unique. Ceci est une conséquence du fait que, dans un système qui se termine, chaque terme a une forme normale et, dans un système qui est confluent, cette forme normale est unique. Ainsi, la totalité et la consistance peuvent être prouvées pour des politiques de contrôle d'accès définies dans le système de réécriture des termes, en vérifiant que la relation de réécriture générée est confluente et terminante. □

Nous pouvons généraliser ces résultats à d'autres systèmes de réécriture $\mathcal{R}$ qui spécifient une politique suivant le modèle D-OrBAC. Pour cela, il doivent respecter certaines conditions telles que

— $\mathcal{R}$ doit être non-chevauchant.

— si une règle de réécriture dans $\mathcal{R}$ définit une fonction récursive f, la partie droite de la règle ne doit contenir que des variables, des fonctions déjà définies précédemment ou bien un appel récursif de f avec des arguments plus petit que ceux de la partie gauche de la règle.

— les dépendances transitives de la ressource R sont une chaine $[R_1, \ldots, R_n]$ et la concaténation des dépendances transitives de tous les R_i ne forme pas de boucle infinie.

Avec ces critères ci-dessus, nous pouvons montrer que le système $\mathcal{R}$ est localement confluent et terminant.

Parfois le système de réécriture spécifiant une politique de sécurité contient un grand ensemble de règles et les règles peuvent être complexes. Il est donc intéressant de vérifier de façon automatique les propriétés du système de réécriture en utilisant des outils de réécriture tels que ceux présentés dans la sections suivante.

3.3 CiME et Vérification automatique

Il est facile de vérifier manuellement les propriétés de la réécriture si le système contient juste quelques règles de réécriture, mais il est beaucoup plus difficile d'effectuer la vérification dans un système qui contient des centaines de règles comme le cas de la spécification de la politique d'une grande organisation. La spécification des politiques basée sur la réécriture nous permet d'utiliser des techniques d'analyse automatique afin de prouver facilement les propriétés essentielles des politiques.

Une autre importante raison de spécifier des politiques de contrôle d'accès en utilisant le framework basé sur la réécriture, est que des langages et des outils de la réécriture tels que MAUDE [Clavel et al., 2003], TOM [Balland et al., 2007], ou CiME [Contejean et al., 2010] peuvent être utilisés : pour tester, comparer et expérimenter avec des stratégies d'évaluation, pour automatiser le raisonnement équationnel, ou pour le prototypage rapide des politiques de contrôle d'accès.

La confluence et la terminaison du système de réécriture sont des propriétés indécidables en général, mais il existe plusieurs techniques disponibles qui assurent la satisfaction de ces propriétés. En exploitant ces techniques, CiME3 est capable de vérifier et de certifier la terminaison et la confluence locale sur un grand nombre de problèmes (voir les résultats obtenus à partir de la base de données de problèmes de terminaison sur le site CiME [1]).

3.3.1 Description de l'outil CiME

L'outil CiME utilisé (CiME3), est une boîte à outils dédiée à la manipulation et l'analyse des programmes de réécriture. Il permet de définir les termes algèbres, les systèmes de réécriture, etc, et à effectuer une série de traitements

1. CiME :`http://a3pat.ensiie.fr`

sur eux : le calcul, la normalisation, le matching et l'unification (modulo les théories équationnelles), la procédure de complétion de Knuth-Bendix et des épreuves de l'égalité (modulo à la fois théories équationnelles), etc. Une partie importante de CiME3 est dédiée aux preuves de terminaison, la confluence locale et la convergence. CiME3 bénéficie d'un mode de haut niveau pour le développement interactif des programmes de réécriture, il peut également être utilisé en *batch* (il prend un fichier d'entrée dans un format spécifié, et retourne un fichier de résultat dans un format de sortie spécifié.) avec une automatisation complète. CiME3 a été développé dans le projet A3PAT pour la certification des preuves automatisées et l'automatisation pour les assistants de preuve. Nous conseillons de consulter [Contejean et al., 2011] et [Contejean et al., 2010] si l'information supplémentaire est nécessaire.

CiME3 peut être utilisé en batch (une simple ligne de commande) qui accepte les fichiers de formats différents (TRS [2],CPF [3],XML [4]) comme les fichiers entrés et sorties du programme. La contrainte de ce mode d'utilisation est que nous sommes obligé d'utiliser d'autres outils (Coq, Coccinelle) pour analyser le certificat et les traces de preuves qui sont les fichiers sorties de CiME, afin de répondre si notre système de réécriture est terminant et confluence. Nous avons choisi donc d'utiliser CiME en mode interactive qui nous permet de lancer directement les commandes pour tester la terminaison et la confluence.

3.3.2 Utilisation de l'outil CiME

Afin de définir un système de réécriture des termes en CiME, il faut définir une signature de l'algèbre et les règles de réécriture de terme sur cette algèbre. Les preuves de terminaison pour le système de réécriture peuvent alors être obtenues en utilisant différents critères, tels que des paires de dépendance et des raffinements de graphes, ainsi que divers ordres : par exemple des interprétations polynomiales, ou encore l'ordre de chemin récursif. La confluence locale est alors prouvée en montrant en particulier que chaque paire critique est joignable : le moteur de réécriture tente de normaliser chaque membre de la paire et de montrer que les deux membres se réduisent en un même terme. La certification des preuves peut être obtenue par une application du lemme de Newman [Newman, 1942].

La méthode le plus simple pour définir la politique de sécurité en CiME est de passer par la création d'un fichier sous le format TRS [5] qui contient les règles de réécriture spécifiant la politique et la liste des variables dans

2. Term Rewrite System :`https://www.lri.fr/~marche/tpdb/format.pdf`
3. Certification Problem Format :`http://cl-informatik.uibk.ac.at/software/cpf`
4. XML :`http://a3pat.ensiie.fr/pub/a3pat.dtd`
5. Term Rewrite System : `https://www.lri.fr/~marche/tpdb/format.pdf`

les règles. Ensuite, utiliser la ligne de commande de CiME pour le convertir en format CiME qui est reconnaissable par l'outil CiME3. Finalement, lancer l'outil en mode interactif et inclure le fichier CiME dans le programme afin de faire les tests et les analyses.

La vérification de la terminaison peut être faite en exécutant le programme suivant [Voir le programme complet en annexe] :

```
let R_trs_0_variable =  variables "a,c,l1,t,u,y,org,l,l2,r,x" ;
let R_trs_0_signature =
  signature " delegate : 3; is_permitted : 3; belong_to : 1;
  grant : 0; empower : 2; role : 2; la : 0;  cm_doctor : 0;
medicine : 0; depends_on : 1; permission : 3;  ...." ;
let R_trs_0_algebra =  algebra R_trs_0_signature ;
let R_trs_0 =
trs R_trs_0_algebra "
depends_on(vitals_service) -> nil;
depends_on(careOrders_service)
     -> cons(testOrders_service,nil)
     ....
     .... Rule definition ....";

 termination R_trs;
```

R_trs contient la définition de l'ensemble des variable $\mathcal{X}$, la signature $\mathcal{F}$, le terme algèbre $\mathcal{T}(\mathcal{F},\mathcal{X})$ et les règles de réécriture définit la politique de contrôle d'accès. Cette spécification est directement générée par CiME à partir du fichier TRS. L'exécution donne le résultat suivant :

```
termination R_trs;
...
 -: bool=true
```

Cela nous informe que la preuve de terminaison est trouvée. Nous continuons de façon similaire pour la confluence :

```
local_confluence R_trs;
 ...
-: bool=true
```

Si les deux propriétés sont satisfaites comme dans notre cas d'étude, cela prouve que la politique de contrôle d'accès est terminante et consistante. Dans le cas contraire, la cause de l'inconsistance peut être détectée par CiME :

```
local_confluence R_trs;
...
The rule [39] permission(cm,cm_doctor,read) overlapps the rule [48]
permission(cm,cm_doctor,read)) at position (epsilon)
yielding the non-joinable critical pair P=...
-: bool=false
```

Dans cet exemple, nous avons vu que les deux différentes listes de permissions sont associées à la même catégorie cm_doctor dans le domaine *Clinical Management*, ce qui rend finalement l'autorisation d'accès inconsistante. L'administrateur de politiques d'accès peut utiliser cette information pour faire la modification nécessaire afin de rendre la spécification de la politique correcte.

L'évaluation de l'autorisation d'accès peut être réalisée par CiME en calculant la forme normale d'un terme initial qui représente une demande d'accès. Par exemple :

```
let t = term R_trs_algebra "is_permitted(bob, read, careOrders_service)";
normalize R_trs_ t;
...
... Term deductions ...
- : R_trs_signature term = grant
```

Ce terme représente une demande d'accès de Bob pour effectuer l'action $read$ sur le service *Care Orders*. Le résultat indique que cette requête est accordée suivant la spécification de la politique de contrôle d'accès du centre médical.

3.4 Travaux Liés

[Alpuente et al., 2010] ont montré que nous pouvons transformer le programme Datalog en système de réécriture en préservant la propriété de terminaison de la programmation logique. De plus, avec leur technique, nous pouvons évaluer une requête en utilisant le système de réécriture transformé contrairement à la plupart des transformations qui ne s'intéressent qu'à l'analyse de la terminaison du programme logique transformé.

Dans un article de [Bertolissi et al., 2007], les systèmes de réécriture sont utilisés pour décrire la politique de contrôle d'accès dans des systèmes distribués. Elles ont présenté un nouveau modèle qui généralise RBAC en considérant les événements comme la source des demandes. Pour traiter avec les systèmes distribués, elles ont introduit des annotations de fonction pour distinguer dans quel nœud d'un système distribué la fonction est définie.

[Oliveira, 2007, 2008] a introduit l'utilisation du formalisme de réécriture stratégique pour la spécification et l'analyse modulaires de politiques de sécurité flexibles basées sur des règles. Il a développé des méthodes basées sur la réécriture de termes pour vérifier des propriétés plus élaborées des politiques telles que la sûreté dans le contrôle d'accès et la détection des flux d'information dans des politiques obligatoires.

[Giesl et al., 2014] ont développé AProVE qui est un outil pour faire l'analyse automatique de la terminaison de programmes Java, C, Prolog et système de réécriture des termes (TRS). Afin d'analyser des programmes écrits avec le langage de haut niveau, AProVE convertit automatiquement ces programme en TRS. Enfin, différentes techniques sont utilisées pour prouver la terminaison de TRS.

Chapitre 4

Mise en œuvre de notre approche

Sommaire

L'architecture orientée service (SOA) nous fournit une possibilité importante qui est de fusionner un service avec d'autres services ou d'autres systèmes d'information (existants ou futurs) sans restriction d'accès ou de développement. Elle nous offre également la réutilisabilité et la modularité permettant de réutiliser ou remplacer facilement un service. Cependant, un système distribué basé sur SOA doit avoir des mécanismes de contrôle d'accès robustes, flexibles et évolutifs afin d'assurer l'efficacité de la sécurité des données.

À présent avec l'évolution de SOA et des services web, le problème de la gestion des droits d'accès devient plus complexe car non seulement les ressources sont partagées mais aussi les politiques de gestion de ces ressources

sont distribuées, dynamiques et gérées par les différentes autorités. Concernant les ressources partagées parmi différentes organisations, nous avons présenté dans le chapitre 1, l'extension du modèle de contrôle d'accès basé sur les organisations D-OrBAC permettant de protéger ces ressources, et de résoudre le problème de détermination des droits d'accès aux ressources appartenant aux différentes organisations dans le système distribué, et en particulier dans le cas d'un appel transitif. Afin de résoudre le problème de gestion des politiques de contrôle d'accès, nous proposons d'utiliser le XACML standard [Parducci, 2013]. De plus, nous incorporons le service de délégation dans son architecture standard afin de fournir les facilités nécessaires au service d'autorisation d'accès pour qu'il puisse prendre en compte la délégation dans la prise de décision. Ce travail a été présenté à la conférence internationale SECRYPT'15 [Uttha et al., 2015a] et pendant la session de démonstration du symposium international ESSoS'15 [Uttha et al., 2015b].

Dans ce chapitre, nous montrons que notre proposition peut être mise en œuvre en étendant les capacités et les aptitudes des outils standards existants sans avoir besoin d'en développer de nouveaux ; notre solution est en outre capable de fournir à la fois la fonctionnalité désirée et la sécurité nécessaire du système grâce à notre modèle de contrôle d'accès D-OrBAC et à la flexibilité de l'architecture XACML. Pour cela, nous présentons la correspondance entre le modèle formel D-OrBAC spécifié en Datalog et l'architecture XACML qui nous amène à l'implémentation en respectant la norme standard XACML. Ensuite, nous décrivons une extension de XACML permettant de déléguer les catégories des utilisateurs afin d'assurer le fonctionnement de l'autorisation des requêtes d'accès transitifs dans les architectures SOA. Finalement, nous montrons la mise en œuvre de la politique de contrôle d'accès basée sur l'organisation dans notre cas d'étude.

