Rewriting and modularity for security policies

Anderson Santana de Oliveira

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Réécriture et Modularité pour les Politiques de Sécurité

THÈSE

présentée et soutenue publiquement le 31 Mars 2008

pour l'obtention du

Doctorat de l'Université Henri Poincaré

(spécialité informatique)

par

Anderson Santana de Oliveira

Composition du jury

Rapporteurs :	Frédéric Cuppens	Professeur, ENST-Bretagne, Rennes, France
	Maribel Fernandez	Professeur, King's College London, Londres, Royaume-Uni
Examinateurs :	Piero A. Bonatti	Professeur, Università di Napoli Federico II, Naples, Italie
	Isabelle Chrisment	Professeur, Université Henri Poincaré , Nancy, France
	Daniel J. Dougherty	Professeur, Worcester Polytechnic Institute, Worcester, MA, USA
	Thérèse Hardin	Professeur, Université Paris VI, Paris, France
	Claude Kirchner	Directeur de recherche, INRIA, Bordeaux, France
	Hélène Kirchner	Directeur de recherche, INRIA, Bordeaux, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Remerciements

Je remercie Claude et Hélène Kirchner qui m'ont fait l'honneur de diriger ma thèse avec une grande rigueur scientifique et qui m'ont soutenu avec enthousiasme tout au long de ce travail. Je tiens à leur exprimer ma profonde gratitude.

Je remercie vivement les membres du jury : Frédéric Cuppens et Maribel Fernandez qui ont eu la gentillesse de rapporter sur cette thèse ; Thérèse Hardin la présidente du jury ; Piero Bonatti qui m'a fait des commentaires très constructifs dès notre premier contact lors de ma présentation à la réunion annuelle du réseau d'excellence Rewerse en 2006 ; Isabelle Chrisment qui, avec ses questions pertinentes, m'a aidé à justifier ma démarche ; et Dan Dougherty qui m'a chaleureusement accueilli au sein du département d'informatique du Worcester Polytechnic Institute et avec qui j'ai eu le plaisir de travailler.

Merci également aux participants du Projet ANR SSURF avec qui j'ai eu des discussions fructueuses, en particulier Charles Morisset pour notre collaboration.

Cette thèse n'aurait jamais été ce qu'elle est aujourd'hui sans mes échanges scientifiques importants avec les membres de l'équipe PROTHEO et PAREO. Je tiens à remercier Horatiu Cirstea et Pierre-Etienne Moreau pour l'intérêt qu'ils ont manifesté à l'égard de cette thèse et pour leurs remarques pertinentes. Mes remerciements vont également à Yohan Boichut qui s'est penché à mes côtés sur le problème de l'indécidabilité de la propriété de sûreté dans le modèle HRU.

Je remercie tous les stagiaires avec qui j'ai travaillé : tout particulièrement, Eric Ke Wang, ainsi que Selina Zhen Wang et Christophe Masson qui ont chacun apporté leur pierre à l'édifice.

Merci à tous les thésards de l'équipe PROTHEO, particulièrement Oana Andrei pour son amitié, et mes anciens collègues de bureau Colin Riba, Laura Lowenthal et Tony Bourdier pour leur gentillesse.

Je remercie la CAPES et l'INRIA pour leur confiance et la bourse qu'ils m'ont accordées. J'adresse également mes remerciements au personnel de l'Université Henri Poincaré et du LORIA qui ont contribué efficacement au bon déroulement de ma thèse, principalement Chantal Llorens pour son incroyable patience et sa bonne humeur.

Enfin je remercie mes parents, mon frère, mes soeurs, ainsi qu'Annette et Alphonse, pour leur soutien inconditionnel. Je garde le meilleur pour la fin en adressant une dédicace spéciale à Marie-Ange qui a su faire preuve d'amour et de patience.

Para Lourdes, Gilberto, Andreza, Elton, Hemiliane, Amanda e Marie-Ange.

Table des matières

Chapitre 1

Résumé Étendu

1.1 Introduction

Ce chapitre présente un résumé en langue française des principales contributions de cette thèse. Sa lecture pourra être effectuée de façon indépendante du restant du document en anglais, à l'exception des définitions formelles sur la réécriture avec des stratégies, données dans le Chapitre 3. Ce chapitre apporte une compréhension de l'ensemble des enjeux autour des politiques de sécurité, de leur composition et de la manière dont ces problèmes sont abordés dans nos travaux.

Ce chapitre expose d'abord les motivations des travaux de recherche réalisés pendant cette thèse dans la Section 1.1.1. Ensuite, nous présentons l'historique de la recherche dans le domaine de la spécification formelle de politiques de sécurité dans la Section 1.1.2. La Section 1.2 présente les principaux résultats obtenus dans cette thèse. La section 1.3 présente des conclusions et montre quelques directions futures de recherche. La section 1.4 présente un panorama sur le restant de ce manuscrit de thèse.

1.1.1 Motivations

À l'âge de l'information, l'économie, les communications et les divertissements dépendent des ordinateurs. Des informations d'une valeur inestimable pour beaucoup d'agents (compagnies, gouvernements, etc.) circulent dans des systèmes d'information de grande taille, souvent reliés en réseau à d'autres systèmes. Dans ce contexte, un problème primordial traité par les politiques de sécurité est la protection des données gérées par ces systèmes contre des utilisations indésirables. Selon Bishop [Bishop, 2004], une politique de sécurité est une déclaration de ce qui est permis et de ce qui ne l'est pas. Ces politiques sont donc des spécifications extrêmement importantes, dans la mesure où la sécurité d'une quantité énorme d'informations manipulées par des ordinateurs (ou de façon plus abstraite, des ressources) dépendent de la formulation claire des exigences de sécurité.

Les exigences de sécurité déterminent les situations dans lesquelles un système peut être exposé à des attaques : (i) des informations secrètes peuvent être révélées à des agents qui ne sont pas censés l'obtenir, (ii) des informations peuvent être corrompues ou modifiées de façon telle qu'on ne peut plus avoir confiance en son contenu, ou encore détruites et (iii) des informations peuvent être retenues et devenir indisponibles pendant un intervalle de temps crucial. Ces trois points sont appelées respectivement de confidentialité, intégrité et disponibilité. Une politique de sécurité se concentre habituellement sur un de ces aspects, mais son but dans une configuration

réelle peut être une combinaison arbitraire de conditions impliquant ces trois conditions.

Le premier problème lié à la spécification des politiques de sécurité auquel nous nous intéressons est de faire face à des exigences de sécurité élaborées des systèmes applicatifs courants. Les politiques sont décrites comme des calculs aboutissant à une décision d'accès. Plus largement, dans un système donné, une politique évalue les données dans son état actuel de façon à indiquer quelles sont les transitions possibles, à partir de cet état, qui mènent à d'autres états qui peuvent être considérés comme sûrs. Par conséquent, les définitions de politiques de sécurité peuvent être arbitrairement complexes et exigent une représentation appropriée.

Une exigence essentielle qu'un langage de spécification de politiques doit satisfaire est d'être compris par des personnes et convenir également à l'analyse automatisée par des programmes. Ce type d'analyse est fortement souhaitable car les politiques de sécurité peuvent être sujettes à des changements fréquents et reflètent des contraintes réelles sur la façon dont on peut accéder à des informations. Il est donc très utile d'avoir des outils pour vérifier la concordance d'une politique avec les conditions de sécurité requises et pour tester des propriétés de sécurité.

Une propriété importante que les politiques devraient satisfaire est d'être non ambiguës. Cependant, la distinction claire entre les états permis et interdits pour un système est difficile à réaliser. En raison de la quantité et de la diversité des données qu'une politique doit évaluer, les conditions de sécurité sont souvent partiellement spécifiées. En effet, les politiques ciblent l'intérêt d'une partie des agents qui utilisent le système, pouvant être en contradiction avec les intérêts d'autres agents. Par exemple, si on considère un système médical où une politique régule l'accès aux données des patients, les fournisseurs de services de santé et les compagnies d'assurance pourraient avoir des demandes de contrôle d'accès incompatibles les uns avec les autres.

Nous étudions également des propriétés sur le flux d'information, c'est-à-dire des exigences de sécurité déterminant que certains agents ne peuvent pas obtenir des informations ou des droits d'accès provenant d'une classe de sécurité supérieure à la leur. En plus de la difficulté de spécifier de ce genre de propriété, les procédures de vérification peuvent être aussi complexes à réaliser.

En outre, les politiques ne sont pas écrites et maintenues à partir d'une seule spécification. Elles sont indépendamment créées par les différentes entités qui proposent leurs propres priorités en définissant les buts de sécurité. De grands systèmes d'information sont utilisés par plusieurs entités distinctes. Ainsi, une politique de sécurité globale doit accommoder les différentes exigences de chaque entité selon des conditions de sécurité, qui à leur tour peuvent être énoncées de diverses manières.

Puisque des systèmes d'information doivent être intégrés les uns avec les autres, la composition de politiques est un problème majeur dans le domaine de la sécurité informatique contemporaine. Les spécifications de ces politiques doivent appréhender les différentes manières de définir une politique composée afin de régir aux différentes parties d'un système. La résolution des problèmes autour de la composition des politiques est ainsi un des points centraux de cette thèse.

La taille et la complexité des politiques de sécurité que nous considérons dans cette thèse indiquent que les politiques elles-mêmes sont des composants logiciels intéressants méritant d'être étudiés. Idéalement, les politiques sont développées, analysées et maintenues séparément des systèmes d'information où elles sont imposées, avec le soutien d'outils spécialisés. Malheureusement, une grande partie des logiciels construits aujourd'hui définissent toujours des

politiques de façon ad-hoc, c'est-à-dire codées à la main directement dans les programmes. Cela rend difficile la compréhension des calculs des autorisations d'accès, pouvant causer des défauts compromettant la sécurité logicielle. Ceci nous a poussé à étudier des méthodologies pour l'imposition systématiques des politiques qui permettent de garder la structure modulaire autant de la politique que du programme cible.

Dans la section qui suit, nous présenterons plusieurs approches qui ont été proposées pour la spécification des politiques de sécurité et pour la construction des mécanismes d'imposition de politiques dans des systèmes informatiques.

1.1.2 Historique

Les politiques de contrôle d'accès sont une manière particulière de mettre en oeuvre une politique de sécurité selon un certain modèle de sécurité. Un *modèle de sécurité* est une spécification des exigences relatives à la confidentialité, l'intégrité ou la disponibilité nécessaires à l'utilisation fiable d'un système donné. Le modèle ne donne aucune indication précise des mécanismes pour leur réalisation. L'appellation "modèle" a été adoptée assez tôt dans la littérature sur la sécurité informatique et a été conservé depuis. Un modèle établit essentiellement des restrictions sur les entrées et les sorties d'un système, qui devront être suffisantes pour assurer un certain objectif de sécurité. Dans cette thèse, nous allons nous concentrer principalement sur la confidentialité.

Les politiques de contrôle d'accès impliquent trois éléments principaux : des ressources à protéger, des actions (ou modes d'accès, ou encore privilèges d'accès) pouvant être exécutées sur ces ressources et des sujets (utilisateurs ou programmes) du système pouvant demander la permission d'exécuter une certaine action sur les ressources existantes.

[Lampson, 1974] a organisé ces éléments dans une matrice de contrôle d'accès, dont les cellules contiennent un ensemble de privilèges. Ces privilèges sont placés à l'intersection d'une ligne, où se situe l'utilisateur et d'une colonne, où se trouve la ressource. Par conséquent, étant donné un ensemble fini de sujets S, un ensemble fini d'actions A et un ensemble fini d'objets O, on peut représenter une entrée dans la matrice de contrôle d'accès (notée M), par

$$M[s, o] = \{a_1, \ldots, a_n\}, \text{où } s \in S, o \in O, a_1, \ldots, a_n \in A$$

L'état global d'un système dans ce modèle est un triplet (S, O, M), où les transitions se produisent par l'intermédiaire des demandes pour modifier l'ensemble courant de privilèges dans la matrice. Ce modèle est encore d'usage aujourd'hui. Mais il a été amélioré pour faciliter la médiation entre de grandes bases d'utilisateurs et de ressources par l'intermédiaire des listes de contrôle d'accès, ou par l'attribution de privilèges à des rôles. Ces rôles peuvent être assumés par un utilisateur pour l'exécution de certaines tâches dans un système.