4.1 Préliminaires : Architecture XACML

4.1.1 Architecture XACML

eXtensible Access Control Markup Language (XACML) [Parducci, 2013] est un OASIS [1] standard qui définit une architecture et un langage pour le contrôle d'accès (autorisation). Le langage consiste en des requêtes, des réponses et des politiques. Il fournit un schéma XML standard pour définir des politiques et des règles de sécurité. Cela nous permet de décrire des exigences générales de contrôle d'accès. Étant donnée que XACML s'appuie sur le langage XML, ceci nous autorise à nous affranchir des problèmes de formatage

1. `https://www.oasis-open.org/`

de données locales lors de l'échange d'informations de contrôle d'accès entre deux systèmes. C'est donc un langage pour l'interopérabilité.

XACML a plusieurs avantages par rapport aux autres langages de spécification des politiques de contrôle d'accès. Par exemple :

— Il est considère comme **standard** car il est souvent déployé dans des différent systèmes, il sera donc plus facile d'interopérer avec d'autres applications utilisant le même langage standard.

— Il est **distribué** car des politiques de sécurité peuvent être écrites et gérées depuis différents endroits et par différentes personnes. XACML est capable de combiner correctement les résultats de ces différentes politiques dans une décision.

— Il est **générique**. Cela signifie que, plutôt que d'essayer de fournir un contrôle d'accès pour un environnement particulier ou un type spécifique de ressources, il peut être utilisé dans n'importe quel environnement. Une politique peut être écrite et pourra ensuite être utilisée par de nombreux types d'applications.

4.1.2 Spécification des politiques de sécurité en XACML

XACML fournit un langage très flexible pour exprimer les politiques de contrôle d'accès. L'élément racine des politiques XACML est soit « Policy » soit « PolicySet » . « **Policy** » représente une seule politique de contrôle d'accès, exprimée à travers un ensemble de règles. Alors que « **PolicySet** » est un conteneur qui peut contenir d'autres politiques ou ensemble des politiques.

En outre, XACML fournit également des extensions standard pour définir des nouvelles fonctions, des types de données et des combinaisons de logique etc. De plus, il spécifie un langage requête/réponse qui nous donne la possibilité d'interroger le système afin de savoir si l'action demandée devrait être autorisée ou non. Le serveur évalue la requête contre des politiques disponibles et renvoie un résultat. Le résultat inclut toujours un des quatre types de réponses :

— **Permit** : l'action demandée est autorisée

— **Deny** : l'action demandée est refusée

— **Indeterminate** : une erreur s'est produite ou une valeur demandée manquait, donc la décision ne peut pas être prise

— **Not Applicable** : la demande ne peut pas être résolue par ce service

4.1.3 Principaux composants de l'architecture XACML

XACML propose une architecture de référence avec des noms communément acceptés pour les différentes entités impliquées dans l'architecture. Le standard XACML décrit également le mécanisme d'évaluation des demandes d'accès selon des règles définies dans les politiques de contrôle d'accès. Toutefois, avant d'expliciter la correspondance entre le programme Datalog spécifiant notre politique de contrôle d'accès et l'architecture XACML, nous présentons dans cette section les composants principaux et leurs fonctionnements au sein de cette architecture.

L'infrastructure générique d'autorisation se compose de [Voir figure 4.1] :

— **Policy Information Point** (PIP) fournit les informations sur des sujets, ressources, environnements

— **Policy Administration Point** (PAP) gère les politiques de sécurité

— **Policy Decision Point** (PDP) évalue les demandes d'accès au regard des politiques d'autorisation avant d'émettre des décisions d'accès

— **Policy Enforcement Point** (PEP) intercepte la requête d'accès d'un utilisateur à une ressource, effectue une requête de décision au PDP pour obtenir la décision d'accès (l'accès à la ressource est approuvé ou rejeté), et agit sur la décision reçue

4.1.4 Mécanisme d'évaluation de la demande d'accès dans XACML standard

— Des politiques de contrôle d'accès sont définies dans le PAP et sont disponibles pour le PDP (Étape 1)

— Une requête d'accès d'un utilisateur est interceptée par le PEP (Étape 2)

— Le PEP transfère la requête d'accès au PDP en passant par le gestionnaire de contexte (Étapes 3-4)

— Le PDP demande plus d'informations sur l'utilisateur, la ressource et l'environnement au PIP (Étapes 5-6)

— Le PIP se connecte aux centres d'informations et récupère les informations demandées et les renvoie au PDP (Étapes 7-10)

— Le PDP sélectionne la politique de sécurité de l'organisation qui dispose de la ressource à laquelle l'utilisateur a demandé l'accès parmi des politiques pré-téléchargées depuis le PAP

— Le PDP prend la décision en fonction des informations qu'il possède et renvoie sa décision d'accès au PEP (Étapes 11 -12)

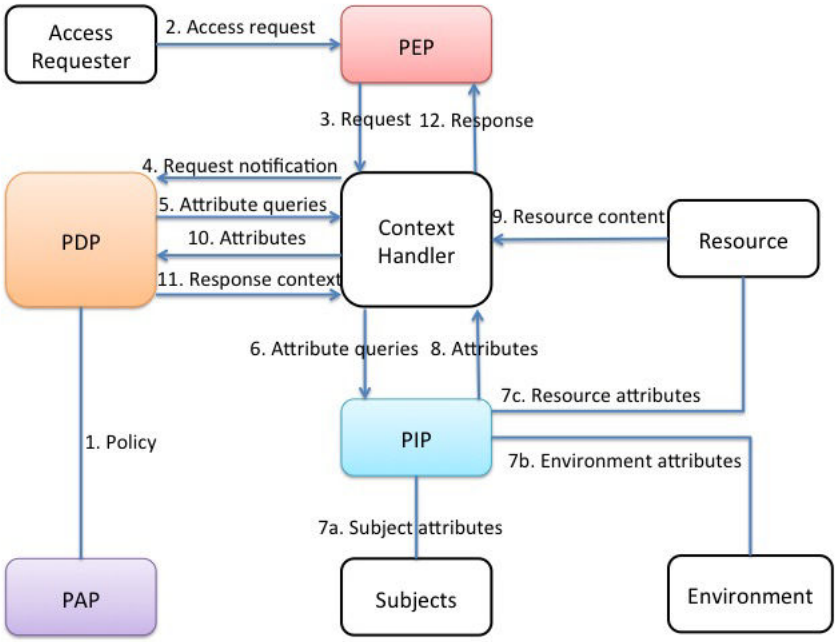


FIGURE 4.1 – Architecture d'XACML.

— Si la demande d'accès est acceptée, le PEP donne l'accès à la ressource demandée, sinon l'accès est rejeté

4.2 Du modèle D-OrBAC vers l'architecture XACML

Nous présentons la correspondance entre notre modèle formel (D-OrBAC) spécifié en Datalog et l'architecture XACML qui nous amène à l'implémentation en respectant le norme XACML standard. Pour cela, Reprenons le programme représentant le cas d'étude sur le centre médical présenté dans la section 2.2 [Voir programme Datalog complet dans la section A.1.1]. Nous pouvons classer les faits et les règles de Datalog en 5 groupes comme ci-dessous :

L'association utilisateur-catégorie : AUC. Faits ou règles qui caractérisent des utilisateurs de chaque organisation : ils représentent différents attributs d'utilisateurs tels que rôle, âge, organisation, diplômes, etc. L'association utilisateur-catégorie est aussi dans ce groupe. Par exemple dans l'organisation *Clinical Management* :

```
org(david, cm).
diploma(david, medecine).
speciality(david, physician).
experience(david,10).

empower(Org,U,Category):- org(U,Org), cat(Org,U,Category).
cat(cm,U,cm_doctor) :- diploma(U, medecine).
cat(cm,U,cm_doctor) :- speciality(U, physician).
cat(cm,U,cm_doctor) :- speciality(U, dentist).
```

Appartenance de services : APS. Faits ou règles qui caractérisent des ressources de chaque organisation : ils représentent l'appartenance d'une ressource. Par exemple :

```
belong(vitals_service, cm).
belong(careOrders_service, cm).
belong(testOrders_service, la).
belong(testResults_service, la).
belong(patientHistory_service, pr).
belong(showProfile_service, wp).
```

Permission : PER. Faits ou règles qui spécifient des permissions : ils représentent des politiques de sécurité. Par exemple :

```
permission(cm,cm_doctor,read, vitals_service).
permission(cm,cm_doctor,read, careOrders_service).
permission(cm,cm_senior_doctor,read, careOrders_service).
permission(cm,cm_senior_doctor,modify, careOrders_service).
permission(cm,cm_nurse,read,vitals_service).
```

Autorisation : AUT. Faits ou règles qui définissent la façon de calculer des permissions concrètes : ils permettent de dériver la permission concrète à partir des permissions abstraites. Par exemple :

```
is_permitted(U, A, R):-
empower(Org, U, C), permission(Org, C, A, R).

is_permitted(U, A, R):-
empower(Org, U, C),
is_permitted_aux( Org, C, A, R).
```

```
%%% 1 delegation %%%
is_permitted_aux( Org, C, A, R):-
belong(R, Org1),
Org \== Org1,
delegate(Org1, C1, Org, C),
permission(Org1, C1, A, R),
depends_on(R,R1),
    R == R1.

%%% >= 2 delegations %%%
is_permitted_aux( Org, C, A, R):-
belong(R, Org1),
Org \== Org1,
delegate(Org1, C1, Org, C),
permission(Org1, C1, A, R),
depends_on(R,R1),
R \== R1,
is_permitted_aux(Org1, C1, A, R1).
```

Délégation : DEL. Faits ou règles qui représentent les délégations entre différentes organisations, et les dépendances des services. Par exemple :

```
delegate(cm, cm_doctor, wp, wp_doctor).
delegate(la, la_clinician, cm, cm_doctor).
delegate(pr, pr_clinician, la, la_clinician).

depends_on(careOrders_service, testOrders_service).
depends_on(testOrders_service, testOrders_service).
```

En fonction de la nature de chaque groupe présenté précédemment, nous pouvons trouver une correspondance dans l'implémentation concrète de l'architecture XACML pour chaque composant comme ci-après :

— Tout ce qui concerne les informations sur des utilisateurs, des ressources ou des services partagés correspond à PIP, le composant qui fournit toute l'information sur des entités apparues dans la requête d'accès à PDP afin que celui-ci puisse prendre la décision d'accès. Par conséquent, les faits et règles appartenant aux groupes AUC et APS peuvent être représentés par le module PIP de l'architecture XACML.

— Les faits et règles appartenant au groupe PER correspondent au module PAP de l'architecture XACML puisqu'ils peuvent être considérés comme

un entrepôt de politique de contrôle d'accès qui est en contact direct avec le module PAP.

— À propos des faits et règles du groupe AUT, ils peuvent être simulés par le module PDP de l'architecture XACML qui rassemble toutes les informations sur les entités données par PIP et récupère des politiques définies dans PAP, afin de prendre la décision concernant la demande d'accès ; ceci est similaire aux règles $is\_permitted()$ du programme Datalog qui réduit tous les faits et règles apparues dans la partie gauche de la règle jusqu'à ce qu'il ne puisse plus effectuer d'autre réduction.

— Enfin les faits et règles appartenant au groupe DEL ne correspondent à aucun composant existant de l'architecture XACML, nous avons donc introduit une extension qui représente le fonctionnement de ces faits et règles et s'intègre dans l'architecture XACML standard. Nous la présentons plus en détail dans la section suivante.

Les correspondances entre les règles définissant la politique et différents modules de l'architecture XACML sont explicitées graphiquement dans la figure 4.2.

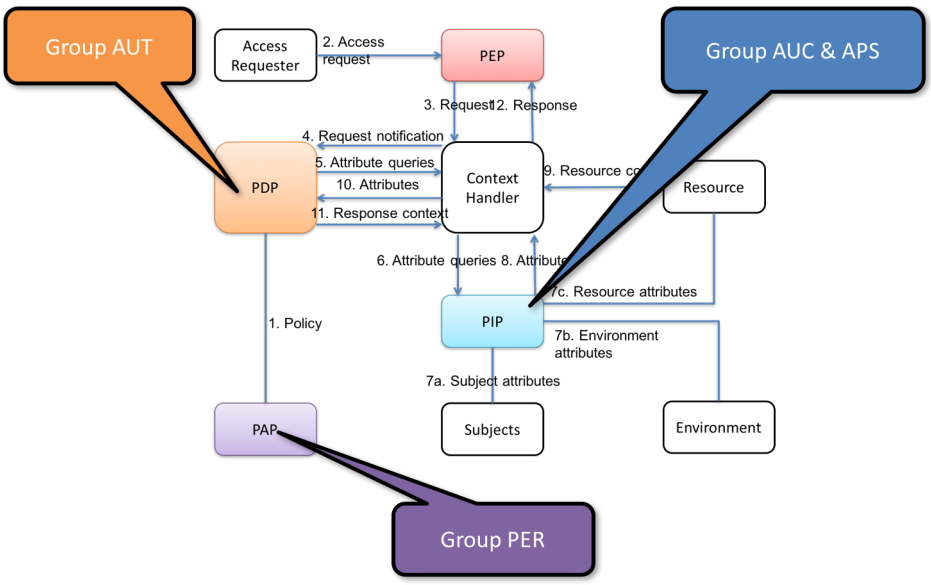


FIGURE 4.2 – Correspondances entre Datalog et XACML.