Les travaux de Lampson ont été affinés par Harrison, Ruzzo et Ullman [Harrison et al., 1976], qui ont proposé le modèle abstrait connu aujourd'hui comme le modèle HRU. Leur objectif était d'analyser la question de la complexité dans le domaine du contrôle d'accès. Le modèle HRU définit des *systèmes de protection* qui consistent en une matrice de contrôle d'accès et une série de petits "programmes" (*commandes*) décrivant la manière de modifier la matrice à l'aide de quelques instructions simples en suivant la syntaxe ci-dessous :

$$
\begin{array}{ll}
\textbf{Command} & name\ (X_1, \ldots, X_k) \\
\textbf{if} & a_1 \in M[X_{s1}, X_{o1}]\ and \\
& a_2 \in M[X_{s2}, X_{o2}]\ and \\
& \ldots \\
& a_m \in M[X_{si}, X_{oj}] \\
\textbf{then} & \\
& op_1 \\
& \ldots \\
& op_n
\end{array}
$$

Les noms de paramètres $X_1, \ldots, X_k$ ne peuvent être qu'instanciées par des identificateurs de sujet ou d'objet. Chaque a_i est un droit d'accès et chaque op_k est une des opérations suivantes, dont la sémantique correspond exactement à leur description :

- **enter** $a \in A$ **into** $M[s, o]$,
- **delete** $a \in A$ **from** $M[s, o]$,
- **create subject** $s \in S$,
- **create object** $o \in O$,
- **destroy subject** $s \in S$,
- **destroy object** $o \in O$.

Par conséquent, il est possible d'écrire de petits scripts qui définissent comment manipuler les permissions dans un système.

Example 1. *Le fait qu'un processus en cours d'exécution dans un système d'exploitation est propriétaire des fichiers qu'il crée est illustré par le programme suivant :*

$$
\begin{array}{ll}
\textbf{Command} & CREATE\ (process, file) \\
& \textbf{create object}\ file \\
& \textbf{enter}\ own\ \textbf{into}\ [process, file]
\end{array}
$$

Où $process$ et $file$ sont des paramètres. Un sujet peut conférer le droit de lecture à des amis par le script conditionnel suivant :

$$
\begin{array}{ll}
\textbf{Command}\ CONFER_{read}\ (owner, friend, file) & \\
\textbf{if} & own \in M[owner, file] \\
\textbf{then} & \textbf{enter}\ read\ \textbf{into}\ [friend, file]
\end{array}
$$

avec les paramètres $owner$, $file$ et $friend$.

La configuration d'un système de protection est une matrice d'accès avec des ensembles de sujets, d'objets et de droits. Une nouvelle configuration est obtenue par l'exécution d'une séquence de commandes. Le modèle HRU présente une propriété importante sur laquelle repose les résultats de complexité que Harrisson, Ruzzo et Ullman ont pu obtenir. Cette propriété est défini ci-dessous :

Definition 1. *Pour un système de protection particulier et un privilège d'accès générique a, la configuration initiale S_0 est* non-sûre *pour $a \in A$, si une suite de commandes insère le droit a dans une cellule ne contenant pas a en S_0.*

De même, un état est sûr pour le droit r s'il n'est pas non-sûr pour r. Néanmoins selon [Harrison et al., 1976], ce problème de sécurité est décidable seulement pour des systèmes *mono-opérationnels*, c'est-à-dire où les commandes contiennent une seule opération primitive et le problème général reste indécidable.

Cette propriété ne paraît pas forcément d'un grand intérêt dans la pratique : par exemple, en partageant un fichier avec un groupe restreint de sujets, il est nécessaire d'ajouter des droits d'accès pour ces sujets à ce fichier, ce qui viole les conditions de sécurité selon le modèle HRU. Cependant, cette propriété de sécurité garantit qu'un deuxième ensemble (non fiable) de sujets n'obtiendra pas de tels privilèges par d'autres moyens. Le travail pionnier dans le modèle HRU montre la difficulté du problème, même si nous avons une compréhension complète des opérations qui propagent les droits dans un système.

Nous appelons Contrôle d'Accès Discrétionnaire (ou *Discretionary Access Control*, DAC), les modèles où les sujets peuvent passer librement les droits qu'ils possèdent aux autres utilisateurs. En général, le DAC a la faiblesse de ne pas offrir un dispositif de gestion prévisionnelle en cas d'attaques par des *Chevaux de Troie*. Lorsque les sujets exécutent des programmes tiers, ils leur livrent tous les privilèges en leur nom. Par conséquent, il n'y a aucune garantie que de tels programmes n'exécutent pas d'action malveillante sur les fichiers de l'utilisateur, ce qui compromet sérieusement la confidentialité.

Le Contrôle d'Accès Obligatoire (ou *Mandatory Access Control*, MAC) est une alternative pour réduire les dégâts causés par ces attaques, par l'étiquetage des sujets et des objets avec un niveau de sécurité. Avec le MAC on considère qu'il existe un ordre entre les niveaux de sécurité et l'observation de l'ordre permet d'éviter la transmission des droits d'accès à l'information à partir d'un objet de niveau élevé de sécurité, vers un sujet avec un niveau de sécurité moins élevé.

Depuis longtemps, les chercheurs en sécurité informatique ont identifié le besoin de rigueur mathématique dans les spécifications de politiques de sécurité. Les premiers modèles de sécurité reflètent le coût encore très important des machines vers la fin des années 70 et leur partage entre plusieurs utilisateurs.

[Bell and LaPadula, 1974] ont présenté un modèle formel (connu sous le nom de modèle BLP), pour capturer les besoins des systèmes militaires automatisés. Ce modèle s'est montré efficace pour assurer la sécurité des archives papier. Dans leur travail, les auteurs ont associé des étiquettes avec des niveaux de sécurité aux utilisateurs et aux objets dans le système. Ces étiquettes sont ordonnées selon une classification, où les utilisateurs avec un niveau inférieur ne devraient pas avoir accès à l'information appartenant à une classe de sécurité plus élevée. De telles conditions indiquent la manière dont l'information circule dans le système et se nomment souvent *politiques de flux d'information*. Un autre travail précurseur est apparu avec [Biba, 1977] pour le traitement de contraintes d'intégrité. Il ressemble BLP, dans le sens où les niveaux d'intégrité remplacent les niveaux de sécurité, et le modèle résultant définit l'impossibilité pour l'information de passer d'un sujet à un objet dont le niveau d'intégrité est plus élevé.

Le modèle BLP est principalement utilisé pour mettre en oeuvre les politiques militaires. Les

niveaux de sécurité sont structurés dans un treillis [McLean, 1988, Sandhu, 1993] L, contenant les éléments $top\ secret \succeq secret \succeq confidential \succeq unclassified$. Parfois, les niveaux sont combinés avec une étiquette, pour créer différents compartiments $[secret, NATO]$, $[secret, Navy]$ incomparables les uns avec les autres.

En plus des niveaux de sécurité, Bell et LaPadula réutilisent la notion de matrice d'accès pour construire une série de définitions qui aboutissent à une caractérisation des systèmes sécurisés. Soit $f : S \cup O \rightarrow L$ la fonction qui appliquée à un sujet ou à un objet, renvoie le niveau de sécurité correspondant. Soient E un ensemble d'états, où chaque état est une paire (f, M), M est une matrice de contrôle d'accès et $T : E \times Q \rightarrow E$ est une fonction de transition, où E est un état et Q une requête pour lire ou écrire. Cette fonction renvoie un nouvel état. Le modèle BLP définit alors les conditions suivantes :

- Un état (f, M) est sécurisé en lecture (appelée *Propriété de sécurité simple*) si et seulement si $\forall s \in S, o \in O, read \in M[s, o] \implies f(s) \geq f(o)$.
- Un état (f, M) est sécurisé en écriture (appelée *propriété étoile*) si et seulement si $\forall s \in S, o \in O, write \in M[s, o] \implies f(o) \geq f(s)$.
- Un état est sûr si et seulement s'il est sécurisé en lecture et en écriture,
- Enfin, un système (S_0, Q, T) est sûr si et seulement si S_0 est sûr et si chaque état atteignable à partir de S_0 par une suite finie de requêtes d'accès de Q est sûr.

À partir de ces définitions, Bell et LaPadula ont pu prouver un théorème important de leur modèle, qui peut être officieusement énoncé comme "pas de lecture vers le haut et pas d'écriture écrit-vers le bas". En d'autres termes, un sujet dont le niveau de sécurité est $secret$ peut lire des dossiers classés $confidential$, mais ne peut pas y ajouter de nouvelle information. D'autre part, le même sujet peut joindre des données aux dossiers $top\ secret$, mais ne peut pas obtenir l'information des objets de niveau plus élevé.

En suivant les mêmes principes que Bell et LaPadula, d'autres auteurs ont proposé des modèles pour la confidentialité, comme le modèle de la *muraille de Chine* (CW) [Brewer and Nash, 1989]. Son but principal est de contrôler l'accès aux données des compagnies concurrentes dans le domaine des services aux entreprises. Par exemple, pour une compagnie donnée proposant de l'expertise financière à d'autres compagnies, un de ses conseillers ne devra pas travailler pour différents clients de cette entreprise qui ont des conflits d'intérêt.

Le modèle CW introduit des concepts différents de BLP. Chaque objet peut être vu comme un fichier et ne concerne qu'une seule entreprise à la fois. Un groupe d'une société (cg) est une collection de fichiers d'une entreprise en particulier. Une classe de conflit (cc) assemble des entreprises concurrentes. Le principe de la politique de sécurité est le suivant : un sujet peut accéder à des informations provenant d'une nouvelle société tant qu'elle n'a jamais lu des informations provenant d'une autre société appartenant à la même classe de conflit.

Ensuite, les conditions de sécurité sont établies comme suit. Soit $prec(s)$ l'ensemble d'objets qu'un sujet s a lu. s peut alors lire un objet o si l'une des situations suivantes est vérifiée :

- $\exists o' \in O$, tel que $o' \in prec(s)$ et $cg(o) = cg(o')$ (les deux appartiennent au même groupe de société),
- $\forall o' \in prec(S) \implies cc(o') \neq cc(o)$.

De la même façon, un système est sécurisé si toutes les transitions préservent la condition de

sécurité, à partir d'un état initial aseptisé, où l'information peut être consultée par tout sujet, car elle ne concerne aucune société en particulier.

[Goguen and Meseguer, 1982] ont introduit la notion de politique de non-interférence, où les actions exécutées par tel utilisateur d'un système ne doivent pas être inférées par un autre utilisateur, capable d'observer le comportement externe du système. Les auteurs ont élaboré une formalisation abstraite de la non-interférence en termes des commandes possibles qui peuvent être appelées par un groupe d'utilisateurs pendant l'exécution d'un système. Ces commandes déterminent la façon dont l'information entre dans le système et la façon dont elle peut être perçue par un utilisateur externe.

Ces travaux sont essentiels dans le domaine de la sécurité informatique. Ils apportent des principes pour la spécification formelle des politiques de sécurité : les propriétés de sécurité doivent être correctement énoncées et avérées. Cependant, les premiers modèles de sécurité sont trop rigides et donc difficilement réutilisables. L'adaptation de chacun des modèles exigerait une nouvelle validation.

Malheureusement, ces modèles généraux ne capturent pas certaines conditions pouvant compromettre leurs propriétés de sécurité dans des configurations plus concrètes. Par exemple, la possibilité de descendre les niveaux de sécurité des objets est omise de la formulation originale de BLP. Ainsi, comme il est mentionné dans [McLean, 1985], un système peu sûr peut satisfaire les propriétés de sécurité du modèle de BLP, étant donné que les niveaux de sécurité des objets ou des sujets peuvent changer tout au long de l'exécution du système. BLP se fonde également sur la notion intuitive que les actions de *lire* et *écrire* impliquent que l'information va dans une direction seulement, des objets vers les sujets et inversement. Or, ce n'est pas le cas dans les systèmes informatiques réels.

La recherche sur les politiques de sécurité s'est concentrée sur des formalismes de spécifications de politiques (que nous appellerons cadres formels) permettant de définir des politiques avec une sémantique formelle, selon les contraintes de chaque domaine d'application. De tels cadres formels, en principe, rendent possible le raisonnement automatique sur les propriétés de sécurité d'un point de vue abstrait. Les propriétés de politiques n'ont pas besoin d'être énoncées et prouvées à nouveau pour chaque système où elles doivent être appliquées. Ceci a permis l'émergence d'approches basées sur des règles pour les politiques de sécurité, ce que nous développons ci-après.

Approches basées sur des règles

Les politiques de sécurité s'expriment sous la forme de règles en langage naturel : si certaines conditions sont satisfaites, une certaine demande d'accès doit être autorisée ou non. Les langages basés sur des règles sont bien adaptés pour la spécification de politiques de sécurité car ils ont une syntaxe compréhensible pour les utilisateurs et sont basés sur une sémantique formelle. La spécification des politiques basées sur des règles est donc un domaine important de recherche en sécurité informatique.

Un grand nombre d'initiatives ont appliqué aux politiques de sécurité différents formalismes basés sur des règles. Les approches existantes adoptent différents cadres sémantiques tels que la logique du premier ordre [Halpern and Weissman, 2003, Jajodia et al., 2001, Becker and Sewell, 2004b], Datalog [Dougherty et al., 2006, Li and Mitchell, 2003],

la logique temporelle [Bertino et al., 1998, Cuppens et al., 2005], la logique déontique [Cholvy and Cuppens, 1997, Kalam et al., 2003], ou la logique annulable (defeasible logic) [Lee et al., 2006].

La caractéristique commune de ces initiatives est de fournir à leurs utilisateurs différents degrés d'expressivité pour décrire une politique. Cette expressivité permet d'indiquer de façon précise les conditions selon lesquelles l'accès doit avoir lieu. Dans le contrôle d'accès, ces conditions impliquent des attributs reliés aux sujets et/ou aux ressources protégés dans le système. Par exemple, dans un hôpital, où la politique de sécurité doit tenir en compte des attributs des sujets, l'accès aux dossiers médicaux des patients peut être limité aux médecins et au personnel soignant.

Néanmoins, d'autres approches offrent des alternatives pour la spécification des politiques de sécurité. Elles s'appuient sur des langages déclaratifs non basés sur des règles. Dans ces travaux les politiques sont des petits programmes qui calculent des autorisations, et qui sont munis de quelques constructions pour faciliter leur réutilisation. Des exemples représentatifs de cette direction de recherche sont le langage Ponder [Damianou et al., 2001], le système Polymer [Bauer et al., 2005] et l'API de Sécurité de la plate-forme Java [Gong et al., 2003].

À l'origine, dans ces formalismes pour la spécification de politiques de sécurité il n'était possible que de spécifier des politiques fermées, c'est-à-dire, des politiques déterminant les situations strictement permises. Ou des politiques ouvertes, c'est-à-dire indiquant seulement les cas interdits et autorisant toute autre situation non spécifiée. Les politiques fermées ou ouvertes devenant rapidement restrictives lors d'exceptions ponctuelles aux règles d'accès (ce qui nécessite l'ajout d'une règle à chaque accès), des règles positives (qui autorisent un accès) et des règles négatives (qui interdisent une demande d'accès) doivent donc être indiquées explicitement dans une même spécification de politique.

Des politiques fermées ou ouvertes contiennent des règles par défaut qui permettent (ou refusent) une grande classe de requêtes d'accès. Ces requêtes ne seraient pas directement concernées par la politique donnée. Ceci apporte une difficulté supplémentaire pour composer de telles politiques de sécurité : lorsque les politiques composantes se superposent, il peut être compliqué d'assumer une décision d'accès par défaut pour une certaine requête. Une manière de fournir davantage de flexibilité à la composition est l'attribution d'une décision neutre aux requêtes d'accès situées en dehors de la portée d'une certaine politique. Une telle politique peut être composée de façon cohérente avec une autre politique qui fournit une décision d'accès différente pour ces requêtes.

Le besoin de multiples décisions d'accès (correspondant aux règles positives, négatives et neutres) est important, notamment pour le problème de la composition des politiques de sécurité. Un tel dispositif apparaît dans la conception de plusieurs cadres récents de spécification de politiques [Moses, 2005, Bauer et al., 2005, Bruns et al., 2007].

Nous allons détailler ces cadres récents dans le Chapitre 3. Nous présentons une comparaison plus précise des approches basées sur des règles dans la section 4.6.

1.2 Contributions

Dans cette thèse, nous allons aborder plusieurs questions liées à la spécification, vérification, composition et implantation des politiques de sécurité expressives basées sur des règles.

1.2.1 Spécification de Politiques de sécurité

La contribution principale de la thèse est la définition générale des politiques de sécurité qui permet une large gamme d'applications. Une des nouveautés de notre approche est d'adopter la réécriture de termes comme formalisme pour la spécification des politiques de sécurité. Les premières approches sur le contrôle d'accès par l'utilisation de la réécriture de termes ont été présentées dans [de Oliveira, 2007], indépendamment de [Barker and Fernández, 2006]. La formalisation adoptée dans cette thèse est apparue dans [Dougherty et al., 2007b], mise à jour et augmentée par rapport à la version du travail dans [de Oliveira, 2007]. Elle inclut la notion de stratégies de réécriture [Kirchner et al., 2008, Kirchner, 2005, Borovansky et al., 2001, Cirstea et al., 2003] dans la définition de politique. Dans les chapitres suivants, nous allons présenter des exemples de politiques de contrôle d'accès pour des systèmes d'information en général, politiques pour des pare-feux dans des réseaux et des politiques de sécurité à multi-niveaux.

La réécriture de termes est un champ de recherche très actif dans le domaine du raisonnement automatisé. C'est un modèle informatique Turing complet, appliqué principalement dans des prouveurs de théorèmes et des systèmes de résolution de contraintes et servant de base théorique aux langages de programmation fonctionnelle. La réécriture de termes comprend une logique associée, la logique de réécriture, présentée dans [Martí-Oliet and Meseguer, 2002] et un calcul, le ρ-calcul [Cirstea and Kirchner, 2001]. Tous les deux expliquent les principaux mécanismes de la réécriture : ses propriétés réflexives, le filtrage et les stratégies de réécriture. L'utilisation de la réécriture de termes pour des politiques de sécurité est justifiée par son expressivité, par de nombreuses techniques de preuve disponibles pour la vérification des propriétés et par l'efficacité des diverses implantations existantes des moteurs de réécriture. Dans cette thèse, nous utilisons tous ces avantages pour fournir un cadre formel aux politiques de sécurité.

Les spécifications de politique sont une activité laborieuse : elles doivent préciser des conditions de sécurité dans les systèmes automatisés et exigent une compréhension de tous les éléments influençant la production des décisions de contrôle d'accès. Il est essentiel que les cadres formels des politiques fournissent l'expressivité (pour capturer des conditions complexes) et des techniques fiables (pour vérifier des propriétés des politiques).

Nous avançons l'argument selon lequel la réécriture de termes est un paradigme approprié pour exprimer et raisonner au sujet des politiques de sécurité, en particulier, au sujet du contrôle d'accès. L'avantage principal de la réécriture est de fournir une sémantique formelle dans la modélisation de politiques flexibles. Une telle sémantique est fondée sur une théorie logique et de preuve, où des propriétés importantes de la politique peuvent être énoncées et prouvées. Les spécifications en réécriture étant exécutables, les politiques peuvent ainsi être facilement prototypées dans un des multiples systèmes disponibles actuellement, tels que ELAN [Borovanský et al., 1998], Maude [Clavel et al., 2007, Clavel et al., 2002], Tom [Balland et al., 2007].

Les développements récents dans le domaine du contrôle d'accès visent à exprimer diverses

contraintes sur l'environnement où les politiques sont imposées, afin de capturer de vraies conditions auxquelles font face les auteurs de logiciels. Dans de tels scénarios, la définition de politique doit permettre de spécifier des calculs sur des demandes d'accès menant à des autorisations.

Nous présentons une formalisation des politiques de contrôle d'accès basées sur la réécriture avec des stratégies. Les politiques sont définies par un ensemble de règles de réécriture régi par une stratégie. Les demandes d'accès et l'environnement où des politiques sont imposées sont représentés sous la forme de termes algébriques. Une fois appliquées à une demande d'accès, les règles de réécriture induisent un processus d'évaluation renvoyant une décision d'autorisation. Cette formalisation nous permet ainsi de capturer beaucoup d'aspects dynamiques importants dans l'application de politique, comme le montre les exemples dans cette thèse.

Definition 2 (Politique de Sécurité)**.** *Une politique de sécurité, $\wp$, est un 5-tuplet $(\mathcal{F}, D, R, Q, \zeta)$ où :*

1. *$\mathcal{F}$ est une signature ;*
2. *D est un ensemble non vide de termes clos appelées décisions : $D \subseteq \mathcal{T}(\mathcal{F})$;*
3. *R est un ensemble fini de règles de réécriture sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
4. *Q est un ensemble de termes clos de $\mathcal{T}(\mathcal{F})$ tel que $Q \subseteq \mathcal{T}(\mathcal{F})$ appelé demandes d'accès ;*
5. *ζ est une stratégie de réécriture pour R.*

Nous expliquons les éléments dans la définition ci-dessus.

– Nous considérons dans un premier temps que les spécifications de politiques et leurs environnements sont décrits comme des termes construits sur une signature $\mathcal{F}$.
– L'ensemble des décisions possibles produites par la politique est noté D. Dans la plupart des langages pour le contrôle de d'accès, D est un ensemble contenant seulement les constantes $permit$ et $deny$. Cependant, nous verrons plus tard qu'il est crucial de modéliser des politiques qui s'abstiennent explicitement de renvoyer une décision. Par conséquent, il est utile d'avoir des éléments additionnels dans l'ensemble de décisions, comme $not\ applicable$, qui exprime le fait qu'une politique donnée ne peut pas décider d'une demande d'accès dans le contexte courant [Bruns et al., 2007]. Le résultat renvoyé par une politique pouvant être davantage élaboré qu'une constante, il s'agit alors d'un terme contenant davantage d'information : par exemple, le temps ou la durée pour laquelle un accès est accordé.
– Le système de réécriture R décrit le comportement de la politique, et au besoin, tous les calculs qui aident à expliquer comment l'environnement de politique évolue.
– Les demandes d'accès sont des termes clos. Elles expriment typiquement des questions de forme : est ce que le système peut faire une transition vers un autre état, étant donné la configuration courante de l'environnement de politique ?
– Le dernier composant est la stratégie qui permet d'indiquer précisément la façon dont les règles de politique doivent être appliquées. Ainsi, une stratégie appropriée choisit des dérivations dans R qui mènent à des décisions d'accès.

Cette approche a pour avantage l'expressivité, que nous allons illustrer par des exemples, et l'application des résultats ayant déjà été prouvés dans la théorie de la réécriture – parmi ces résultats, nous nous concentrons sur la confluence, la terminaison et la complétude suffisante des systèmes de réécriture.