4.3 Extension de XACML avec un module de délégation

Dans la section 1.3, nous avons montré notre extension du modèle de contrôle d'accès basé sur des organisations (D-OrBAC), ainsi que la modélisation de la délégation des catégories en utilisant un graphe orienté acyclique, mais nous n'avons pas encore présenté le fonctionnement du module de délégation ni la mise en œuvre de ce module. Nous avons montré dans la section précédente les correspondances entre différentes parties de Datalog et les composants de l'architecture XACML. Nous remarquons que la délégation des catégories ne correspond à aucun module de XACML. C'est ainsi que dans cette section, nous présentons l'extension de XACML qui pourra supporter des délégations afin de résoudre le problème d'appel transitif dans un système distribué où plusieurs organisations sont impliquées, en particulier des services web.

4.3.1 Module de délégation

Afin que le PDP puisse prendre une décision concernant des demandes d'accès, il doit récupérer toutes les données nécessaires telles que les informations sur des sujets, des ressources et des environnements fournies par le PIP ou les politiques correspondantes fournies par le PAP. Avec l'architecture XACML standard, le PIP ne peut fournir que les informations sur le sujet $s1$ (sa catégorie et ses attributs) obtenues par l'organisation du sujet ($org1$), et les informations sur la ressource $r1$ (son organisation et ses attributs) obtenues par l'organisation de la ressource($org2$) mais il n'y a aucun lien entre ces deux organisations.
Ceci n'est pas suffisant pour le PDP lorsqu'il doit répondre à une demande d'accès transitif dans laquelle l'organisation du sujet et celle de la ressource sont différentes, car il n'existe aucune politique de sécurité qui spécifie que le sujet de l'organisation $org1$ puisse manipuler ou non la ressource de l'organisation $org2$. Puisque la politique qui protège la ressource $r1$ (basée sur la catégorie) ne peut pas déterminer la catégorie du sujet $s1$ dans son organisation, le PDP va répondre toujours que le sujet $s1$ ne peut pas accéder à la ressource $r1$.
C'est ainsi que nous voulons introduire un module intermédiaire permettant de fournir la délégation des catégories (laquelle représente le lien entre deux organisations) afin que les sujets puissent être reconnus dans ses organisations partenaires quand ils demandent l'accès aux ressources appartenant aux autres organisations. Nous appelons ce module intermédiaire "*le module de*

délégation".

Le fonctionnement du module de délégation est le suivant :

1. le module est initialisé avec des délégations de catégories qui sont déjà prédéfinies selon des accords entre différentes organisations
2. quand il y a une demande d'informations sur un sujet et une ressource, il vérifie si le sujet et la ressource appartiennent à des organisations différentes ou non :
   - si le sujet et la ressource appartiennent à la même organisation, la délégation des catégories n'est pas nécessaire
   - dans le cas contraire, la délégation des catégories est calculée en respectant le graphe de délégation ; l'algorithme de la récupération d'une délégation est présenté dans le paragraphe suivant
3. l'information sur le sujet (soit la catégorie de ce sujet au sein de son organisation, soit la catégorie déléguée obtenue par l'organisation de la ressource) est retournée au demandeur de l'information

Algorithme de la récupération d'une délégation.

Supposons que :

- chaque catégorie possède un préfixe sous forme "*abréviation du nom de l'organisation* :*nom de la catégorie*" par exemple *org1 :cat1, org2 :cat2*, etc.
- une catégorie ne peut correspondre qu'à une seule catégorie pour chaque organisation

Algorithme 4.1

Nom de algorithme : getDelegation1(cat, org1, org2)
Entrée : cat : *catégorie de sujet* , org 1 : *l'organisation du sujet*, org2 : *l'organisation de la ressource*
Sortie : newCat : *Catégorie déléguée*

Si $org1 == org2$ alors
$newCat = cat$
Sinon
vérifier s'il existe une délégation de catégorie pour la catégorie cat dans l'organisation $org2$:

- récupérer tous les arcs du graphe de délégation où leurs sommets sources sont égaux à cat
- supprimer tous les arcs où leurs sommets cibles ne commencent pas par $org2$
- il nous reste au plus un arc (cat, $org2 : cat2$)

$newCat = cat2$

Retourner *newCat*

Remarquons que dans le cas d'un appel transitif, l'organisation du sujet n'a peut-être pas de lien direct avec l'organisation de la ressource dont il souhaite avoir accès, mais il peut tout même avoir accès à cette ressource via les délégations transitives. Afin d'améliorer la performance de l'algorithme de la récupération d'une délégation et de prendre en compte une chaine d'invocations de services, nous introduisons une liste des catégories déléguées (pour éviter de recalculer la catégorie du sujet à chaque fois qu'il revient dans la même organisation) et une liste ordonnée des organisations que le sujet a déjà visité.

Algorithme 4.2

Nom de algorithme :
getDelegation2(cat, org1, org2, delegatedCat, invocationChain)
Entrée : cat : *catégorie de sujet*, org 1 : *l'organisation du sujet*,
org2 : *l'organisation de la ressource*,

delegatedCat : *la liste des catégories déléguées*,
invocationChain : *la liste ordonnée des organisations déjà visitées*
Sortie : newCat : *Catégorie déléguée*

Si $org1 == org2$ alors
    $newCat = cat$
Sinon
    Si $org2 \in invocationChain$ alors
        $newCat = delegatedCat.get(org2)$
    Sinon
        $org1 = invocationChain.last()$
        $cat = delegatedCat.get(org1)$
        $newCat = getDelegation1(cat, org1, org2)$
        $invocationChain.add(org2)$
        $delegatedCat.put(org2, newCat)$

Retourner *newCat*

Nous pouvons utiliser cet algorithme pour gagner du temps de calcul en gardant la liste de catégories déjà calculées dans les cas où les mises à jour et les changements de la politique ne sont pas très fréquents. Néanmoins, dans un contexte plus dynamique où nous avons besoin de prendre en compte des mises à jour fréquentes de paramètres tel que le changement des attributs de l'environnement (heure, localisation d'une demande, etc.), nous sommes obligés de réinitialiser les historiques de catégories calculées afin d'obtenir une réponse d'accès prenant en compte les nouveaux paramètres.

4.3.2 Emplacement du module de délégation

Notre principal objectif en ajoutant le module de délégation dans l'architecture XACML, est de ne pas changer l'implémentation des principaux composants de XACML ni de modifier des applications existantes pour pouvoir utiliser ce module. Le module de délégation est un composant supplémentaire qui peut être appelé soit par le PEP, soit par le gestionnaire de contexte (Context handler), ou bien incorporé dans le PIP. Chaque choix a des avantages et des inconvénients suivant :

- *Le module de délégation appelé par le PEP* : l'avantage est qu'il n'est pas nécessaire de changer l'implémentation de XACML, il suffit d'appeler le module de délégation afin de récupérer la délégation de catégories avant de passer ces données, avec une requête d'accès, au PDP. Puisque le PEP

est un composant dépendant d'une application, il est nécessaire de le modifier afin qu'il puisse utiliser ce module de délégation.

- *Le module de délégation appelé par le gestionnaire de contexte* (composant offrant une interface intermédiaire entre PEP, PIP et PDP) : l'avantage est que les applications existantes (*i.e.* PIP, PDP, etc.) n'ont pas besoin d'être modifiées, le seul composant qui doit être modifié afin d'utiliser le module de délégation est le gestionnaire de contexte lui-même. L'inconvénient de cette approche est que nous ne savons pas encore comment nous pouvons prendre en compte la catégorie déléguée au cours d'une évaluation d'autorisation de XACML.
- *Le module de délégation incorporé dans le PIP* (choix que nous avons privilégié) : l'avantage est que nous pouvons récupérer les données liées aux sujets, ressources et environnements afin de déterminer la délégation des catégories, car le PIP est le seul composant qui pourra communiquer directement avec différents sources de données. Les autres applications existantes n'ont pas besoin d'être changées. Le PIP appellera le module pour récupérer la délégation quand le PDP exigera plus d'informations afin de prendre une décision pour une requête d'accès transitive. Le positionnement du module de délégation au sein de l'architecture XACML est représenté dans la figure 4.3.

Mécanisme de l'exécution de l'architecture XACML étendu avec la délégation. Dans ce qui suit, nous examinons les interactions entre les différents modules.

- Étape 1, la demande d'accès de l'utilisateur est interceptée par le PEP.
- Étape 2, la demande et les attributs sont transférés au PDP ; si le PDP détermine, à partir de la politique et de la requête d'accès, que l'organisation de l'utilisateur u est différente de celle de la ressource, alors la délégation de la catégorie est nécessaire. Il demande ensuite ces informations au PIP.
- Étape 3-5, le PIP va demander au module de délégation de déterminer une nouvelle catégorie pour l'utilisateur u dans l'organisation de la ressource. Ce module contient le graphe de délégation et l'historique des délégations précédentes. Il vérifie tout d'abord dans l'historique de délégations de l'utilisateur u, si la catégorie a été précédemment déléguée à cet utilisateur dans cette organisation ; le module retourne alors cette catégorie au PDP. Sinon, il va vérifier la correspondance de la catégorie de l'utilisateur dans l'organisation de la ressource dans le graphe de délégation. Par la suite, il retourne la nouvelle catégorie trouvée au PDP.

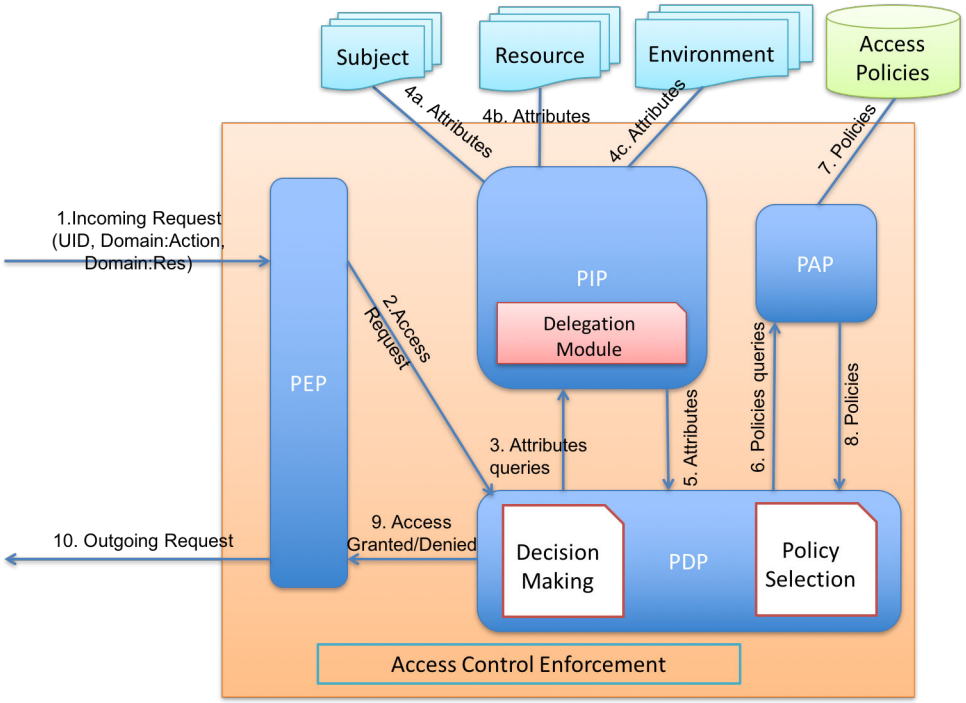


FIGURE 4.3 – Emplacement du module de délégation au sein de l'architecture XACML.

- Étape 6-8, le PDP récupère la politique de sécurité de l'organisation possédant la ressource qui correspond à la demande d'accès effectuée par l'utilisateur u.
- Étape 9, le PDP prend une décision et la retourne au PEP.
- Étape 10, si la permission demandée est acceptée, le PEP transmet la requête au service transitivement invoqué. Sinon le PEP déclenche une exception d'accès refusé.

4.4 Implémentation du cas d'étude

Nous avons vu la modélisation (design time) des politiques de contrôle d'accès ainsi que la technique d'analyse automatique liée, mais ce ne sont que les premières étapes importantes pour la production des applications SOA de haute qualité. Dans cette section, nous discutons de la mise en œuvre en temps d'exécution (run-time) du modèle D-OrBAC en étendant l'architecture XACML standard. Afin de l'expliquer d'une manière plus

concrète, nous reprenons notre exemple sur le centre médical [Voir la section 1.2.1].

4.4.1 Implémentation et mise en œuvre de la politique

L'objectif principal est non seulement que de fournir une solution efficace et adaptative pour le contrôle d'accès dans le contexte des services Web, mais aussi d'accroître les capacités et les aptitudes des outils standards existants tel que XACML en préservant à la fois la fonctionnalité souhaitée et la sécurité requise du système. Il existe plusieurs implémentations de l'architecture XACML, par exemple NDG-XACML [2], Sun-XACML [3] et Balana XACML [4].