Example 2. *Comme mentionnée précédemment, nous pouvons modéliser une politique de contrôle d'accès pour un système médical (cet exemple est adapté de la spécification [Moses, 2005] de XACML, et présenté dans le formalisme basé sur la réécriture dans [de Oliveira, 2007]).*

– *La signature $\mathcal{F}$ de la politique contient les symboles suivants, avec leurs déclarations de profil correspondants :*

$$\begin{array}{lll}
accs & : Request \times Attribute & \rightarrow Decision \\
req & : Subject \times Action \times Object & \rightarrow Request \\
read,\ write & : & \rightarrow Action \\
permit,\ deny,\ na & : & \rightarrow Decision \\
patient,\ phy & : Number & \rightarrow Subject \\
admin,\ per & : Number & \rightarrow Subject \\
record & : Number & \rightarrow Object \\
guard & : Subject \times Subject & \rightarrow Attribute \\
respPhy & : Subject \times Subject & \rightarrow Attribute
\end{array}$$

Dans la signature ci-dessus, la sorte de la valeur renvoyé par la fonction $accs$ est $Decision$, qui est la même pour les constantes $permit$, $deny$ et na. Avec des signatures ordo-sortées*, nous pourrions donner une représentation plus naturelle pour ces symboles, comme cela a été proposé dans [de Oliveira, 2007]. Cependant, dans cette thèse, nous avons préféré garder la compatibilité avec le système Tom, qui traite seulement des signatures* multi-sortées.

– *L'ensemble de décisions est $D = \{permit, deny, na\}$.*

– *$R = R' \cup R''$ est l'ensemble de règles ci-dessous, avec comme variables $x, y : Number$; $r : Object$; $a : Attribute$:*

$$R' = \begin{cases}
accs(req(patient(x), read, record(x)), a) & \rightarrow permit \\
accs(req(per(x), read, record(y)), guard(per(x), patient(y)))) & \rightarrow permit \\
accs(req(phy(x), read, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\
accs(req(phy(x), write, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\
accs(req(admin(x), read, r), a) & \rightarrow deny \\
accs(req(admin(x), write, r), a) & \rightarrow deny.
\end{cases}$$

$$R'' = \{\ accs(q, a) \ \rightarrow \ na.$$

Dans l'ordre de leur occurrence, ces règles déclarent qu'un patient peut lire son propre dossier médical, qu'un tuteur d'une personne peut lire son dossier médical, le médecin responsable pour un patient peut lire ou écrire des données dans son dossier. De plus, les deux dernières règles refusent n'importe quelle demande d'accès des administrateurs aux dossiers des patients. Enfin, la dernière règle s'applique par défaut dans cette politique, où $q : Request$.

– *L'ensemble de requêtes d'accès Q contient tout terme bien typé avec $accs$ comme symbole de tête.*

– *On peut adopter la stratégie* $\zeta = \texttt{choice}(R', R'')$. *Les termes dans* Q *qui ne sont pas réduits par les règles de* R' *seront réécrits en* na *par* R''.

Ce résumé introduit de manière large les éléments nécessaires à la compréhension des contributions de cette thèse. Nous développerons plus loin de nombreux exemples.

Les Propriétés des Politiques de Sécurité

Cette section discute quelques propriétés cruciales des politiques de sécurité. Nous commencerons par définir la *cohérence* d'une politique, qui est un problème récurrent dans plusieurs approches basées sur des règles permettant d'exprimer à la fois des permissions et des interdictions. Ensuite nous présenterons la propriété de *terminaison*, qui s'assure que chaque évaluation de demande d'accès par une politique est finie, puis nous définirons la *complétude d'une politique*, propriété assurant qu'il existe une décision d'accès correspondante à chaque demande.

Cohérence Une politique de sécurité est cohérente si tous les calculs pour une demande d'accès donnée mènent au plus à une décision. Néanmoins, les règles d'une politique sont intrinsèquement partielles ou contradictoires. Les auteurs de politiques de sécurité ont des difficultés à exprimer ce qui est admis et ce qui est interdit, notamment par rapport aux ambiguïtés. Par conséquent, il est extrêmement important d'avoir des techniques robustes de raisonnement détectant les contradictions parmi les règles d'une politique.

Definition 3 (Cohérence)**.** *Une politique de sécurité* $\wp = (\mathcal{F}, D, R, Q, \zeta)$ *est* cohérente *si pour chaque requête* $q \in Q$, ζ *appliquée à* q *renvoie au plus un seul résultat :* $\forall q \in Q$, *la cardinalité de* $[\zeta](q) \cap D$ *est inférieure ou égale à* 1.

L'exemple suivant illustre cette propriété :

Example 3. *Soit un conducteur qui croise un feu* (tl). *La politique qui doit être respectée est :*

$$
\begin{aligned}
\wp_1 = (\quad \mathcal{F}_1 &= \begin{cases} red, green, amber & : Color \\ stop, go & : Decision \\ tl & : Color \rightarrow Decision \end{cases} \\
D_1 &= \{stop, go\}, \\
R_1 &= \begin{cases} tl(red) & \rightarrow stop, \\ tl(green) & \rightarrow go, \\ tl(amber) & \rightarrow go, \\ tl(amber) & \rightarrow stop \end{cases} \\
Q_1 &= tl(\mathcal{T}(\mathcal{F}_1)), \\
\zeta_1 &= \texttt{universal}(R))
\end{aligned}
$$

Ainsi $\wp_1$ *n'est pas une politique cohérente, car* $[\zeta_1](tl(amber)) = \{stop, go\}$. *Par contre, une stratégie alternative possible,* ζ_2, *génère une politique cohérente et exécute les règles dans l'ordre présenté ci-dessus. Dans ce cas,* $[\zeta_2](tl(amber)) = go$, *peut être considéré comme une stratégie pour un conducteur pressé. Un conducteur prudent pourra préférer*

la stratégie ζ_3, qui donne priorité à la dernière règle. On obtient alors une politique cohérente parce que $[\zeta_3](tl(amber)) = stop$.

Dans cette thèse nous identifions des conditions nécessaires à la cohérence de politiques basées sur la réécriture en lui associant la propriété de confluence du système de réécriture sous-jacent.

Terminaison La terminaison est une question d'analyse essentielle dans le domaine des politiques de sécurité. Cette propriété permet de dire si le processus d'évaluation induit par les règles d'une politique est fini. Ce problème peut être défini comme suit :

Definition 4 (Terminaison)**.** *Une politique de sécurité $\wp = (\mathcal{F}, D, R, Q, \zeta)$ termine si pour toute requête q dans Q, toute dérivation de source q dans ζ est finie.*

Dans cette thèse nous avons donné des conditions nécessaires à l'application de techniques assez puissantes pour vérifier la terminaison de systèmes de réécriture, tels que les chemins récursifs sur les ordres [Dershowitz, 1987], l'étiquetage sémantique [Zantema, 1995], les paires de dépendance [Arts and Giesl, 2000], etc. Ces techniques et bien d'autres ont été implantées dans plusieurs outils qui peuvent vérifier la terminaison d'une classe large de systèmes de réécriture : par exemple AProVe [Giesl et al., 2004], TTT [Hirokawa and Middeldorp, 2007], et CiME [Marché and Urbain, 2004]. Toutes ces méthodes et outils de vérification testent la terminaison de façon statique, ce qui présente un avantage intéressant pour la spécification des politiques flexibles de sécurité.

Complétude d'une Politique Une autre propriété importante pour les politiques de sécurité est la capacité à évaluer toute requête d'accès entrante en au moins une autorisation. Nous appelons cette propriété "complétude d'une politique". Dans les travaux de [Tschantz and Krishnamurthi, 2006] et [Barker and Fernández, 2006] cette propriété est appelée "totalité". Nous définissons cette propriété comme suit :

Definition 5. *Une politique de sécurité $\wp = (\mathcal{F}, D, R, Q, \zeta)$ est complète si $\forall q \in Q, \exists d \in D$, tel que $[\zeta](q) \xrightarrow{*} d$.*

La propriété de complétude pour les politiques basées sur la réécriture correspond à la notion classique de complétude suffisante d'un système de réécriture. Cette notion établit que tout terme clos s'évalue dans un terme contenant exclusivement des constructeurs et peut être des variables [Comon, 1986, Kapur et al., 1991]. Plusieurs algorithmes ont été développés pour vérifier la complétude suffisante ou pour compléter un ensemble de motifs de façon à garantir cette propriété [Bouhoula, 1994]. Dans [Gnaedig and Kirchner, 2006], les auteurs ont présenté une méthode alternative où il est possible de prendre en compte des systèmes faiblement terminants.

Le travail présenté ci-dessus a comme objectif principal de donner une sémantique formelle à un langage de politique de sécurité expressif, incorporant des contraintes dynamiques sur l'environnent de la politique et avoir des définitions récursives. De plus, notre cadre formel permet la définition de politiques où les décisions ont une structure et des valeurs par défaut. La caractérisation des propriétés de politiques que nous avons par l'association des propriétés du système de réécriture sous-jacent fournit un niveau élevé de confiance aux développeurs logiciels. Ces

derniers peuvent utiliser des techniques et des outils avancés pour vérifier les politiques dans leurs systèmes.

Néanmoins, toutes ces propriétés étant indécidables en général, il est important d'identifier des classes de systèmes de réécriture pour lesquelles nous pourrons toujours avoir une procédure qui vérifie une de ces propriétés.

Classes Décidables de Systèmes de Réécriture

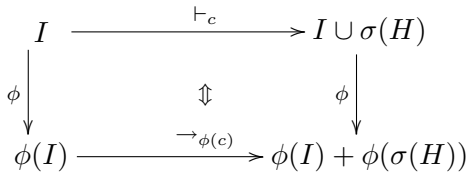
Une des avantages à employer des systèmes de réécriture dans des spécifications de politiques de sécurité est la possibilité de décrire des fonctions (récursives) complexes qui peuvent être impliquées dans le calcul des décisions d'accès. D'autre part, quand nous employons la pleine puissance expressive fournie par la réécriture et par les stratégies, nous devons tenir compte que la quasi-totalité des propriétés importantes pour des politiques deviennent indécidables. Par conséquent, il est important d'identifier quelques configurations appropriées afin de s'assurer que des procédures de décision existent bien pour l'examen de la cohérence, de la complétude par rapport aux requêtes et de la terminaison des politiques basées sur la réécriture. Les spécifications des politiques qui suivent une syntaxe restreinte pour les règles de réécriture donneraient quelques garanties dans ce sens, plutôt que dans le cadre général présenté dans la section précédente.

Nous avons examiné les résultats de décidabilité pour des systèmes de réécriture et présenté des classes syntaxiques des systèmes de réécriture (comme, *shallow, flat*, ou systèmes linéaires) pouvant avoir des bonnes propriétés, en particulier, sous une certaine stratégie fixe, c'est-à-dire, *innermost*, *outermost*, etc. Maintenant, nous allons introduire une approche alternative et inédite qui permet d'avoir des systèmes de réécriture toujours terminants, en créant une correspondance entre les programmes Datalog et les systèmes de réécriture.

Relation de Datalog et de la réécriture de termes Datalog [Ceri et al., 1989, Ceri et al., 1990] est un langage de programmation logique utilisé à l'origine pour exécuter des requêtes déductives sur des bases de données relationnelles. Ce langage est apparu pendant les années 80, et a été principalement étudié dans le cadre de la recherche sur les bases de données. Les principaux avantages de Datalog sont sa flexibilité et sa basse complexité. Il a été massivement employé en sécurité pour la spécification de politiques de contrôle d'accès [Fournet et al., 2007, Dougherty et al., 2006, Naldurg et al., 2006] et gestion de la confiance [Winslett et al., 2005, Li and Mitchell, 2003]. Nous allons présenter une transformation des programmes Datalog vers un système de réécriture équivalent. Cette traduction préserve les avantages de Datalog et nous permet de déterminer une classe syntaxique de systèmes de réécriture qui terminent toujours.

La transformation que nous développons dans cette thèse génère un système de réécriture conditionnel qui simule l'exécution du programme originel Datalog lors de la dérivation de nouveaux faits (de termes clos). L'avantage d'une telle traduction est le calcul d'un point fixe sur un terme associatif et commutatif, de la même manière que Datalog. Les résultats obtenus par l'application du système de réécriture généré par la transformation sont identiques aux faits clos qui seraient produits par le programme correspondant en Datalog. Pour prouver cela, nous

avons montré que le diagramme suivant commute (où ϕ est la fonction de traduction, I est un ensemble de faits clos, et σ une substitution) :



La transformation donnée ne tient pas compte de l'évaluation de buts (des requêtes avec des variables). Par contre, la résolution de buts peut être obtenue avec l'application d'une technique d'optimisation tel que les *magic sets* [Bancilhon et al., 1986] au programme Datalog d'origine avant sa traduction vers un système de réécriture.

Les travaux décrits dans la section courante représentent la partie fondamentale de la thèse. Dans les prochaines sections, nous nous appuierons sur ce modèle formel pour expliquer les contributions restantes de cette thèse. En plus de l'expressivité, l'adoption des stratégies est utile pour préserver des propriétés sous composition et pour la définition des opérateurs de composition de politiques eux-mêmes. L'exécution des moniteurs de référence basés sur la réécriture s'appuie également sur ce cadre, ainsi que la vérification des propriétés de flux de l'information.

1.2.2 Vérification de Propriétés

En plus des spécifications et de la composition de politiques de sécurité, nous présenterons également un algorithme d'analyse pour la vérification flux de l'information dans les systèmes de contrôle d'accès multi-niveaux. L'algorithme a été écrit dans le langage Tom [1] [Balland et al., 2007], qui est une implantation des systèmes de réécriture en Java. Ce travail est apparu dans [Morisset and de Oliveira, 2007]. D'ailleurs, nous avons également développé un algorithme permettant de vérifier la sûreté (comme définit dans le modèle HRU) en codant les modèles de contrôle d'accès comme des politiques basées sur la réécriture.

Le principe des politiques de flux d'information est d'éviter le stockage des données de niveau de sécurité plus élevé dans des objets appartenant à des niveaux de sécurité plus bas. Plus généralement, l'information sensible ne devrait pas être accessible aux sujets moins privilégiés. Par exemple, une politique de flux d'information peut déclarer qu'un sujet ne doit pas être autorisé à copier un fichier sensible, comme /etc/shadow (qui contient tous les mots de passe chiffrés dans un système de fichiers UNIX), vers un dossier dont les droits d'accès sont moins restrictifs, même si un programme en train d'être exécuté par root essaye d'effectuer une telle opération.

Les mécanismes utilisés afin d'imposer ce genre de politique impliquent du contrôle d'accès discrétionnaire et une politique obligatoire au niveau du système qui stipule la possession des ressources par différents utilisateurs et la possibilité pour le système de dépasser, dans certaines circonstances, notamment dans le cas de violation de la propriété de flux d'information, toutes les permissions des utilisateurs. Il s'agit de l'approche suivie par Bell et LaPadula dans leur travail séminal sur le contrôle d'accès obligatoire [Bell and LaPadula, 1974]. Néanmoins, il est très

[1] http://tom.loria.fr/

difficile de prouver qu'un système débutant par une configuration valide atteindra seulement des états surs ou valides, par l'analyse des règles d'une certaine politique. C'est en partie dû au fait que des propriétés typiques de flux d'information sont exprimées en termes de traces lecture/écriture et écriture/lecture souhaitables dans un système. En général, comme les systèmes sont abstraits comme des machines de Turing, la vérification d'une propriété de flux d'information correspond à tester si une telle machine atteint un état donné, ce qui est en général indécidable.

Dans cette partie, nous montrons que les accès indiqués correctement par une politique de sécurité ne sont pas uniquement basées sur les traces de lecture-écriture. En effet, certains modèles de sécurité peuvent faire des demandes d'accès qui violent indirectement la politique. Nous proposons une méthode explicitant de tels flux d'information à partir d'un état donné du système. Nous employons des règles de réécriture pour décrire la politique de contrôle d'accès et explorer les flux d'information sur des états accessibles d'un système. Nous présentons une technique d'analyse qui peut automatiquement identifier les fuites d'information pouvant invalider une implantation d'une politique de flux d'information.

Un algorithme d'analyse de flux d'information Nous allons fournir un algorithme basé sur un travail fait précédenment sur l'analyse de protocoles de sécurité [Cirstea et al., 2005]. Le type de vulnérabilité recherché ici se fonde sur le fait que certaines séquences permises d'actions de écriture/lecture peuvent fournir des contre-exemples pour la propriété globale de sécurité.

Un flux d'information est implicite lorsqu'un sujet peut lire un objet, mais que la permission d'accès correspondante n'est pas dans la matrice courante de contrôle d'accès. Voici un exemple de flux implicite d'information : si un sujet s_1 lit un objet o_1 et écrit dans un objet o_2, et si un sujet distinct s_2 lit l'objet o_2, alors nous pouvons considérer que s_2 peut également lire l'objet o_1. En d'autres termes, si les accès $(s_1, o_1, read)$, $(s_1, o_2, write)$ et $(s_2, o_2, read)$ appartiennent à l'ensemble de tous les privilèges dans le système, alors l'accès $(s_2, o_1, read)$ devrait également appartenir à cet ensemble.

L'algorithme que nous proposons consiste à expliciter chaque flux d'information dans la configuration d'un système et à vérifier que chaque flux est autorisé par la politique de sécurité en place. Quand la politique refuse ce genre d'accès, alors un flux d'information est caractérisé. Si la politique autorise l'accès implicite alors, selon son modèle de sécurité, le flux n'est pas un danger au regard de la politique mise en place.

Nous appliquons notre technique sur des modèles bien connus de la littérature sur la sécurité informatique. Nous considérons le modèle de sécurité proposé par McLean dans [McLean, 1988], dont l'intention est de généraliser le modèle de Bell et de LaPadula pour traiter l'accès commun (ou "joint" où l'accès est conféré s'il est requis par un ensemble de sujets de façon concurrente). Les simplifications de ce cadre proposés en vue d'obtenir un modèle équivalent au BLP compromettent la propriété de sécurité [Jaume and Morisset, 2006a]. Le travail décrit ici est paru dans [Morisset and de Oliveira, 2007].

Le principe permettant de vérifier la généralisation par McLean du modèle BLP réside dans les différentes manières dont la configuration initiale des ensembles d'accès peut être créée, étant donnés un ensemble de sujets et d'objets. En fait, l'entrée pour notre algorithme concerne la taille de l'instance à vérifier – le nombre de sujets et d'objets. Nous produisons alors différents éléments sur chaque ensemble avec différents niveaux de sécurité – dans la mesure où il n'est

pas inintéressant de vérifier le flux de l'information pour des éléments du système ayant tous les mêmes étiquettes. Nous choisissons de placer notre recherche sur ce point plutôt que de donner toutes les combinaisons possibles de classifications aux différents objets et sujets et fournir une stratégie d'exploration en largeur ou en profondeur pour explorer les états atteignables du système.

Pour conclure, la technique que nous avons présentée peut être adaptée afin de vérifier d'autres modèles de contrôle d'accès. Dans ce cas, il faut substituer à l'implantation de la politique, celle qui correspond au nouveau modèle.

Vérification de la Sûreté pour les Politiques Basées sur la Réécriture Nous allons revenir sur le problème de la sûreté dans des modèles de contrôle d'accès, qui a été défini par Harrison, Ruzzo et Ullman [Harrison et al., 1976]. Cette question permet de déterminer si une certaine configuration (état d'un système) permet la fuite d'un privilège à un sujet non fiable. Dans leur travail [Harrison et al., 1976], les auteurs fournissent un ensemble simplifié d'opérations, pour lesquelles la complexité est limitée. Ces opérations contrôlent l'assignation et le retrait de droits dans une matrice de contrôle d'accès. Ils ont prouvé que la sûreté est indécidable en général. Ceci ne signifie pas pour autant l'impossibilité de fournir un algorithme pour vérifier la sûreté d'un ensemble spécifique de commandes construit à partir de ces opérations et d'une configuration de système donnée. Nous codons dans ce but le modèle HRU dans notre cadre basé sur la réécriture et nous analysons la sûreté pour certains de ses exemples.

Nous fixons les objectifs suivants :
- illustrer comment ce modèle peut être codé dans notre cadre,
- fournir une implantation courante du modèle HRU qui peut être réutilisée avec d'autres objectifs,
- raisonner sur les propriétés élaborées, telles que la sûreté, en utilisant des techniques de réécriture de termes, et
- identifier les raisons de l'indecidabilité du problème de sûreté.

Dans le codage, nous avons défini que les demandes d'accès correspondent aux appels de commandes, dont l'exécution est effectuée par un ensemble de règles de réécriture transformant la matrice de contrôle d'accès courante pour une certaine configuration. Les règles mettent en application l'effet prévu par les instructions primitives (pour créer un nouveau sujet, ou objet, pour ajouter un droit d'accès dans à une position donnée de la matrice, etc..), seulement si les conditions sur la commande sont satisfaites. Nous séparons les spécifications de l'exécution de commande en deux parties. La première partie nous indique comment consommer une instruction de la liste d'instructions d'une commande. La deuxième partie contrôle la modification de la matrice selon l'opération primitive exécutée.

Dans HRU, la sûreté est la propriété qui traduit l'impossibilité pour tout état obtenu par l'exécution d'une instruction de perdre un droit dans un système de protection et dans une configuration donnés. Cette propriété peut être exprimée de la manière suivante dans notre codage.

Soient m une matrice , c une commande et m' la matrice obtenu par l'exécution de c sur m via la dérivation suivante

$$q(cmd(cl, i \odot il), m) \xrightarrow{*} q(cmd(cl, il), m')$$

où i est une opération primitive, cl est une liste de conditions, et il est une liste d'instructions. Nous disons que m' *perd un droit* r s'il y a un objet $o(x, sl)$ dans m', où sl est une liste de sujets, et un sujet $s(y, r + rl))$ tels que $check(in(r, x, y), m) \xrightarrow{*} NotOK$, pour un certain r. Ou, plus simplement, étant donné un droit $r \in M'[x, y]$, (dans la notation standard), la question est de savoir si ce même droit appartient à la même entrée dans la configuration précédente de la matrice : $r \in M[x, y]$.

Nous présentons une procédure pour tester la perte d'un droit (fuite) entre deux états du système, que nos appellerons ici de $safe(m', m)$ où m' représente l'état postérieur à m. L'algorithme pour la sûreté consiste à : étant donnés un ensemble d'instructions $\{c_1, \ldots, c_n\}$ et une configuration initiale m_0, exécuter chaque permutation des commandes [2] dans un ordre donné, en réduisant le terme $q(c_i, m), 1 \leq i \leq n$; Pour chaque exécution d'une opération primitive ayant comme conséquence une nouvelle matrice m', appeler la fonction $safe(m', m)$. Si le résultat de cet appel est faux pour une paire donnée (m', m) alors une fuite est détectée. Si le résultat est vrai, l'algorithme continue jusqu'à ce que toutes les permutations soient examinées. Dans ce cas-ci, un système de protection est sûr pour tout droit r. Bien que coûteux, cet algorithme permet d'examiner la sûreté pour des systèmes de protection dans HRU.

Pour appliquer cet algorithme, les commandes doivent avoir leurs paramètres instanciés. En effet, il n'y a aucune variable dans les requêtes pour notre implantation. C'est la simplification la plus importante que nous avons apporté dans ce développement. Nous pouvons calculer correctement la prochaine configuration d'un système de protection à chaque étape de l'exécution d'une commande. Ainsi, la vérification de la sûreté dans cette classe des systèmes de protection devient possible.

Le problème général dans HRU est indécidable parce qu'il serait nécessaire de vérifier chaque instantiation possible des paramètres pour appliquer le procédé présenté ci-dessus. Étant donné que les identificateurs de sujets et des objets sont définis sur l'ensemble Nat, nous avons un nombre infini de sujets et d'objets qui peuvent être créés par différentes instantiations de la même commande. De plus il existe un nombre infini de configurations à examiner pour une fuite de droits. En conséquence, le procédé que nous avons donné pourrait ne pas terminer.