Nous mettons en place le mécanisme de mise en œuvre de la politique de contrôle d'accès déjà présenté dans les sections précédentes, sur la plateforme de *WSO2 Identity Server* [5], laquelle fait partie d'une open source middleware supportant le développement d'applications SOA. Elle offre un support pour l'administration et l'application des politiques XACML ; en particulier, elle nous permet de personnaliser des PAP, PEP et PDP. Le WSO2 Identity Server dispose également d'un mécanisme de mise en cache des valeurs d'attribut et des décisions d'autorisation qui permet d'éviter de répéter la même opération plusieurs fois, améliorant ainsi les performances de calcul des autorisations pour des chaînes d'invocations de service longues.

Des administrateurs de politique peuvent spécifier des politiques de contrôle d'accès en utilisant un fichier XML ou une interface graphique. Cette dernière permet de définir des politiques en utilisant un langage de haut niveau, qui élimine des détails de XML et la syntaxe XACML. Tous les modules et les interactions représentés sur la figure 4.3 sont pris en charge par WSO2 Identity Server excepté le module complémentaire du PIP pour le traitement du graphe de délégation. Nous pouvons trouver dans le listing 4.1, l'exemple de règle de sécurité définie en XACML afin de protéger le service *Care Orders* de l'organisation *Clinical Management*.

Listing 4.1 – Règle de sécurité protégeant le service *Care Orders* de l'organisation *Clinical Management*

```
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"  PolicyId="
    CM-CareOrders" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
    combining-algorithm:first-applicable" Version="1.0">
  <Target>
    <AnyOf>
```

2. http://ndg-security.ceda.ac.uk/wiki/XACML
3. http://sunxacml.sourceforge.net/
4. https://github.com/wso2/balana
5. http://wso2.com/products/identity-server

```
            <AllOf>
               <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
                   equal">
                  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
                      string">CM:getCareOrders</AttributeValue>
                  <AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1
                      .0:resource:resource-id" Category="
                      urn:oasis:names:tc:xacml:3.0:attribute-category:resource
                      " DataType="http://www.w3.org/2001/XMLSchema#string"
                      MustBePresent="true"></AttributeDesignator>
               </Match>
            </AllOf>
         </AnyOf>
      </Target>
      <Rule Effect="Permit" RuleId="Rule-1">
         <Target>
            <AnyOf>
               <AllOf>
                  <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string
                      -equal">
                     <AttributeValue DataType="http://www.w3.org/2001/
                         XMLSchema#string">read</AttributeValue>
                     <AttributeDesignator AttributeId="
                         urn:oasis:names:tc:xacml:1.0:action:action-id"
                         Category="urn:oasis:names:tc:xacml:3.0:attribute-
                         category:action" DataType="http://www.w3.org/2001/
                         XMLSchema#string" MustBePresent="true"></
                         AttributeDesignator>
                  </Match>
               </AllOf>
            </AnyOf>
         </Target>
         <Condition>
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:any-of">
               <Function FunctionId="urn:oasis:names:tc:xacml:1.0
                   :function:string-equal"></Function>
               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
                   string">CM:Doctor</AttributeValue>
               <AttributeDesignator AttributeId="Category" Category="
                   urn:oasis:names:tc:xacml:1.0:subject-category:access-
                   subject" DataType="http://www.w3.org/2001/XMLSchema#string"
                    MustBePresent="true"></AttributeDesignator>
            </Apply>
         </Condition>
      </Rule>
      <Rule Effect="Deny" RuleId="Deny-Rule"></Rule>
   </Policy>
```

Nous avons implémenté cela en Java en exploitant l'API JDBC afin de communiquer avec les bases de données relationnelles où nous enregistrons les sujets ainsi que leurs attributs. Le module de délégation se compose de deux classes : la première est en charge de déléguer des catégories, récupérer des catégories déléguées qui sont peut-être déjà calculées et stockées dans une base de données. Elle est également responsable de la mise à jour de l'historique des délégations.
La seconde est une extension de $AbstractPIPAttributeFinder$, la classe Java abstraite d'API WSO2 qui nous permet de mettre en œuvre le module de dé-

couvreur d'attribut (PIP) étendu avec la délégation de catégories, et de le connecter sur le WSO2 Identity Server.

Afin de réaliser notre cas d'étude sur le centre médical [6], nous développons tous les services suivant la norme de service Web. C'est-à-dire que nous utilisons UDDI en tant qu'annuaire de services, WSDL pour la définition des interfaces, SOAP pour des invocations de services et XML comme format d'échanges de messages. Le Fonctionnement des services web est décrit ci-dessous :

1. le fournisseur de service envoie un fichier WSDL pour UDDI.
2. le demandeur de service contacte UDDI pour savoir qui est le fournisseur de données dont il a besoin.
3. le UDDI renvoie le fichier WSDL au demandeur de service.
4. le demandeur de service contacte le fournisseur de service utilisant le protocole SOAP (Simple Object Access Protocol).
5. le fournisseur de service valide la demande et renvoie des données structurées dans un fichier XML utilisant le protocole SOAP.
6. ce fichier XML serait validé à nouveau par le demandeur de service en utilisant un fichier XSD.

Nous avons également implémenté le PEP en utilisant le *Java Servlet Filter* pour intercepter les demandes d'autorisation, les transmettre au PDP, attendre ses décisions, et enfin les appliquer.

4.4.2 Scénario d'une invocation transitive

Le centre médical se compose de quatre organisations différentes (*Web Portal, Clinical Management, Laboratory, Patient Records*). Chaque organisation fournit différents services web (cf. figure 1.2.1) protégés par le module d'application de contrôle d'accès (Access Control Enforcement : ACM) pour ses utilisateurs et ses partenaires. Toutes les communications entre différentes organisations sont faites via le service de médiation (où se trouve le module d'application de contrôle d'accès) en utilisant un canal de communication sécurisé.

Une invocation transitive se produit quand un service impliqué doit en appeler un autre qui à son tour peut être susceptible d'invoquer un autre service afin de compléter sa tâche et de satisfaire la demande initiale. À chaque fois que la nouvelle demande d'accès arrive à l'organisation de la ressource demandée, le module d'application de contrôle d'accès filtre les accès avant de

6. Le prototype du centre médical se trouve à `http://pageperso.lif.univ-mrs.fr/~worachet.uttha/dl/MedicalCenter.zip`

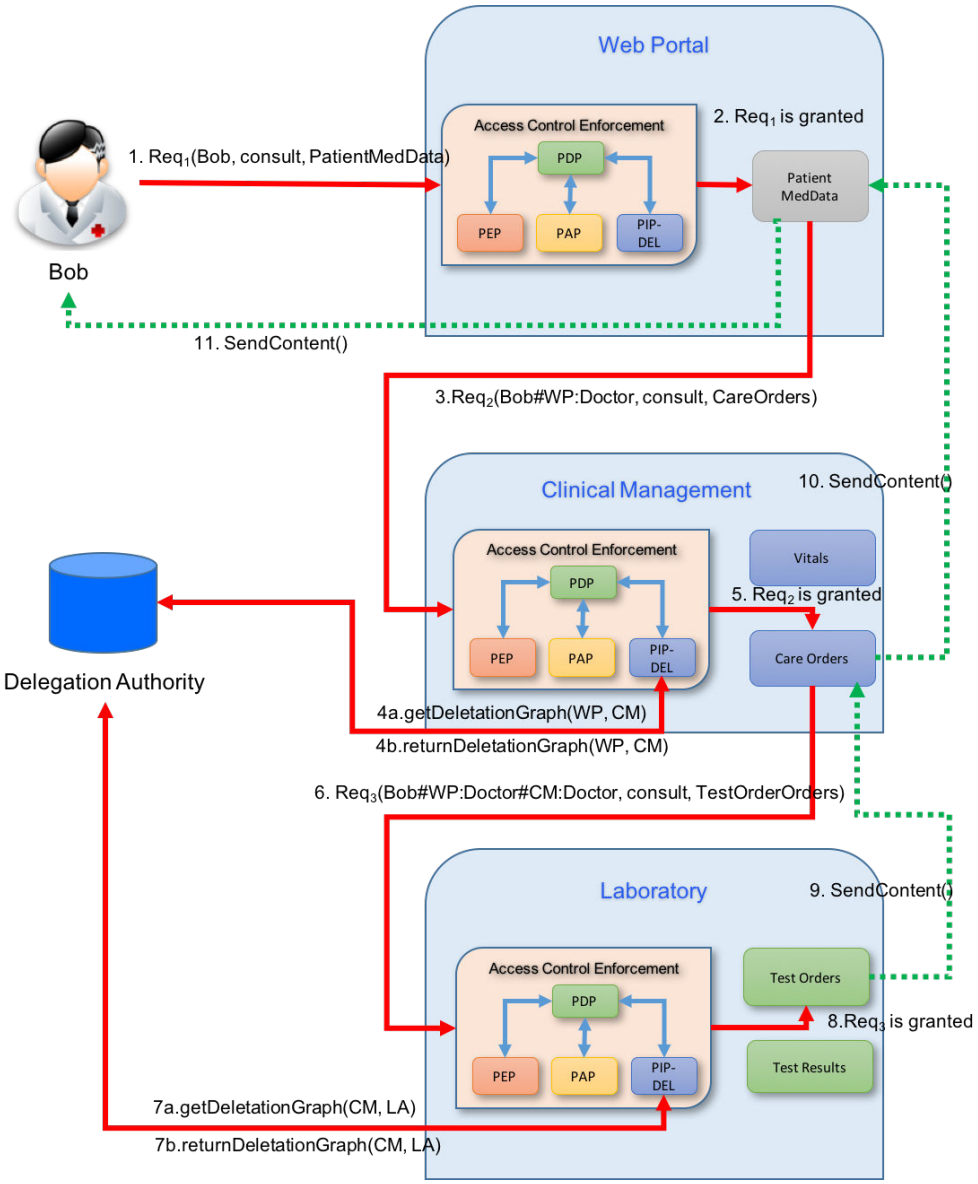


FIGURE 4.4 – Chaîne d'invocations transitive.

laisser l'utilisateur effectuer une action sur cette ressource. Puisque chaque organisation est complètement indépendante, et possède sa propre politique de contrôle d'accès, nous passons la catégorie de l'utilisateur (ou la nouvelle catégorie déléguée) en paramètre avec la demande d'accès. La détermination des droits d'accès pour chaque demande est faite avec l'aide du module de délégation incorporé dans le PIP de chaque organisation. Ce module est ca-

pable de se connecter au centre de politique de délégation où sont stockés les différents accords entre organisations partenaires. Le calcul d'une nouvelle catégorie pour un utilisateur qui fait un appel au travers d'autres organisations est basé sur ces accords.

La figure 4.4 nous montre un scénario possible de l'invocation transitive des services dans notre cas d'étude : *Bob* se connecte au portail web pour demander l'accès au service *Care Order Service* de l'organisation *Clinical Management*(CM). Quant au *Care Order service*, il doit invoquer le *Test Order Service* de l'organisation *Laboratory*(LA) afin de compléter sa tâche. La scénario est comme ci-dessous :

1. Bob s'authentifie sur le portail web et demande l'accès au $PatientMedData$, une interface qui permet de communiquer avec l'autre organisation.
2. le module ACM a intercepté la requête. Selon la politique de contrôle d'accès définie dans l'organisation $WebPortal$, Bob a la permission de consulter $PatientMedData$.
3. Bob envoie une requête d'accès $req(bob\#WP : Doctor, consult, careOrders)$ qui contient sa catégorie et ses attributs à CM.
4. le module ACM de l'organisation CM a reçu la demande de Bob. Il constate que l'organisation de Bob est différente de la sienne, la délégation est alors nécessaire. Il demande au module de délégation (PIP-DEL) d'établir une nouvelle catégorie déléguée pour Bob. Le module de délégation demande au centre de délégation le graphe de délégation définissant les accords entre CM et WP. Il renvoie la nouvelle catégorie déléguée de Bob au PDP afin de prendre une décision pour la requête d'accès de Bob.
5. la demande d'accès de Bob est autorisée, le module ACM transmet l'accès au service $CareOrder$.
6. afin que le service $CareOrder$ puisse continuer le calcul, il doit invoquer le service $Testorder$ appartenant à l'organisation $Laboratory$. Puisque $CareOrder$ appelle le service $TestOrder$ au nom de Bob, il envoie alors la nouvelle catégorie que Bob vient d'obtenir, avec la demande d'accès afin de garder la trace d'exécution, en les séparant avec le symbole "#".
7. le module ACM de LA intercepte la requête d'accès, il constate que c'est une demande d'accès traversant les organisations. Le module PIP-DEL récupère le graphe de délégation défini des accords entre LA et CM afin de déterminer la catégorie de Bob au sein de LA en utilisant la catégorie déléguée par CM.

8. la demande d'accès de Bob est acceptée, il peut donc accéder au service $TestOrder$.
9. une fois que le service $TestOrder$ a fini sa tâche, il renvoie le résultat au service $CareOrder$.
10. le service $CareOrder$ a obtenu la donnée dont il a besoin du service $TestOrder$. Il continue son travail, il renvoie le résultat au $PatientMedData$ dès qu'il a fini sa tâche.
11. le service $PatientMedData$ retourne les données demandées à Bob.