1.2.3 Composition des Politiques avec des Stratégies de Réécriture

Une autre contribution apportée par cette thèse concerne le domaine de la composition de politiques basées sur des règles. Les politiques dans notre cadre formel peuvent renvoyer multiples décisions d'accès possibles, comme dans la norme industrielle XACML [Moses, 2005], et bien d'autres langages de spécification de politiques de sécurité. Ces langages exigent une théorie de composition de politiques qui s'adapte en juste proportion aux besoins courants de composition. Contrairement à d'autres approches, notre cadre peut traiter plusieurs situations concrètes de composition sans introduire de contradictions à la logique utilisée comme base. En outre, notre cadre de composition en politique est extensible : nous pouvons définir de nouvelles formes de combinaison de politiques en déterminant de façon explicite l'ordre dans lequel les règles doivent être appliquées, ou en fournissant des règles qui permettent de éliminer les ambiguïtés parmi des décisions possiblement contradictoires et produites par les sous-politiques. Ce travail

[2] Il n'est pas nécessaire de changer l'ordre des instructions à l'intérieur des commandes.

est la principale contribution apportée par [Dougherty et al., 2007b].

Dans ce contexte, l'utilisation des stratégies de réécriture a deux buts principaux : contrôler la façon dont les règles de réécriture dans une politique doivent être appliquées afin de dériver une décision d'accès et donner la sémantique de la politique combinant des algorithmes. Le choix d'une stratégie de réécriture donnée peut également faciliter la conservation de différentes propriétés des composantes dans le contexte d'une politique globale.

Nous pouvons admettre que les règles décrivant les diverses politiques de sécurité dans de grands organismes ne sont pas décrites et maintenues dans une seule politique monolithique. Il est utile de décomposer la complexité de grandes spécifications dans des ensembles de règles plus petits. Des politiques plus petites habituellement couvrent quelques types de demandes d'accès, pouvant être mieux comprises et avoir des preuves plus concises de leurs propriétés. Le besoin de politiques modulaires apparaît dans plusieurs situations pratiques, par exemple, quand différents organismes (ou différents départements d'une même organisation) doivent coopérer. Très souvent, les entités impliquées dans un système ont des conditions différentes pour le contrôle d'accès, pouvant être incompatibles ou contradictoires, puisqu'elles mettent différents accents sur les buts de sécurité. Par exemple, dans un hôpital, une politique contient des règles régissant l'accès des patients à leurs dossiers médicaux et financiers, mais aussi des règles pour contrôler l'accès des employés administratifs à ces mêmes dossiers, comme aux ressources tout à fait différentes des dossiers médicaux. En attendant, d'autres entités telles que des compagnies d'assurance sont sujettes à encore un autre ensemble de règles pour l'accès à ces données. Il est nécessaire, donc, de disposer d'outils théoriques qui puissent nous aider à comprendre dans quelles situations la composition de l'ensemble des sous-politiques préserve les propriétés de sécurité.

Les contributions décrites ici visent à résoudre deux problèmes centraux en composition de politiques de sécurité :

- fournir des mécanismes pour combiner des politiques avec une sémantique formelle uniforme. Dans notre cadre, les utilisateurs peuvent définir comment les politiques sont combinées en ajoutant des nouveaux symboles de fonctions et des stratégies pour la politique combinée ; et
- permettre le raisonnement automatisé au sujet de l'interaction entre les règles d'accès de multiples politiques combinées et sur la manière dont cette interaction affecte le comportement de la politique globale, en particulier, si les propriétés des politiques composantes sont préservées.

Nous verrons que la réécriture stratégique est un paradigme expressif permettant de définir divers types de combinateurs de politiques, grâce à sa capacité à sélectionner les dérivations d'intérêt pour la politique en question parmi celles induites par les règles de réécriture dans les systèmes composants. Nous expérimentons diverses possibilités pour la composition de politiques, telles que des opérations de composition proposés par le standard XACML, en plus des compositions séquentielles et conservatrices. Nous utilisons également des résultats connus sur la modularité des systèmes de réécriture, en particulier, pour les propriétés de cohérence et de terminaison des politiques de sécurité.

Définitions

La définition présentée ci-dessous nous permet d'appliquer directement de nombreux résultats au sujet des propriétés modulaires des systèmes de réécriture. Elle est compatible avec la définition d'une politique de sécurité basé sur la réécriture donnée précédemment. Une combinaison de deux politiques ou plus est donc une politique de sécurité. Ici, nous considérons des combinaisons de politiques par paires, sans perte de généralité.

Definition 6 (Policy Composition)**.** *Une* composition *de deux politiques* $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_i)$ *(*$i = 1, 2$*) est une politique* $\wp = (\mathcal{F},\ D,\ R,\ Q,\ \zeta)$*, où :*

1. $\mathcal{F}_1 \cup \mathcal{F}_2 \subseteq \mathcal{F}$ *;*
2. $D_1 \cup D_2 \subseteq D \subseteq \mathcal{T}(\mathcal{F})$ *;*
3. $R_1 \cup R_2 \subseteq R$ *;*
4. $Q_1 \cup Q_2 \subseteq Q \subseteq \mathcal{T}(\mathcal{F})$ *;*
5. ζ *est une stratégie de réécriture pour* R*.*

Nous pouvons faire les observations suivantes au sujet de cette définition :

– En combinant des politiques, il peut être nécessaire d'y ajouter des symboles n'appartenant pas aux politiques originales. C'est pourquoi la signature $\mathcal{F}$ peut contenir plus de symboles que $\mathcal{F}_1 \cup \mathcal{F}_2$.
– L'ensemble des requêtes d'accès pour la politique combinée contient des termes de la forme déterminée par ses sous-politiques, mais peut également contenir des termes additionnels bien formés qui peuvent être construits à partir de la signature combinée de politique. Par exemple, supposons que $\mathcal{F}_1 = \{0, f\}$, $Q_1 = f(\mathcal{T}(\mathcal{F}_1))$ et $\mathcal{F}_2 = \{g\}$, $Q_2 = g(\mathcal{T}(\mathcal{F}_2))$, alors une demande valide dans le contexte de la politique combinée serait $g(f(0))$.
– La stratégie pour la combinaison définit les séquences de réduction des termes appartenant à l'ensemble des requêtes d'accès de la politique composée. Intuitivement, la stratégie peut être exprimée comme la composition fonctionnelle des stratégies des sous-politiques, en employant n'importe quel opérateur de stratégie. Cependant, elle peut être construite d'une manière non-modulaire, telle que la nouvelle stratégie néglige les stratégies originalement utilisées par les sous-politiques. Cela est utile dans plusieurs situations, où il est nécessaire de donner une fonction d'évaluation plus appropriée pour le nouvel ensemble de termes représentant des requêtes d'accès. Par conséquent, nous fournissons dans cette définition le choix de réutiliser seulement une partie des règles de réécriture disponibles dans les politiques composantes et d'établir librement une nouvelle stratégie pour leur combinaison.
– Nous donnons la définition de composition de politiques en termes de deux politiques, mais naturellement la définition peut être étendue sans problème à n politiques via leur combinaison par paires.

Nous décrivons ci-dessous plusieurs exemples de combinateurs de politiques.

Combinateurs de Politiques de Sécurité

Nous avons exploré la sémantique et les propriétés formelles intéressantes des combinaisons de politique en employant le formalisme de réécriture stratégique.

Union des règles de politique Cet opérateur consiste en l'union simple de l'ensemble de règles de réécriture des politiques composantes. Ce genre de combinaison est bien compris dans la perspective de la réécriture de termes. Plusieurs résultats au sujet de la conservation des propriétés des systèmes combinés sont disponibles. En réécriture, une propriété est dite modulaire si elle est préservée par union. Dans ce travail, nous avons donné des exemples de politiques cohérentes et terminantes dont l'union ne satisfait plus ces propriétés. Néanmoins, nous présentons quelques conditions syntaxiques sur l'opération d'union et des stratégies qui préservent ces propriétés pour ce type de composition de politiques. En général, les propriétés de confluence et de terminaison ne sont pas modulaires. Cependant, plusieurs résultats au sujet de la modularité pour des systèmes de réécriture ont été produits. Un aperçu est donné dans [Ohlebusch, 2002a]. Grâce au travail que nous avons fait dans cette thèse pour définir les politiques basées sur la réécriture, toute la connaissance de cette théorie peut être importée dans le contexte de composition de politiques de sécurité.

Combinaison Séquentielle de Politiques Une combinaison séquentielle de politiques capture la nécessité d'évaluer davantage les décisions intermédiaires produites par une politique pour une demande d'accès, en utilisant les règles d'une deuxième politique, afin de produire une décision d'accès. La décision finale sera en forme normale en ce qui concerne la deuxième politique et sera un terme appartenant à son ensemble de décisions. Par conséquent, dans ce cas-ci, les décisions produites par la première politique est l'entrée pour la deuxième.

Extension Conservatrice de Politiques Une manière assez intuitive de réutiliser une politique est d'y ajouter des règles telles que la nouvelle politique générée puisse traiter une plus grande classe de demandes d'accès. Une manière sûre de créer une telle politique combinée consiste à construire une *extension conservatrice* de la politique originale. Dans ce cas, les décisions produites pour chaque requête d'accès dans le contexte de la politique originale ne sont pas changées par composition. Cette propriété est énoncée comme suit :

Definition 7 (Extension Conservatrice)**.** *La composition* $\wp = (\mathcal{F},\ D,\ R,\ Q,\ \zeta)$*, de deux politiques* $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_i)$ *(*$i = 1, 2$*) est conservatrice si* $\forall q_i \in Q_i$*, alors* $[\zeta](q_i) = [\zeta_i](q_i)$.

En d'autres termes, le résultat de l'évaluation d'une demande d'accès donnée sous la stratégie de la politique combinée est identique au résultat de son évaluation sous la stratégie composante originale. Une stratégie appropriée pour de telles combinaisons est `choice`, puisqu'elle assure que le premier ensemble de règles de réécriture sera appliqué à tout terme seulement en cas d'échec ; le deuxième ensemble de règles de réécriture est employé. Nous avons identifié dans cette thèse des situations où il est possible d'obtenir des compositions conservatrices en utilisant cette stratégie.

Les combinaisons conservatrices peuvent assurer la validité des principes d'autonomie et de sécurité définis dans [Cholvy and Cuppens, 1997]. L'autonomie signifie que n'importe quel accès autorisé dans le système individuel doit également être autorisé par la politique composée. Le principe de sécurité signifie que l'accès non autorisé par une politique isolée doit également être refusé par la politique composée.

Combinateurs de Politiques à la XACML Nous montrons que l'approche de réécriture stratégique peut capturer le comportement des combinateurs de politique de résolution de conflit, comme ceux proposés dans la norme industrielle de langage de contrôle d'accès XACML [Moses, 2005]. L'idée d'éliminer l'ambiguïté entre des décisions probablement contradictoires apparaît dans plusieurs initiatives précédentes, telles que dans [Jajodia et al., 2001, Kalam et al., 2003]. Cette idée est exactement le but des combinateurs de politique de XACML, dont les principaux opérateurs sont présentés ci-dessous.

- *permit-overrides* : une requête d'accès est autorisée si au moins une des réponses des sous-politiques à cette requête est $permit$. La politique produira un refus d'accès seulement si au moins une des sous-politiques refuse la demande et si toute autre renvoie la décision *not-applicable* ou $inderterminate$. Quand aucune des sous-politiques ne renvoie ni $permit$ ni $deny$, mais explicitement *not-applicable*, le résultat final est *not-applicable*. La décision est $indeterminate$ si toute sous-politique ne renvoie aucune décision, c'est-à-dire, quand des erreurs inattendues se produisent pour chaque tentative d'évaluation.
- *deny-overrides* : ce combinateur a une sémantique similaire à *permit-overrides* à la différence que les refus sont prioritaires.
- *first-applicable* : la décision produite par la politique combinée correspond à l'autorisation déterminée par la première sous-politique qui n'échoue pas, et dont la décision est différente de *not-applicable*.
- *only-one-applicable* : la décision résultante est $indeterminate$ si plus d'une sous-politique renvoie une décision différente de *not-applicable*. La décision résultante sera $permit$ ou $deny$ si la seule politique qui s'applique à la demande produit une de ces décisions. Le résultat sera *not-applicable* si toutes les politiques renvoient une telle décision.

Résultats Obtenus pour la Composition Nous montrons que la réécriture stratégique fournit les fondements et l'expressivité nécessaire pour dresser un cadre formel pour la composition de politiques de sécurité. La sémantique opérationnelle du langage de stratégie que nous utilisons nous permet de combiner des ensembles de décisions, d'accorder des priorités aux règles (et aux ensembles de règles) et d'interférer également sur le flux d'information entre les politiques par l'ordonnancement des demandes d'accès entre des politiques. Nous avons codé plusieurs approches existantes grâce à l'expressivité fournie par notre langage de stratégie.

Plus précisément, nous avons codé des combinateurs de résolution de conflits, comme ceux de XACML, supposant que les politiques peuvent avoir une nature contradictoire. Cette approche a été largement adoptée dans plusieurs travaux récents portant sur la composition de politiques [Moses, 2005, Bruns et al., 2007, Tschantz and Krishnamurthi, 2006].

Par ailleurs, notre définition de politique est très générale et nous pouvons capturer quelques politiques pour lesquelles il serait très difficile trouver une correspondance dans leur modèle abstrait. En effet, dans notre cadre, nous pouvons énoncer des permissions sur des concepts autres que des sujets, des objets et des actions.

L'autre avantage de notre cadre est que nous pouvons raisonner formellement sur la composition de politiques en présence d'une multiplicité de décisions d'accès, y compris lorsque les

politiques assignent explicitement une décision neutre pour certaines requêtes d'accès. Or ceci ne peut pas être capturé dans plusieurs cadres précédents.

Nous avons également montré l'intérêt de la réécriture stratégique dans la conservation de certaines propriétés de politique comme la cohérence, la terminaison et sur les extensions conservatrices de politiques.

1.2.4 Implantation des Politiques de Sécurité

Le système Tom a permis la construction de mécanismes pour imposer des politiques à des programmes. Un principe de conception dans le contrôle d'accès est la séparation entre la spécification d'une politique et le mécanisme pour l'imposer. Ceci coïncide avec le concept d'îlot formel [Balland et al., 2006b, Balland et al., 2006c], qui donne les bases pour inclure des langages formels dans des environnements de programmation à tout usage, principes qui ont guidé la construction du système Tom lui-même.

Nous montrons comment la programmation orientée par aspects peut être employée pour réaliser cette séparation, pendant que nous déployons des politiques basées sur la réécriture décrites dans le langage Tom et dans des programmes Java existants. Nous fournissons une méthodologie d'aide aux développeurs de logiciels à la construction de mécanismes fiables de contrôle d'accès pour des applications.

Cette méthodologie permet la manutention et la mise à jour des politiques sans besoin de modifier manuellement le code du programme sur lequel elles sont imposées.

Nous présentons une architecture pour imposer des politiques basées sur des règles à des programmes existants, en combinant le système Tom [Balland et al., 2007] et la programmation par aspects [Kiczales et al., 1997]. Dans Tom les politiques basées sur des règles peuvent être convenablement écrites. La programmation par aspects est une technique de modularisation qui fournit les outils nécessaires à l'exécution des transformations de programmes non-fiables vers des programmes respectant une certaine politique. Le travail décrit ici est paru dans [de Oliveira et al., 2007].

La difficulté à imposer des politiques flexibles provient du changement des conditions de sécurité au cours de l'exécution d'un système, contrairement à des modèles tels que la matrice de contrôle d'accès et RBAC. Dans ces modèles, l'ensemble de privilèges ne subit pas d'altération une fois fixé. Les politiques dépendantes de l'environnement doivent évaluer chaque requête selon l'état courant du programme courant pour fournir une décision de contrôle d'accès. Quelques modèles récents pour le contrôle d'accès mettent en avant le besoin de politiques flexibles de contrôle d'accès, comme par exemple [Kalam et al., 2003, Jajodia et al., 2001, di Vimercati et al., 2005].

Nous concevons un schéma générique pour imposer des politiques basées sur la réécriture dans des programmes cibles. La méthode permet de construire un moniteur de référence pour une politique donnée par *l'alignement* des règles de la politique à l'intérieur du programme. En outre, nous fournissons une méthodologie pour mettre en rapport la description de la politique avec le code d'un programme, en faisant explicitement la correspondance entre l'environnement de la politique et les données traitées le programme, appelé *cible*. Cette technique tient compte de la séparation entre les politiques et les programmes, en fournissant des modules réutilisables de politiques de sécurité.

Notre solution est construite à l'aide de deux outils principaux, Tom et AspectJ [Kiczales et al., 2001]. Ce dernier est une mise en oeuvre très populaire de la technique de programmation par aspects (*Aspect-Oriented Programming*, AOP, en anglais) [Kiczales et al., 1997] pour le langage Java. Leur inter-opération nous permet d'insérer des règles de contrôle d'accès comme un moniteur de programme, et par conséquence, de fournir une implantation efficace pour une politique donnée.

Programmation Orienté par Aspects Le concept de programmation orientée par aspects (AOP) a été présenté dans [Kiczales et al., 1997] comme une approche visant à identifier des fonctionnalités orthogonales aux fonctionnalités principales d'un programme, c'est-à-dire, des fonctionnalités indépendantes et sans lesquelles le programme principal satisfait les objectifs indiqués par l'utilisateur. Ces fonctions orthogonales sont appelées seulement à certains points de l'exécution d'un programme et peuvent normalement être groupées dans des unités simples appelées *aspects*.

Les aspects contiennent des comportements qui affectent des classes multiples (ou unités, modules, etc...) dans des fragments de code réutilisables. Puisque les aspects centralisent le code pour les fonctionnalités orthogonales, qui autrement seraient propagées au long du code du programme (même dans sa structure hiérarchique), l'AOP améliore rigoureusement la réutilisation potentielle de ces objets et diminue les coûts d'entretien.

Afin de donner un exemple (classique) de fonctionnalité orthogonale, considérons un système de gestion de conférence. Ses fonctionnalités de base sont la soumission des papiers, l'attribution des papiers aux membres de comité de programme et la soumission des revues. De plus, d'autres exigences non-fonctionnelles peuvent exister, telles que le registre des actions effectués dans le système par les utilisateurs dans un fichier. Dans ce cas, le code pour mettre en application ce "journal" est forcement dispersé dans plusieurs parties du programme.

Les blocs de base des programmes orientés par aspects sont les *pointcuts* et le *code advice*. Les pointcuts définissent où une fonctionnalité orthogonale donnée doit être appelée dans le programme. Les pointcuts peuvent être associés à un ensemble de noms de fonctions par exemple. Ils sont souvent exprimés par des motifs. Nous appelons *join points* les endroits dans le code de l'application qui filtrent un pointcut donné.

À son tour, le code advice est le code réel exécuté dans un join point. Dans un système de conférence, par exemple, un pointcut est n'importe quel appel à la méthode pour soumettre des papiers. Le code advice doit imprimer dans le journal du système l'identité de l'utilisateur courant et quelques informations sur le papier soumis, avec la date et l'heure.

La dernière étape essentielle dans AOP est de tisser (en anglais *weave*) des aspects dans le code de l'application principale. La compilation d'aspects est un processus de transformation de programmes qui filtre les pointcuts définis à l'intérieur des aspects et insère le code advice correspondant, avant, après ou autour d'un join point. L'expressivité d'un langage orienté aspects se fonde sur les spécifications des pointcuts, par exemple, sur les genres de motifs permis par le langage d'aspects, ou sur la manière dont le code advice peut être tissé. Pour expérimenter avec ces concepts, dans cette thèse, nous avons adopté AspectJ [Kiczales et al., 2001] qui est une implantation courante d'AOP pour Java.

Imposition de Politiques Basées sur la Réécriture Un moniteur de référence est un mécanisme pour imposer une politique de sécurité. Il observe l'exécution d'un programme sur lequel on ne peut pas avoir confiance (a priori) et peut interrompre son exécution dans le cas d'une violation de la politique de sécurité.

Des exemples de moniteurs de programmes incluent des pare-feux, des noyaux des systèmes d'exploitation et d'autres composants qui interceptent des appels faits par des applications cibles. Ces moniteurs peuvent être vus comme des unités indépendantes dans l'architecture de système, mais très souvent ils sont inclus dans le code de l'application, pendant la compilation du programme ou pendant son chargement. Dans ce dernier cas, le code non fiable est transformé de telle manière que les actions du programme sont forcées de passer par le moniteur, qui décide d'avorter son exécution ou pas, évitant de ce fait que le système entre dans un état non sûr.

Les langages adoptés dans les travaux précédents pour décrire la politique imposée par un moniteur de référence donné ont un niveau d'abstraction plutôt bas, comparable au langage employé pour décrire le programme cible [Bauer et al., 2005, Erlingsson and Schneider, 2000]. Par exemple, il est possible d'exprimer des politiques qui rejettent des appels système pour l'ouverture de dossiers sur une machine, ou de bloquer l'accès aux services de réseau après la lecture de données, mais il est difficile de coder le comportement dynamique d'une politique flexible, qui prend une décision d'autorisation basée sur les conditions courantes de son environnement.

Ainsi, le modèle de l'environnement de politique devient un élément crucial pour imposer des politiques flexibles. Dans cette thèse, l'état actuel de l'application est structuré comme un terme, qui est modifié au long de l'exécution du programme. À chaque fois que l'application exécute une action concernée par la politique, ce terme est évalué afin de calculer une décision d'accès en appliquant les règles de réécriture qui définissent la politique. La politique elle-même ne décrit pas comment l'état du système évolue, mais quelles transitions à partir d'un tel état sont autorisées. Cette approche s'appelle *execution monitoring*.

Architecture du Système Notre approche est illustrée dans la Figure 1.1. Nous mettons en évidence la correspondance entre la description formelle de la politique, par sa signature, et le code du programme. La politique déclare un certain nombre d'actions qui sont liées aux appels de fonctions dans le programme source. Ces actions déterminent des "méthodes sensibles" où des demandes d'accès doivent être faites au moniteur de référence. Ce moniteur est l'implantation des règles de la politique de sécurité, contenue dans un morceau de de code advice qui est "tissé" dans le code de l'application cible, produisant un programme qui respecte la politique.

Aspects pour le contrôle d'accès Ci-dessous nous décrivons les étapes nécessaires pour la mise en oeuvre de politiques basées sur la réécriture avec l'aide de la programmation par aspects.

1. *Obtention de l'environnement de politique* : Guidé par la construction d'un *mapping* entre la signature de politique et les objets traités par l'application cible, le développeur du programme doit définir un ensemble d'indicateurs pointcuts sur ces objets de façon à capturer des changements de leurs valeurs. La manière la plus pratique de contrôler ceci est de déclarer quelques variables auxiliaires dans l'aspect pour garder les dernières valeurs de tels objets. Par exemple, nous considérons l'utilisateur courant connecté au système :

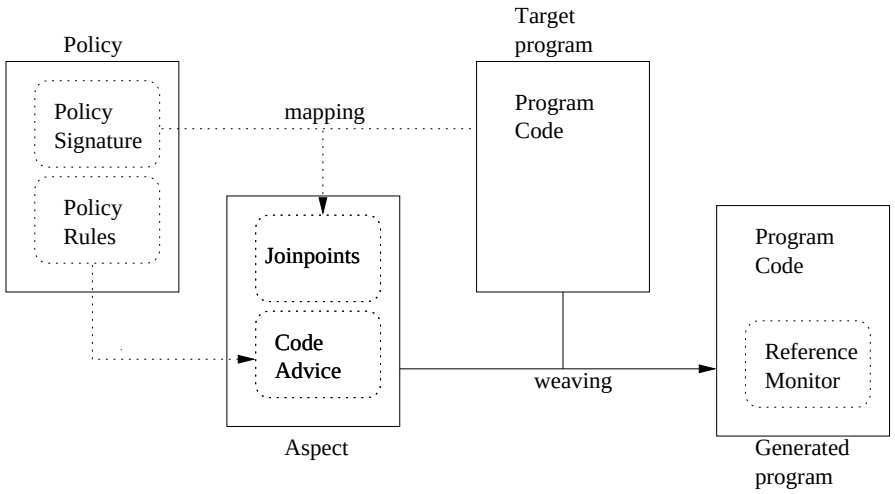


FIG. 1.1: Architecture du Système

il est possible de déclarer une variable globale qui garde cette information au moment où l'utilisateur est authentifié par l'intermédiaire de son identification et mot de passe.