4.5 Travaux liés

XACML est utilisé dans de nombreux services et applications sous des contextes différents. Diverses extensions sont proposées.

[Lischka et al., 2009] ont proposé une extension de l'architecture XACML afin de supporter la réduction de la décision dans le système développé selon le modèle "logiciel en tant que service" ou "software as a service (SaaS)". L'objectif de leur approche est de développer un langage de spécification de la politique de contrôle d'accès basé sur XACML qui permette de diviser la politique en plusieurs blocs. D'autres politiques pourraient se référer à cet ensemble de politiques sans avoir à en connaître les détails internes. Comme chacune d'entre elles pourraient avoir sa propre notion de sujets, de ressources et d'actions, elles doivent être capables de transformer la requête provenant du domaine A en un contexte particulier du domaine B. Leur méthode est de redéfinir l'élément $PolicySet$ de XACML afin de pouvoir intégrer leur extension. Dans leur approche, ce sont les autorisations qui dépendent des autres domaines, tandis que dans la nôtre, ce sont des calculs qui dépendent des autres services appartenant aux autres organisations.

[Ardagna et al., 2010] ont proposé une autre extension de XACML ; leur but est d'apporter un contrôle d'accès basé sur des certificats en préservant la confidentialité des données. Ils utilisent XACML comme un langage de spécification de la politique de contrôle d'accès et SAML pour échanger des valeurs d'attributs attachées avec des certificats. Leur approche consiste à ajouter des éléments supplémentaires dans le langage XACML afin d'exprimer des conditions sur des certificats qu'un demandeur d'accès doit posséder, ainsi que les actions qu'il doit effectuer pour obtenir l'accès. Par la suite, ils ajoutent des éléments supplémentaires dans l'affirmation de SAML. Cela permet d'exprimer des conditions sur les actions et les attributs attachés dans un ou plusieurs certificats, associés aux preuves nécessaires. Ils étendent le PEP avec un module vérificateur de preuves pour que le PEP puisse recevoir une de-

mande d'accès attachée avec une affirmation de SAML, puis vérifier la validité de cette affirmation et la rendre à disposition du PDP afin que celui-ci puisse prendre la décision concernant cette demande. Les auteurs sont ici surtout intéressés par la confidentialité des données que les utilisateurs délivrent au moteur de contrôle d'accès afin que celui-ci calcule l'autorisation ; leur approche est donc orientée sur cet aspect tandis que la nôtre se focalise sur la protection des ressources partagées, d'éventuels accès non autorisés. Ainsi ces deux travaux pourront être très complémentaires pour protéger efficacement des données des utilisateurs et des ressources partagées.

Concernant la mise en œuvre de la politique de contrôle d'accès dans le cas des inter-domaines, [Alam et al., 2011] ont proposé un framework xDAuth, un ensemble de composants structurels permettant de protéger des ressources partagées (en particulier des services web) avec différentes organisations. Dans leur approche, chaque domaine doit donner la politique de délégation au centre délégation. Quand il y a une demande d'accès d'un utilisateur, le fournisseur de services va la rediriger vers ce centre pour l'authentification et l'autorisation. Quant au centre de délégation, au lieu d'authentifier cet utilisateur lui-même, il le renvoie à sa propre organisation pour s'authentifier, puis revenir au centre de délégation avec ses attributs pour déterminer sa permission. Leur approche est adaptée dans le cas où l'organisation de l'utilisateur et l'organisation de la ressource possèdent un accord commun. Néanmoins, cette méthode ne pourra pas s'appliquer dans le cas des invocations transitives (comme nous l'avons étudié dans ce travail) où l'organisation de l'utilisateur peut ne pas avoir une collaboration avec l'organisation du service qui est plus loin dans la chaine d'invocation, ce qui pourrait produire une erreur d'autorisation indirecte.

Une autre solution pour le problème d'accès transitif a été proposée dans [Karp and Li, 2010] en utilisant le modèle de contrôle d'accès basé sur l'autorisation (ZBAC). Dans leur approche, un utilisateur doit s'authentifier à son domaine, puis récupère la liste des droits d'accès qui lui sont autorisé. Il peut ensuite déléguer ces droits individuellement à des processus exécutant en son nom. Afin de résoudre le problème d'invocations transitives, l'utilisateur délègue sa permission d'utilisation de son service au contrôleur de son domaine (*i.e.* administrateur de sécurité). Par la suite, c'est à la charge des contrôleurs de chaque domaine de s'arranger afin de gérer les autorisations déléguées et de créer un lien de confiance entre eux. Des utilisateurs peuvent obtenir des autorisations déléguées lorsqu'ils s'authentifient à leurs domaines sous la décision des administrateurs. Dans cette méthode, des délégations de droits d'accès sont affectées directement aux sujets ce qui est différent de notre approche où des droits d'accès sont attribués aux sujets via leur affectation à des

catégories, ce qui est beaucoup plus flexible dans une grande organisation au sein de laquelle les sujets à gérer sont plus nombreux.

Conclusion et Perspectives

La spécification des politiques de sécurité est un problème crucial lorsqu'il est question de protéger des données sensibles partagées parmi différentes organisations indépendantes. Seules les personnes autorisée doivent pouvoir accéder aux informations demandées, tout accès indésirable devant être empêché. Toutefois, un très haut niveau de sécurité est parfois incompatible avec la meilleure fonctionnalité ; trouver le compromis entre le fonctionnement désiré et l'exigence de sécurité devient alors crucial afin de rendre le système capable de répondre aux besoins des utilisateurs tout en protégeant les données sensibles.

Nous avons proposé un nouveau modèle de contrôle d'accès (D-OrBAC : Distributed-Organisation Based Access Control), une extension du modèle de contrôle d'accès basé sur des organisations permettant de protéger des ressources partagées. Dans ce modèle, nous avons associé des sujets avec des catégories (qui peuvent être déléguées aux autres sujets de différentes organisations) afin de résoudre le problème d'autorisation d'accès aux ressources partagées dans un environnement distribué avec plusieurs organisations partenaires, chacune de ces organisations possédant sa propre politique de contrôle d'accès. Nous avons utilisé un langage formel basé sur la logique du premier ordre pour représenter les différentes relations de notre modèle : utilisateurs-catégorie, catégories-permissions.

Par la suite, nous avons utilisé Datalog pour expérimenter les comportements du système (spécifié par notre modèle de contrôle d'accès avant son déploiement). Nous avons montré la correspondance entre la politique de contrôle d'accès décrite par le modèle D-OrBAC et le programme Datalog simulant le comportement de l'évaluation des requêtes d'accès. Cela afin de tester si notre modèle peut gérer des demandes d'accès aux ressources partagées, en particulier dans le cas d'un appel transitif. Soulignons que le problème d'accès transitif apparait par exemple quand un service invoque un autre service afin de terminer la tâche initiale ; même si un utilisateur peut avoir l'accès au service original, cela ne signifie pas qu'il a le droit d'invoquer transitivement le second service, cela pourrait en effet conduire à des erreurs d'autorisations

indirectes. Afin d'expliciter notre approche d'une manière plus concrète et de prouver que notre modèle est assez expressif, nous avons validé notre modèle au travers deux cas d'études : le centre médical et le centre de recherche. L'avantage d'utiliser le moteur Datalog comme une simulation de système est qu'il ne nécessite pas l'implémentation d'un prototype pour surveiller le fonctionnement du système, cela nous aide à réduire les coûts de correction des erreurs de l'implémentation et à éviter les difficultés des tests des applications dans le cas du système distribué.

Ensuite, nous avons vérifié si la politique de contrôle d'accès spécifiée par notre modèle répond aux propriétés nécessaires telles que la totalité, la terminaison et la consistance, ce qui assure la confidentialité des informations sensibles. Nous avons prouvé ces propriétés en utilisant le système de réécriture des termes. Les politiques sont représentées comme des ensembles de règles de réécriture, tandis que les demandes d'accès sont représentées comme des termes algébriques dont l'évaluation produit des décisions d'autorisation. Ainsi, l'évaluation de la demande est effectuée par la réduction des termes en une forme normale. Nous avons présenté également l'outil CiME que nous pouvons utiliser pour analyser automatiquement un système de réécriture. Avec l'outil CiME, nous pouvons facilement vérifier et certifier des propriétés de système de réécriture telles que la terminaison et la confluence locale, ce qui nous aide à prouver les propriétés associées de la politique de contrôle d'accès.

Nous avons montré que notre approche satisfait à la fois la fonctionnalité souhaitée et la sécurité requise du système. Pour cela, nous avons implémenté nos cas d'étude par des services web en utilisant Java et les technologies standards. Nous avons montré des correspondances entre notre modèle et différentes technologies déjà connues que nous avons choisi afin de prouver que notre approche peut s'appliquer en pratique et que nous pouvons réutiliser des outils standards existants tel que l'architecture XACML pour la mise en œuvre du mécanisme de contrôle d'accès de notre modèle. Nous avons également montré notre extension de XACML permettant de déléguer les catégories des utilisateurs afin d'assurer le fonctionnement de l'autorisation des requêtes d'accès transitifs dans les architectures SOA.

Dans notre modèle de contrôle d'accès, nous n'avons considéré que des organisations avec des topologies statiques. Dans des travaux futurs, nous avons l'intention d'étudier la façon d'étendre nos techniques à des applications SOA avec des topologies dynamiques [Mecella et al., 2006], qui sont particulièrement utiles dans la pratique pour répondre aux besoins des modèles de gestion (business model) d'aujourd'hui. Il serait ensuite intéressant de traiter explicitement des questions administratives dans la politique de sécurité

en s'inspirant du modèle Ad-OrBAC [Cuppens and Miège, 2003] pour contrôler des affectations de différentes entités dans le modèle OrBAC. Un modèle administratif permettrait en effet dans notre approche de contrôler la modification de la politique, nous pourrions par exemple définir qui pourrait affecter ou enlever des permissions des utilisateurs. Il serait également intéressant de tenir compte des questions sur la négociation de confiance [Lee et al., 2007] entre des partenaires, car dans le modèle de gestion d'aujourd'hui, la plupart des interactions se produisent entre entités inconnues (n'ayant aucune relation préexistante et ne partageant pas le même domaine de sécurité). La négociation de confiance aidera des organisations à mieux définir des accords entre elles pour mettre en place un ensemble de délégations partagées.

Annexe A

Implémentations de la politique de sécurité

A.1 Cas d'étude 1 : centre médical

A.1.1 Programme Datalog représentant le centre médical

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
empower(Org,U,Category):- org(U,Org),
cat(Org,U,Category).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
depends_on(vitals_service, vitals_service).
depends_on(careOrders_service,
        testOrders_service)
depends_on(testOrders_service,
                testOrders_service).
depends_on(testResults_service,
        testResults_service).
depends_on(patientHistory_service,
        patientHistory_service).
depends_on(showProfile_service,
        showProfile_service).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
belong(vitals_service, cm).
belong(careOrders_service, cm).
belong(testOrders_service, la).
belong(testResults_service, la).
belong(patientHistory_service, pr).
belong(showProfile_service, wp).
belong(patientMedData_service, wp).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
delegate(cm, cm_doctor, wp, wp_doctor).
delegate(cm, cm_nurse, wp, wp_nurse).
delegate(la, la_clinician, cm, cm_doctor).
delegate(la, la_billing, wp, wp_doctor).
delegate(pr, pr_clinician, wp, wp_doctor).
```

```
delegate(pr, pr_clinician, la, la_clinician).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
org(alice, wp). org(bob, wp). org(ceci, cm).
org(david, cm). org(damien, cm). org(eric, la).
org(elena, la). org(francois, la). org(gaspard, pr).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Web Portal %%
cat(wp,U,wp_nurse) :- role(U, nurse).
cat(wp,U,wp_doctor) :- role(U, doctor).

role(alice, nurse). role(bob, doctor).

permission(wp,wp_doctor,read, patientMedData_service).
permission(wp,wp_doctor,read, showProfile_service).
permission(wp,wp_nurse,read,  showProfile_service).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Clinical Management %%

cat(cm,U,cm_nurse) :- diploma(U, nurse).
cat(cm,U,cm_nurse) :- speciality(U, dentalNurse).
cat(cm,U,cm_doctor) :- diploma(U, medecine).
cat(cm,U,cm_doctor) :- speciality(U, physician).
cat(cm,U,cm_doctor) :- speciality(U, dentist).
cat(cm,U,cm_senior_doctor) :- cat(cm,U,cm_doctor),
    experience(U,Exp), Exp >= 5.

diploma(ceci, nurse).
diploma(david, medecine).
speciality(catherin, dentalNurse).
speciality(david, physician).
speciality(damien, dentist).
experience(david,10).
experience(damien, 4).

permission(cm,cm_doctor,read, vitals_service).
permission(cm,cm_doctor,read, careOrders_service).
permission(cm,cm_senior_doctor,read, careOrders_service).
permission(cm,cm_senior_doctor,modify, careOrders_service).
permission(cm,cm_nurse,read,vitals_service).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Laboratory %%

cat(la,U,la_clinician):- diploma(U,chemistry).
cat(la,U,la_clinician):- diploma(U, biology).
cat(la,U,la_billing):- diploma(U,accounting).

diploma(eric, chemistry).
diploma(elena, biology).
diploma(francois, accounting).

permission(la,la_clinician,read,testOrders_service).
permission(la,la_billing,read,testResults_service).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Patient Record %%

cat(pr,U,pr_clinician):- role(U,clinician).
role(gaspard, clinician).
```

```
permission(pr,pr_clinician,read, patientHistory_service).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Concrete permission %%%

is_permitted(U, A, R):-
        empower(Org, U, C), permission(Org, C, A, R).

is_permitted(U, A, R):-
        empower(Org, U, C),
        is_permitted_aux( Org, C, A, R).

%%% 1 delegation %%%
is_permitted_aux( Org, C, A, R):-
        belong(R, Org1),
        Org \== Org1,
        delegate(Org1, C1, Org, C),
        permission(Org1, C1, A, R),
        depends_on(R,R1),
    R == R1.

%%% >= 2 delegations %%%
is_permitted_aux( Org, C, A, R):-
        belong(R, Org1),
        Org \== Org1,
        delegate(Org1, C1, Org, C),
        permission(Org1, C1, A, R),
        depends_on(R,R1),
        R \== R1,
        is_permitted_aux(Org1, C1, A, R1).
```

A.1.2 Système de réécriture spécifiant la politique du centre médical

```
(VAR org org1 org2 org3 u category a r r1 r2 c c1 c2 c3 exp x y l l1 l2 t)
(RULES
        if(true,x,y) -> x
        if(false,x,y) -> y
        or(true,y) -> true
        or(false,y) -> y
        and(true,y) -> y
        and(false,y) -> false
        not(true) -> false
        not(false) -> true
        eq(0,0) -> true
        eq(0,s(x)) -> false
        eq(s(x),0) -> false
        eq(s(x),s(y)) -> eq(x,y)
        le(0,y) -> true
        le(s(x),0) -> false
        le(s(x),s(y)) -> le(x,y)
        ge(x,0) -> true
        ge(0,s(y)) -> false
        ge(s(x),s(y)) -> ge(x,y)
        is_member(x, nil) -> false
        is_member(x, cons(y,l)) -> ifeq(equal(x,y),x,l)
        ifeq(true,x,l) -> true
        ifeq(false, x, l) -> is_member(x,l)
        head(cons(x, l)) -> x
```

```
head(nil) -> nil
tail(nil) -> nil
tail(cons(x, l)) -> l
append(l1,l2) -> ifappend(l1,l2,l1)
ifappend(l1,l2,nil) -> l2
ifappend(l1,l2,cons(x,l)) -> cons(x,append(l,l2))
is_empty(nil,nil) -> true
is_empty(cons(x,l),nil)-> false

equal(showProfile_service,showProfile_service) -> true
equal(showProfile_service,patientMedData_service) -> false
equal(showProfile_service,careOrders_service) -> false
equal(showProfile_service,testOrders_service) -> false
equal(showProfile_service,testResults_service) -> false
equal(showProfile_service,patientHistory_service) -> false
equal(showProfile_service,vitals_service) -> false

equal(patientMedData_service,patientMedData_service) -> true
equal(patientMedData_service,showProfile_service) -> false
equal(patientMedData_service,careOrders_service) -> false
equal(patientMedData_service,testOrders_service) -> false
equal(patientMedData_service,testResults_service) -> false
equal(patientMedData_service,patientHistory_service) -> false
equal(patientMedData_service,vitals_service) -> false

equal(vitals_service,vitals_service) -> true
equal(vitals_service,careOrders_service) -> false
equal(vitals_service,testOrders_service) -> false
equal(vitals_service,testResults_service) -> false
equal(vitals_service,patientHistory_service) -> false
equal(vitals_service,showProfile_service) -> false
equal(vitals_service,patientMedData_service) -> false

equal(careOrders_service,careOrders_service) -> true
equal(careOrders_service,vitals_service) -> false
equal(careOrders_service,testOrders_service) -> false
equal(careOrders_service,testResults_service) -> false
equal(careOrders_service,patientHistory_service) -> false
equal(careOrders_service,showProfile_service) -> false
equal(careOrders_service,patientMedData_service) -> false

equal(testOrders_service,testOrders_service) -> true
equal(testOrders_service,vitals_service) -> false
equal(testOrders_service,careOrders_service) -> false
equal(testOrders_service,testResults_service) -> false
equal(testOrders_service,patientHistory_service) -> false
equal(testOrders_service,showProfile_service) -> false
equal(testOrders_service,patientMedData_service) -> false

equal(testResults_service,testResults_service) -> true
equal(testResults_service,vitals_service) -> false
equal(testResults_service,careOrders_service) -> false
equal(testResults_service,testOrders_service) -> false
equal(testResults_service,patientHistory_service) -> false
equal(testResults_service,showProfile_service) -> false
equal(testResults_service,patientMedData_service) -> false

equal(patientHistory_service,patientHistory_service) -> true
equal(patientHistory_service,vitals_service) -> false
equal(patientHistory_service,careOrders_service) -> false
equal(patientHistory_service,testOrders_service) -> false
equal(patientHistory_service,testResults_service) -> false
```

```
equal(patientHistory_service,showProfile_service) -> false
equal(patientHistory_service,patientMedData_service) -> false

equal(cm,cm) -> true
equal(cm,wp) -> false
equal(cm,la) -> false
equal(cm,pr) -> false

equal(wp,wp) -> true
equal(wp,cm) -> false
equal(wp,la) -> false
equal(wp,pr) -> false

equal(la,la) -> true
equal(la,wp) -> false
equal(la,cm) -> false
equal(la,pr) -> false

equal(pr,pr) -> true
equal(pr,wp) -> false
equal(pr,la) -> false
equal(pr,cm) -> false

equal(deny,deny) -> true
equal(grant, grant) -> true
equal(deny, grant) -> false
equal(grant, deny) -> false

depends_on(vitals_service) -> nil
depends_on(careOrders_service) -> cons(testOrders_service, cons(
    patientHistory_service, nil))
depends_on(testOrders_service) -> cons(vitals_service,nil)
depends_on(testResults_service) -> nil
depends_on(patientHistory_service) -> nil
depends_on(showProfile_service) -> nil

dp_action(careOrders_service, testOrders_service) -> read
dp_action(careOrders_service, patientHistory_service) -> read
dp_action(testOrders_service, vitals_service) -> read

belong_to(vitals_service) -> cm
belong_to(careOrders_service) -> cm
belong_to(testOrders_service) -> la
belong_to(testResults_service) -> la
belong_to(patientHistory_service) -> pr
belong_to(showProfile_service) -> wp
belong_to(patientMedData_service) -> wp

delegate(wp, wp_doctor, cm) -> cm_doctor
delegate(wp, wp_doctor, la) -> la_billing
delegate(wp, wp_doctor, pr) -> pr_clinician

delegate(wp, wp_nurse, cm) -> cm_nurse
delegate(wp, wp_nurse, la) -> la_unknown
delegate(wp, wp_nurse, pr) -> pr_unknown

delegate(wp, wp_unknown, la) -> la_unknown
delegate(wp, wp_unknown, cm) -> cm_unknown
delegate(wp, wp_unknown, pr) -> pr_unknown

delegate(cm, cm_doctor, la) -> la_clinician
delegate(cm, cm_doctor, wp) -> wp_unknown
```

```
delegate(cm, cm_doctor, pr) -> pr_clinician

delegate(cm, cm_nurse, la) -> la_unknown
delegate(cm, cm_nurse, wp) -> wp_unknown
delegate(cm, cm_nurse, pr) -> pr_unknown

delegate(cm, m_doctor_senior, la) -> la_clinician
delegate(cm, m_doctor_senior, wp) -> wp_unknown
delegate(cm, m_doctor_senior, pr) -> pr_unknown

delegate(cm, cm_unknown, la) -> la_unknown
delegate(cm, cm_unknown, wp) -> wp_unknown
delegate(cm, cm_unknown, pr) -> pr_unknown

delegate(la, la_clinician, pr) -> pr_clinician
delegate(la, la_clinician, cm) -> cm_doctor
delegate(la, la_clinician, wp) -> wp_unknown

delegate(la, la_billing, pr) -> pr_unknown
delegate(la, la_billing, cm) -> cm_unknown
delegate(la, la_billing, wp) -> wp_unknown

delegate(la, la_unknown, cm) -> cm_unknown
delegate(la, la_unknown, wp) -> wp_unknown
delegate(la, la_unknown, pr) -> pr_unknown

delegate(pr, pr_clinician, la) -> la_unknown
delegate(pr, pr_clinician, cm) -> cm_unknown
delegate(pr, pr_clinician, wp) -> wp_unknown

delegate(pr, pr_unknown, la) -> la_unknown
delegate(pr, pr_unknown, wp) -> wp_unknown
delegate(pr, pr_unknown, cm) -> wm_unknown

organization(alice) -> wp
organization(bob) -> wp
organization(ceci) -> cm
organization(david) -> cm
organization(damien) -> cm
organization(eric) -> la
organization(elena) -> la
organization(francois) -> la
organization(gaspard) -> pr

empower(wp,u) -> if (role(u, nurse), wp_nurse, if(role(u, doctor),
    wp_doctor, wp_unknown))

role(alice, nurse) -> true
role(bob, nurse) -> false
role(bob, doctor) -> true
role(alice, doctor) -> false
permission(wp, wp_doctor, read) -> cons(patientMedData_service,cons
    (showProfile_service,nil))
permission(wp, wp_nurse, read) -> cons(showProfile_service,nil)
permission(wp, wp_unknown, read) -> nil

empower(cm,u) -> if(or(diploma(u, nurse),specialist(u, dentalNurse)
    ), cm_nurse,if(or(or(diploma(u, medicine),specialist(u,
    physician)),specialist(u, dentist)), cm_doctor,if(ge(experience
    (u), s(s(s(s(s(0))))),cm_senior_doctor,cm_unknown)))
diploma(ceci, nurse) -> true
diploma(david, medicine) -> true
```

```
specialist(catherin, dentalNurse) -> true
specialist(david, physician) -> true
specialist(damien, dentist) -> true
experience(david) -> s(s(s(s(s(s(s(s(s(0)))))))))
experience(damien) -> s(s(s(s(0))))

permission(cm,cm_doctor,read) -> cons(vitals_service, cons(
    careOrders_service, nil))
permission(cm,cm_senior_doctor,read) -> cons(careOrders_service,
    nil)
permission(cm,cm_senior_doctor,modify) -> cons(careOrders_service,
    nil)
permission(cm,cm_nurse,read) -> cons(vitals_service,nil)
permission(cm,cm_unknown,read) -> nil

empower(la,u) -> if(or(diploma(u,chemistry),diploma(u, biology)),
    la_clinician,if(diploma(u, accounting),la_billing,la_unknown))
diploma(eric, chemistry) -> true
diploma(elena, biology) -> true
diploma(francois, accounting) -> true

permission(la,la_clinician,read) -> cons(testOrders_service,nil)
permission(la,la_billing,read ) -> cons(testResults_service, nil)
permission(la,la_unknown,read ) -> nil

empower(pr,u) -> if (role(u,clinician),pr_clinician,pr_unknown)
role(gaspard, clinician) ->  true
permission(pr,pr_clinician,read) -> cons(patientHistory_service,
    nil)
permission(pr,pr_unknown,read) -> nil

authorize(org, c, a, r) -> if(is_member(r,permission(org, c, a)),
    grant, deny)
is_permitted(u, a, r) -> if(is_member(deny,perm(u, a,r, nil)), deny
    , grant)
perm(u, a, r, l) -> if(equal(organization(u),belong_to(r)),
    is_permitted_local(u, a, r, l),is_permitted_global(organization
    (u),empower(organization(u),u),a,r,l))
is_permitted_local(u, a, r, l) -> append(l, cons(authorize(
    organization(u), empower(organization(u),u),a,r),nil))

is_permitted_global(org, c, a, r, l) -> if_independent(is_empty(
    depends_on(r), nil), org, c, a, r, l)

if_independent(true, org, c, a, r, l) -> append(l, cons(authorize(
    belong_to(r), delegate(org, c ,belong_to(r)),a,r), nil))

if_independent(false, org, c, a, r, l) -> if_one_dep(is_empty(tail(
    depends_on(r)), nil), org, c, a, r, l)

if_one_dep(true, org, c, a, r, l) -> is_permitted_global(belong_to(
    r), delegate(org, c ,belong_to(r)),dp_action(r, head(depends_on
    (r))),head(depends_on(r)),append(l, cons(authorize(belong_to(r)
    ,delegate(org, c ,belong_to(r)),a,r),nil)))

if_one_dep(false, org, c, a, r, l) -> append(is_permitted_global(
    belong_to(r), delegate(org, c ,belong_to(r)),dp_action(r, head(
    depends_on(r))),head(depends_on(r)),append(l, cons(authorize(
    belong_to(r),delegate(org, c ,belong_to(r)),a,r),  nil))),
    is_permitted_aux(belong_to(r),delegate(org, c ,belong_to(r)),r,
     tail(depends_on(r)),nil))
```

```
        is_permitted_aux(org,c, r, t,q) -> if_last_dep(is_empty(tl(t),nil),
            org,c, r, t,q)

        if_last_dep(true,org,c, r, t,q) ->is_permitted_global(org,c,
            dp_action(r, head(t)),head(t),q)

        if_last_dep(false,org,c, r, t,q) -> append(is_permitted_global(org,
            c,dp_action(r, head(t)),head(t),q),is_permitted_aux(org,c,r,
            tail(t),q)))

)
```

A.2 Cas d'étude 2 : centre de recherche

A.2.1 Programme Datalog représentant le centre de recherche

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
empower(Org,U,Category):- org(U,Org),
cat(Org,U,Category).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
depends_on(createRequest, noAction, noDepend, noAction, noDepend).
depends_on(sendRequest, approve, approveRequest, noAction, noDepend).
depends_on(update, update, updateBudget, update, updateMissionHistory).
depends_on(approveRequest, read, getBudget, consult, getMissionHistory).
depends_on(getBudget, noAction, noDepend, noAction, noDepend).
depends_on(updateBudget, noAction, noDepend, noAction, noDepend).
depends_on(getMissionHistory, noAction, noDepend, noAction, noDepend).
depends_on(updateMissionHistory, noAction, noDepend, noAction, noDepend).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 belong(createRequest, sec).
 belong(sendRequest, sec).
 belong(update, sec).
 belong(approveRequest, adm).
 belong(getBudget, acc).
 belong(updateBudget, acc).
 belong(getMissionHistory, itd).
 belong(updateMissionHistory, itd).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 delegate(adm, adm_director, sec, sec_administrativeSecretary).
 delegate(acc, acc_budgetManager, adm, adm_director).
 delegate(itd, itd_director, adm, adm_director).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 org(alice, sec).
 org(anna, sec).
 org(bob, adm).
 org(billy, adm).
 org(celine, acc).
 org(chirst, acc).
 org(david, itd).
 org(daniel, itd).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Secretarys Office %%
```

```
cat(sec,U,sec_administrativeSecretary) :- task(U, administrative).
cat(sec,U,sec_officeSecretary) :- task(U, office).
 task(alice, administrative).
 task(anna, office).

 permission(sec,sec_administrativeSecretary,perform,createRequest).
 permission(sec,sec_administrativeSecretary,perform,sendRequest).
 permission(sec,sec_officeSecretary,perform,update).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Administrative Department %%
cat(adm,U,adm_director) :- position(U, director).
cat(adm,U,adm_projectManager) :- position(U, projectManager).

 position(bob, director).
 position(billy, projetManager).

 permission(adm,adm_projectManager,consult,approuveRequest).
 permission(adm,adm_director,approve,approveRequest).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Accounting Department %%
cat(acc,U,acc_accountingDirector):- role(U,accountingDirector).
cat(acc,U,acc_budgetManager):- role(U, budgetManager).

 role(celine, accountingDirector).
 role(chirst, budgetDirector).

 permission(acc,acc_accountingDirector,update,updateBudget).
 permission(acc,acc_budgetManager,read,getBudget).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% IT Department %%

cat(itd,U,itd_director):- role(U,director).
cat(itd,U,itd_systemAdmin):- role(U,systemAdmin).

 role(david, systemAdmin).
 role(daniel, director).

 permission(itd,itd_director,consult,getMissionHistory).
 permission(itd,itd_systemAdmin,update,updateMissionHistory).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
is_permitted(U, A, R):-
        empower(Org, U, C), permission(Org, C, A, R).

is_permitted(U, A, R):-
        empower(Org1, U, C1),
        is_permitted_aux( Org1, C1, A, R).

%%% The last delegation %%%
is_permitted_aux( Org1, C1, A, R):-
        belong(R, Org2),
        Org1 \== Org2,
        delegate(Org2, C2, Org1, C1),
        permission(Org2, C2, A, R),
        depends_on(R,A1,D1, A2, D2, A3, D3, A4, D4, A5, D5),
    D1 == noDepend.

%%% >= 2 delegation (1 dependency)%%%
is_permitted_aux( Org1, C1, A, R):-
```

```
        belong(R, Org2),
        Org1 \== Org2,
        delegate(Org2, C2, Org1, C1),
        permission(Org2, C2, A, R),
        depends_on(R,A1,D1, A2, D2, A3, D3, A4, D4, A5, D5),
        D1 \== noDepend,
        D2 == noDepend,
        is_permitted_aux(Org2, C2, A1, D1).

%%% >= 2 delegation (2 dependency)%%%
is_permitted_aux( Org1, C1, A, R):-
        belong(R, Org2),
        Org1 \== Org2,
        delegate(Org2, C2, Org1, C1),
        permission(Org2, C2, A, R),
        depends_on(R,A1,D1, A2, D2, A3, D3, A4, D4, A5, D5),
        D1 \== noDepend,
        D2 \== noDepend,
        is_permitted_aux(Org2, C2, A1, D1),
        is_permitted_aux(Org2, C2, A2, D2).
```

A.2.2 Système de réécriture spécifiant la politique du centre de recherche

```
(VAR org org1 org2 org3 u category a r r1 r2 c c1 c2 c3 exp x y l l1 l2 t)
(RULES
        if(true,x,y) -> x
        if(false,x,y) -> y
        or(true,y) -> true
        or(false,y) -> y
        and(true,y) -> y
        and(false,y) -> false
        not(true) -> false
        not(false) -> true
        eq(0,0) -> true
        eq(0,s(x)) -> false
        eq(s(x),0) -> false
        eq(s(x),s(y)) -> eq(x,y)
        le(0,y) -> true
        le(s(x),0) -> false
        le(s(x),s(y)) -> le(x,y)
        ge(x,0) -> true
        ge(0,s(y)) -> false
        ge(s(x),s(y)) -> ge(x,y)
        is_member(x, nil) -> false
        is_member(x, cons(y,l)) -> ifeq(equal(x,y),x,l)
        ifeq(true,x,l) -> true
        ifeq(false, x, l) -> is_member(x,l)
        head(cons(x, l)) -> x
        head(nil) -> nil
        tail(nil) -> nil
        tail(cons(x, l)) -> l
        append(l1,l2) -> ifappend(l1,l2,l1)
        ifappend(l1,l2,nil) -> l2
        ifappend(l1,l2,cons(x,l)) -> cons(x,append(l,l2))
        is_empty(nil,nil) -> true
        is_empty(cons(x,l),nil)-> false


        equal(createRequest, createRequest) -> true
```

```
equal(createRequest, sendRequest) -> false
equal(createRequest, update) -> false
equal(createRequest, approveRequest) -> false
equal(createRequest, getBudget) -> false
equal(createRequest, updateBudget) -> false
equal(createRequest, getMissionHistory) -> false
equal(createRequest, updateMissionHistory) -> false

equal(sendRequest, createRequest) -> false
equal(sendRequest, sendRequest) -> true
equal(sendRequest, update) -> false
equal(sendRequest, approveRequest) -> false
equal(sendRequest, getBudget) -> false
equal(sendRequest, updateBudget) -> false
equal(sendRequest, getMissionHistory) -> false
equal(sendRequest, updateMissionHistory) -> false

equal(update, createRequest) -> false
equal(update, sendRequest) -> false
equal(update, update) -> true
equal(update, approveRequest) -> false
equal(update, getBudget) -> false
equal(update, updateBudget) -> false
equal(update, getMissionHistory) -> false
equal(update, updateMissionHistory) -> false

equal(approveRequest, createRequest) -> false
equal(approveRequest, sendRequest) -> false
equal(approveRequest, update) -> false
equal(approveRequest, approveRequest) -> true
equal(approveRequest, getBudget) -> false
equal(approveRequest, updateBudget) -> false
equal(approveRequest, getMissionHistory) -> false
equal(approveRequest, updateMissionHistory) -> false

equal(getBudget, createRequest) -> false
equal(getBudget, sendRequest) -> false
equal(getBudget, update) -> false
equal(getBudget, approveRequest) -> false
equal(getBudget, getBudget) -> true
equal(getBudget, updateBudget) -> false
equal(getBudget, getMissionHistory) -> false
equal(getBudget, updateMissionHistory) -> false

equal(updateBudget, createRequest) -> false
equal(updateBudget, sendRequest) -> false
equal(updateBudget, update) -> false
equal(updateBudget, approveRequest) -> false
equal(updateBudget, getBudget) -> false
equal(updateBudget, updateBudget) -> true
equal(updateBudget, getMissionHistory) -> false
equal(updateBudget, updateMissionHistory) -> false

equal(getMissionHistory, createRequest) -> false
equal(getMissionHistory, sendRequest) -> false
equal(getMissionHistory, update) -> false
equal(getMissionHistory, approveRequest) -> false
equal(getMissionHistory, getBudget) -> false
equal(getMissionHistory, updateBudget) -> false
equal(getMissionHistory, getMissionHistory) -> true
equal(getMissionHistory, updateMissionHistory) -> false
```

```
equal(updateMissionHistory, createRequest) -> false
equal(updateMissionHistory, sendRequest) -> false
equal(updateMissionHistory, update) -> false
equal(updateMissionHistory, approveRequest) -> false
equal(updateMissionHistory, getBudget) -> false
equal(updateMissionHistory, updateBudget) -> false
equal(updateMissionHistory, getMissionHistory) -> false
equal(updateMissionHistory, updateMissionHistory) -> true



equal(sec,sec) -> true
equal(sec,adm) -> false
equal(sec,acc) -> false
equal(sec,itd) -> false

equal(adm,sec) -> false
equal(adm,adm) -> true
equal(adm,acc) -> false
equal(adm,itd) -> false

equal(acc,sec) -> false
equal(acc,adm) -> false
equal(acc,acc) -> true
equal(acc,itd) -> false

equal(itd,sec) -> false
equal(itd,adm) -> false
equal(itd,acc) -> false
equal(itd,itd) -> true

equal(deny,deny) -> true
equal(grant, grant) -> true
equal(deny, grant) -> false
equal(grant, deny) -> false

depends_on(createRequest) -> nil
depends_on(sendRequest) -> cons(approveRequest,nil)
depends_on(update) -> cons(updateBudget, cons(updateMissionHistory,
     nil))
depends_on(approveRequest) -> cons(getBudget, cons(
    getMissionHistory, nil))
depends_on(getBudget) -> nil
depends_on(updateBudget) -> nil
depends_on(getMissionHistory) -> nil
depends_on(updateMissionHistory) -> nil


dp_action(sendRequest, approveRequest) -> approve
dp_action(update, updateBudget) -> update
dp_action(update, updateMissionHistory) -> update
dp_action(approveRequest, getBudget) -> read
dp_action(approveRequest, getMissionHistory) -> consult

belong_to(createRequest) -> sec
belong_to(sendRequest) -> sec
belong_to(update) -> sec
belong_to(approveRequest) -> adm
belong_to(getBudget) -> acc
belong_to(updateBudget) -> acc
belong_to(getMissionHistory) -> itd
belong_to(updateMissionHistory) -> itd
```

```
delegate(adm, adm_director, sec) -> sec_administrativeSecretary
delegate(adm, adm_director, acc) -> acc_unknown
delegate(adm, adm_director, itd) -> itd_unknown

delegate(adm, adm_projectManager, sec) -> sec_unknow
delegate(adm, adm_projectManager, acc) -> acc_unknown
delegate(adm, adm_projectManager, itd) -> itd_unknown

delegate(adm, adm_unkonwn, sec) -> sec_unknow
delegate(adm, adm_unkonwn, acc) -> acc_unknown
delegate(adm, adm_unkonwn, itd) -> itd_unknown

delegate(acc, acc_budgetManager,adm) -> adm_director
delegate(acc, acc_budgetManager,sec) -> sec_unknown
delegate(acc, acc_budgetManager,itd) -> itd_unknown

delegate(acc, acc_accountingDirector,adm) -> adm_unknown
delegate(acc, acc_accountingDirector,sec) -> sec_unknown
delegate(acc, acc_accountingDirector,itd) -> itd_unknown

delegate(acc, acc_unkonwn,adm) -> adm_unknown
delegate(acc, acc_unkonwn,sec) -> sec_unknown
delegate(acc, acc_unkonwn,itd) -> itd_unknown

delegate(itd, itd_director, adm) -> adm_director
delegate(itd, itd_director, sec) -> sec_unknown
delegate(itd, itd_director, acc) -> acc_unknown

delegate(itd, itd_systemAdmin, adm) -> adm_unknown
delegate(itd, itd_systemAdmin, sec) -> sec_unknown
delegate(itd, itd_systemAdmin, acc) -> acc_unknown

delegate(itd, itd_unknown, adm) -> adm_unknown
delegate(itd, itd_unknown, sec) -> sec_unknown
delegate(itd, itd_unknown, acc) -> acc_unknown


organization(alice) -> sec
organization(anna) -> sec
organization(bob) -> adm
organization(billy) -> adm
organization(celine) -> acc
organization(christ) -> acc
organization(david) -> itd
organization(daniel) -> itd


empower(sec,u) -> if (task(u, administrative),
    sec_administrativeSecretary, if(task(u, office),
    sec_officeSecretary, sec_unknown))

task(alice, administrative) -> true
task(anna, administrative) -> false
task(anna, office) -> true
task(alice, office) -> false

permission(sec,sec_administrativeSecretary,perform) -> cons(
    createRequest, cons(sendRequest, nil))
permission(sec,sec_officeSecretary,perform) -> cons(update,nil)
permission(sec,sec_unknown,read) -> nil
```

```
empower(adm,u) -> if (position(u, director), adm_director, if(
    position(u, projectManager), adm_projectManager, adm_unknown))

position(bob, director) -> true
position(billy, director) -> false
position(billy, projetManager) -> true
position(bob, projetManager) -> false

permission(adm,adm_projectManager,consult) -> cons(approuveRequest,
     nil)
permission(adm,adm_director,approve) -> cons(approveRequest, nil)
permission(adm,adm_unknown,read) -> nil

empower(acc,u) -> if (role(u, accountingDirector),
    acc_accountingDirector, if(role(u, budgetManager),
    acc_budgetManager, acc_unknown))
role(celine, accountingDirector) -> true
role(chirst, accountingDirector) -> false
role(chirst, budgetDirector) -> true
role(celine, budgetDirector) -> false

permission(acc,acc_accountingDirector,update) -> cons(updateBudget,
     nil)
permission(acc,acc_budgetManager,read) -> cons(getBudget,nil)
permission(acc,acc_unknown,read) -> nil

empower(itd,u) -> if (role(u, director), itd_director, if(role(u,
    systemAdmin), itd_systemAdmin, itd_unknown))
role(david, systemAdmin) -> true
role(daniel, systemAdmin) -> false
role(daniel, director) -> true
role(david, director) -> false

permission(itd,itd_director,consult) -> cons(getMissionHistory, nil
    )
permission(itd,itd_systemAdmin,update) -> updateMissionHistory,nil)
permission(itd,itd_unknown,read) -> nil

authorize(org, c, a, r) -> if(is_member(r,permission(org, c, a)),
    grant, deny)
is_permitted(u, a, r) -> if(is_member(deny,perm(u, a,r, nil)), deny
    , grant)
perm(u, a, r, l) -> if(equal(organization(u),belong_to(r)),
    is_permitted_local(u, a, r, l),is_permitted_global(organization
    (u),empower(organization(u),u),a,r,l))
is_permitted_local(u, a, r, l) -> append(l, cons(authorize(
    organization(u), empower(organization(u),u),a,r),nil))

is_permitted_global(org, c, a, r, l) -> if_independent(is_empty(
    depends_on(r), nil), org, c, a, r, l)

if_independent(true, org, c, a, r, l) -> append(l, cons(authorize(
    belong_to(r), delegate(org, c ,belong_to(r)),a,r), nil))

if_independent(false, org, c, a, r, l) -> if_one_dep(is_empty(tl(
    depends_on(r)), nil), org, c, a, r, l)

if_one_dep(true, org, c, a, r, l) -> is_permitted_global(belong_to(
    r), delegate(org, c ,belong_to(r)),dp_action(r, head(depends_on
    (r))),head(depends_on(r)),append(l, cons(authorize(belong_to(r)
    ,delegate(org, c ,belong_to(r)),a,r),nil)))
```

```
        if_one_dep(false, org, c, a, r, l) -> append(is_permitted_global(
            belong_to(r), delegate(org, c ,belong_to(r)),dp_action(r, head(
            depends_on(r))),head(depends_on(r)),append(l, cons(authorize(
            belong_to(r),delegate(org, c ,belong_to(r)),a,r),  nil))),
            is_permitted_aux(belong_to(r),delegate(org, c ,belong_to(r)),r,
             tail(depends_on(r)),nil))

        is_permitted_aux(org,c, r, t,l) -> if_last_dep(is_empty(tl(t),nil),
            org,c, r, t,l)

        if_last_dep(true,org,c, r, t,l) ->is_permitted_global(org,c,
            dp_action(r, head(t)),head(t),l)

        if_last_dep(false,org,c, r, t,l) -> append(is_permitted_global(org,
            c,dp_action(r, head(t)),head(t),l),is_permitted_aux(org,c,r,
            tail(t),l)))

)
```

Bibliographie

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Alam, M., Zhang, X., Khan, K., and Ali, G. (2011). xdauth : A scalable and lightweight framework for cross domain access control and delegation. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 31–40, New York, NY, USA. ACM.

Alpuente, M., Feliú, M., Joubert, C., and Villanueva, A. (2010). *Defining Datalog in Rewriting Logic*. Springer Berlin Heidelberg.

Ardagna, C. A., di Vimercati, S. D. C., Neven, G., Paraboschi, S., Preiss, F. S., Samarati, P., and Verdicchio, M. (2010). Enabling privacy-preserving credential-based access control with xacml and saml. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1090–1095.

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

Baader, F. and Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press.

Baina, A. and Laarouchi, Y. (2012). Multilevel-orbac : Multi-level integrity management in organization based access control framework. In *Multimedia Computing and Systems (ICMCS), 2012 International Conference on*, pages 933–938.

Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. (2007). *Term Rewriting and Applications : 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007. Proceedings*, chapter Tom : Piggybacking Rewriting on Java, pages 36–47. Springer Berlin Heidelberg, Berlin, Heidelberg.

Barker, S. and Fernández, M. (2006). Term rewriting for access control. In Damiani, E. and Liu, P., editors, *Data and Applications Security XX*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin Heidelberg.

Benantar, M. (2006). *Mandatory-Access-Control Model*, pages 129–146. Springer US, Boston, MA.

Bertolissi, C. and Fernández, M. (2014). A metamodel of access control for distributed environments : Applications and properties. *Inf. Comput.*, 238 :187–207.

Bertolissi, C., Fernández, M., and Barker, S. (2007). Dynamic event-based access control as term rewriting. In Barker, S. and Ahn, G.-J., editors, *Data and Applications Security XXI*, volume 4602 of *Lecture Notes in Computer Science*, pages 195–210. Springer Berlin Heidelberg.

Biba, K. J. (1977). Integrity considerations for secure computer systems. Technical report, DTIC Document.

Bratko, I. (2001). *Prolog (3rd Ed.) : Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc.

Carbonnelle, P. (2014). *pyDatalog*. `https://sites.google.com/site/pydatalog/`.

Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1) :146–166.

Chadwick, D., Otenko, S., and Nguyen, T. (2009). Adding support to xacml for multi-domain user to user dynamic delegation of authority. *International Journal of Information Security*, 8(2) :137–152.

Chen, W., Swift, T., David, and Warren, S. (1994). Efficient top-down computation of queries under the well-founded semantics.

Chen, W. and Warren, D. S. (1996). Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43 :43–1.

Clark, D. D. and Wilson, D. R. (1987). A comparison of commercial and military computer security policies. In *Security and Privacy, 1987 IEEE Symposium on*, pages 184–184. IEEE.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2003). *Rewriting Techniques and Applications : 14th International Conference, RTA 2003 Valencia, Spain, June 9–11, 2003 Proceedings*, chapter The Maude 2.0 System, pages 76–87. Springer Berlin Heidelberg, Berlin, Heidelberg.

Contejean, É., Courtieu, P., Forest, J., Pons, O., and Urbain, X. (2011). Automated Certified Proofs with CiME3. In *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21–30.

Contejean, E., Paskevich, A., Urbain, X., Courtieu, P., Pons, O., and Forest, J. (2010). A3pat, an approach for certified automated termination proofs. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 63–72.

Cuppens, F. and Miège, A. (2003). *On The Move to Meaningful Internet Systems 2003 : OTM 2003 Workshops : OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES,WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*, chapter Administration Model for Or-BAC, pages 754–768. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cuppens, F. and Miege, A. (2004). AdOrBAC : an administration model for Or-BAC. *Computer Systems Science and Engineering*, 19 :151–162.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3) :374–425.

David, F. and Richard, K. (1992). Role-based access controls. *Proceedings of 15th NIST-NCSC National Computer Security Conference*, 563.

Dershowitz, N. (1982). Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3) :279 – 301.

Dougherty, D., Fisler, K., and Krishnamurthi, S. (2006). Specifying and reasoning about dynamic access-control policies. In Furbach, U. and Shankar, N., editors, *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer Berlin Heidelberg.

Fischer, J. and Majumdar, R. (2008). A theory of role composition. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 320–328.

Fournet, C., Gordon, A. D., and Maffeis, S. (2007). A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5).

Gallagher, R. P. (1987). A guide to understanding discretionary access control in trusted systems.

Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and others (2014). Proving termination of programs automatically with AProVE. In *Automated Reasoning*, pages 184–191. Springer.

Huet, G. P. (1980). Confluent reductions : Abstract properties and applications to term rewriting systems : Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4) :797–821.

Kalam, A., Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miege, A., Saurel, C., and Trouessin, G. (2003). Organization based access control. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131.

Karp, A. H. and Li, J. (2010). Solving the transitive access problem for the services oriented architecture. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pages 46–53.

Klop, J. W., van Oostrom, V., and van Raamsdonk, F. (1993). Combinatory reduction systems : Introduction and survey. *Theor. Comput. Sci.*, 121(1&2) :279–308.

LaPadula, L., Bell, D. E., and LaPadula, L. J. (1973). Secure computer systems : Mathematical foundations. Technical report.

Lee, A. J., Seamons, K. E., Winslett, M., and Yu, T. (2007). *Automated Trust Negotiation in Open Systems*, pages 217–258. Springer US, Boston, MA.

Lin, A. (1999). Integrating policy-driven role based access control with the common data security architecture.

Lischka, M., Endo, Y., and Sánchez Cuenca, M. (2009). Deductive policies with xacml. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09, pages 37–44, New York, NY, USA. ACM.

Long, Y.-H., Zhi-Hong, T., and Liu, X. (2010). Attribute mapping for cross-domain access control. In *Computer and Information Application (ICCIA), 2010 International Conference on*, pages 343–347.

Manna, Z. and Ness, S. (1970). On the termination of Markov algorithms. In *Procceedings of the 3rd Hawaii International Conference on System Science*, pages 789–792.

Mecella, M., Ouzzani, M., Paci, F., and Bertino, E. (2006). Access control enforcement for conversation-based web services. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 257–266, New York, NY, USA. ACM.

Newman, M. (1942). On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43(2) :223–243.

Oliveira, A. S. D. (2007). Rewriting-based access control policies. *Electronic Notes in Theoretical Computer Science*, 171(4) :59 – 72. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006).

Oliveira, A. S. D. (2008). Réécriture et modularité pour les politiques de sécurité.

Parducci, B. (2013). extensible access control markup language (xacml) version 3.0.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition.

Samarati, P. and de Vimercati, S. (2001). Access control : Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin Heidelberg.

Sedgewick, R. and Wayne, K. (2011). *Algorithms, 4th Edition*. Addison-Wesley.

Totel, E., Blanquart, J.-P., Deswarte, Y., and Powell, D. (1998). Supporting multiple levels of criticality. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 70–79.

Uttha, W. (2012). Spécification et vérification des politiques de sécurité basées sur la réécritur. Master's thesis, Université d'Aix-Marseille. `http://pageperso.lif.univ-mrs.fr/~worachet.uttha/dl/memoire_UTTHA.pdf`.

Uttha, W., Bertolissi, C., and Ranise, S. (2014a). Towards a reference architecture for access control in distributed web applications. In *ESSOS-2014*

- Proceedings of the 2014 ESSoS Doctoral Symposium co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2014), Munich, Germany, February 26, 2014., volume 1298 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Uttha, W., Bertolissi, C., and Ranise, S. (2014b). Towards a reference architecture for access control in distributed web applications. Poster presented at International Symposium on Engineering Secure Software and Systems (ESSoS 2014), Munich, Germany, February 26, 2014.

Uttha, W., Bertolissi, C., and Ranise, S. (2015a). Modeling authorization policies for web services in presence of transitive dependencies. In *SECRYPT 2015 - Proceedings of the 12th International Conference on Security and Cryptography, Colmar, Alsace, France, 20-22 July, 2015.*, pages 293–300. SciTePress.

Uttha, W., Bertolissi, C., and Ranise, S. (2015b). A transitive access solution for web services. Demonstration presented on Industry and Demo session at International Symposium on Engineering Secure Software and Systems (ESSoS 2015), Milan, Italy, March 4-6, 2015. `https://distrinet.cs.kuleuven.be/events/essos/2015/programme-accepteddemos.html`.

Uttha, W., Bertolissi, C., and Ranise, S. (2016). Access control model for web services in presence of transitive dependencies. Poster presented at Thai Student Academic Conference (TSAC2016), Alghero, Italy, May 11-14, 2016.

V. C. Hu, D. Ferraiolo, R. K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication 800-162.

Vardi, M. Y. (1982). The complexity of relational query languages (extended abstract. In *In Proceedings of the fourteenth annual ACM symposium on Theory of computing (STOC '82*, pages 137–146.

Wahsheh, L. A. and Alves-Foss, J. (2008). Security policy development : Towards a life-cycle and logic-based verification model. 5(9) :1117–1126.

Y. Deswarte, A. A. E. K. (2009). *Poly-OrBAC : An access control model fior inter-organizational web services*. IGI-Global.