2. *Identification des demandes d'accès* : dans les modèles classiques de contrôle d'accès, la notion d'opération, ou d'action effectuée par les entités actives du système a été toujours placée comme l'origine des demandes d'accès. Dans la pratique, chaque appel de fonction (ou appel de méthode dans le paradigme orienté à objets) implique en une demande d'accès. Par contre, dans notre cadre formel, l'ensemble de requêtes est indiqué dans la définition de la politique de sécurité. Ceci oblige à imposer une politique plus mécanique. De plus, des méthodes formelles existent pour vérifier qu'une politique est complète par rapport à un ensemble de requêtes d'accès. Néanmoins la définition de la politique doit prendre en compte tout accès aux données qui doivent être protégées.
3. *Tissage (weaving) des règles de la politique* : Dans Figure 1.2 nous illustrons les interactions entre les composants et les outils utilisés dans notre processus de tissage. Puisque le compilateur Tom produit du code Java, il n'y a aucune restriction pour faire les transformations de programme nécessaires pour l'application du contrôle d'accès avec les fonctionnalités disponibles dans AspectJ.

Résultats Obtenus pour L'implantation de Politiques de Sécurité Le besoin de politiques flexibles est prédominant dans les systèmes d'information d'aujourd'hui. La compréhension de la façon de contrôler des contraintes complexes sur l'environnement des politiques a augmenté sensiblement avec l'introduction de plusieurs formalismes basés sur des règles pour le contrôle d'accès [Dougherty et al., 2006, Antoniou et al., 2007, Jajodia et al., 2001, di Vimercati et al., 2005]. Ceci est dû au fait que des langages basées sur des règles peuvent exprimer ces politiques raisonnablement bien, dans des cadres théoriques permettant d'exécuter de l'analyse statique pour détecter sur les vulnérabilités possibles d'une politique.

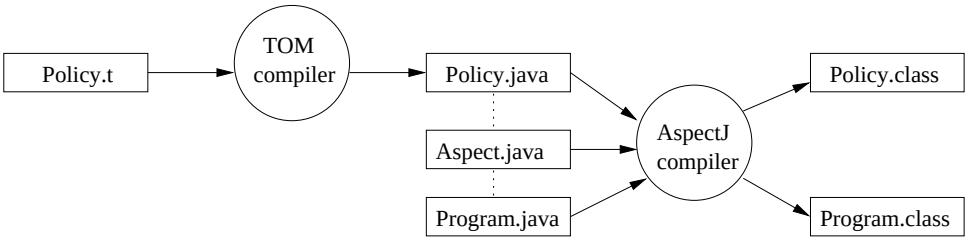


FIG. 1.2: Tissage des règles de contrôle d'accès avec AspectJ et Tom

Dans notre modèle [de Oliveira, 2007, Dougherty et al., 2007b], les correspondances entre les propriétés du système de réécriture et la politique qu'elle met en place sont claires, ainsi nous pouvons appliquer directement un corpus riche de techniques existantes et des outils de preuve. Dans ce contexte, nous avons proposé une méthode générale d'imposition de politiques dans laquelle nous avons réalisé des exemples pertinents grâce à la coopération du système Tom avec de la programmation par aspects.

1.3 Conclusions et Perspectives

Les spécifications précises des politiques de sécurité sont d'une importance fondamentale pour les buts globaux de sécurité d'un système donné. Comme les conditions de sécurité pour les systèmes d'information courants impliquent de plus en plus de contraintes pour définir les états non-sûrs d'une application, il est nécessaire que les formalismes de spécification de politiques fournissent de l'expressivité et des outils pour la vérification formelle de politique.

Dans cette thèse nous avons présenté un modèle formel de politiques de sécurité dont la sémantique est fondée sur la réécriture de termes avec des stratégies. Nous avons montré que ce paradigme fournit un puissant langage basée sur des règles pour exprimer et raisonner au sujet des propriétés des politiques de sécurité et de leur composition. En utilisant notre cadre, nous pouvons définir une grande classe de politiques comme nous avons vu dans le monttrent les nombreux exemples présentés au long de ce manuscrit.

Sommaire des Contributions

Nous récapitulons les contributions principales réalisées pendant le développement de cette thèse.

Spécification de Politiques de Sécurité Le principal avantage de notre cadre pour la spécification de politiques est de fournir l'expressivité et l'agilité offertes par les stratégies de réécriture. Puisque les stratégies nous permettent d'avoir un contrôle fin sur l'application des règles, nous pouvons facilement modéliser des politiques réalistes, avec des priorités entre les règles et des règles par défaut.

L'originalité de l'approche se fonde également sur la liberté d'indiquer tous les éléments d'une politique — des décisions, des requêtes d'accès et des règles — et sur la capa-

cité d'exécuter le raisonnement formel au sujet du contrôle d'accès. Par l'association des propriétés du système de réécriture sous-jacent, nous pouvons vérifier la cohérence, la complétude et la terminaison d'une politique.

Bien qu'examiner ces propriétés pour la plupart des exemples que nous nous présentons dans la thèse n'exigent pas des techniques très avancées, ces propriétés sont indécidables en général. Nous avons alors identifié des conditions nécessaires où ces propriétés peuvent être vérifiées automatiquement. Ceci est crucial pour améliorer la confiance en un modèle formel de politique, par contre des limitations syntaxiques doivent être respectées.

Vérification de Politiques Nous avons donné des procédures basées sur la réécriture pour détecter des fuites d'information dans des politiques de contrôle d'accès obligatoires et pour le problème de la sûreté dans le modèle de contrôle d'accès HRU. Nous montrons dans cette thèse que les différentes réalisations d'une politique de sécurité multi-niveaux peuvent laisser des flux interdits d'information se produire. En outre, nous avons proposé une approche pour vérifier la sûreté de contrôle dans le modèle classique HRU qui permet de vérifier la sûreté en temps d'exécution. D'ailleurs, les techniques présentées dans cette thèse peuvent être adaptées pour vérifier d'autres modèles de contrôle d'accès.

Composition de Politiques Nous montrons dans cette thèse que la réécriture stratégique fournit un modèle formel, ainsi que l'expressivité nécessaire pour un cadre de composition de politiques de sécurité. La sémantique opérationnelle du langage de stratégies nous permet de combiner des ensembles de décisions, d'accorder des priorités aux règles (et aux ensembles de règles) et d'interférer également sur le flux d'information entre les politiques, par un ordonnancement des demandes d'accès parmi des politiques. Nous avons codé plusieurs approches existantes en utilisant de règles de réécriture grâce à l'expressivité fournie par notre langage de stratégies. Par exemple, nous avons montré comment coder des opérateurs de résolution de conflits, et nous avons traduit l'ensemble complet des combinateurs de XACML dans notre cadre.

Un autre avantage de notre cadre est que nous pouvons exécuter du raisonnement formel pour la composition de politiques en présence d'une multiplicité de décisions d'accès, incluant les politiques assignant explicitement des décisions neutres pour une certaine demande d'accès, ce qui ne peut pas être facilement capturé dans d'autres travaux.

Nous avons également montré l'intérêt de la réécriture stratégique dans la conservation de certaines propriétés de politiques comme la cohérence, la terminaison et sur les extensions conservatrices.

Imposition de Politiques Nous avons présenté une méthodologie pour construire des moniteurs de référence pour des politiques de contrôle d'accès basées sur des règles. Nous avons utilisé la programmation orientée par aspects pour capturer de façon appropriée l'environnement des politiques et pour transformer le code non fiable de façon à satisfaire une politique donnée. Notre méthodologie est assez générale pour être appliquée sur de divers exemples. Nous avons montré la faisabilité de l'approche sur l'application des politiques basée sur la réécriture des programmes Java, en reliant le système Tom et le langage orientée par aspects AspectJ.

Perspectives Futures

Puisque notre approche est récente, il y a un certain nombre de concepts associés aux politiques et au contrôle d'accès pour lequel nous n'offrons pas explicitement de support. Citons par exemple les obligations [Dougherty et al., 2007a], qui représentent le compromis d'un certain agent à remplir une certaine contrainte, ou à effectuer une certaine action à l'avenir. Nous ne traitons pas explicitement des issues administratives dans les politiques [Sandhu et al., 1999, Barker and Fernández, 2006, Cuppens and Miège, 2004]. Ceci implique l'existence d'une entité ayant le privilège de modifier ou de questionner la politique de sécurité en place.

Nous n'abordons pas non plus la négociation de confiance non plus. Dans les négociations de confiance [Bonatti and Olmedilla, 2005, Yu et al., 2001] des entités autonomes doivent convenir, en utilisant un certain protocole, dans une politique concernant combien d'information elles doivent révéler mutuellement afin de réaliser un certain but commun. Ces sujets, entre autres, sont des thèmes intéressants pour des études ultérieurs.

En ce qui concerne les spécifications de politiques, une direction prometteuse de recherche est l'impact que les anti-patterns [Kirchner et al., 2007] peuvent apporter aux spécifications de politiques de sécurité. Un anti-pattern explicite ce qu'on ne veut pas filtrer sur un terme. Considérer que les règles peuvent contenir des anti-patterns aurait comme conséquence des spécifications encore plus concises.

Par rapport à l'analyse de politiques, il semble intéressant de étudier la perspective offerte par le *narrowing*. Une telle technique peut fournir un moyen d'analyse statique pour des politiques basées sur des règles. Par exemple, on pourra savoir quelles sont les instantiations possibles des variables dans un terme donné qui peuvent mener à une permission.

Une direction importante de recherches liée à l'imposition de politiques est de considérer une approche automatique pour imposer des politiques flexibles basées sur la réécriture. Pour cela, il est nécessaire de développer des techniques sophistiquées pour la vérification de types et des algorithmes d'inférence de types. En effet dans les langages ayant des systèmes types polymorphes et de la généricité tels que Java, des actions sensibles peuvent avoir plusieurs formes, rendant difficile leur détection par un moniteur de référence. Dès lors, les auteurs des politiques ne devraient pas être obligés d'indiquer manuellement toutes ces formes.

Plusieurs questions demeurent ouvertes en ce qui concerne les propriétés des systèmes de réécriture sous stratégies. Par exemple, nous ne nous connaissons pas de résultats sur la complétude suffisante d'un système de réécriture sous une stratégie définie par l'utilisateur. Nous ne nous connaissons pas de conditions nécessaires pour la modularité de la propriété de complétude suffisante non plus. Pour le moment, nous n'avons pas d'algorithmes pour vérifier la confluence sous stratégies. De tels développements seraient non seulement intéressants pour les spécifications et la composition de politiques de sécurité, mais également pour d'autres domaines utilisant intensivement de la réécriture stratégique, tels que la transformation de programmes.

1.4 Plan de la Thèse

Le restant du manuscrit, en langue anglaise, est organisé de la façon suivante :

- Le chapitre 3 présente les notions préliminaires sur la réécriture de termes et ses extensions, comme la réécriture conditionnelle et la réécriture modulo des théories équationnelles. Puis, ce chapitre présente la définition de stratégies abstraites, et aussi l'ensemble d'opérateurs de stratégies que nous adoptons dans le reste de la thèse.
- Le chapitre 4 présente la définition formelle des politiques basées sur la réécriture. Nous motivons, définissons et discutons les propriétés des politiques avec plusieurs exemples. Puisque ces propriétés sont indécidables en général, nous indiquons les classes restreintes de systèmes de réécriture pour lesquelles on peut fournir une procédure pour les tester.
- Le chapitre 5 établit des techniques pour la vérification des propriétés de politiques des sécurité : la vérification du flux de l'information dans les politiques de sécurité multi-niveaux, le problème de la sûreté dans certains modèles contrôle d'accès. Dans les deux cas, nous fournissons des procédures en utilisant des techniques de réécriture.
- Le chapitre 6 définit la composition de politiques de sécurité via la réécriture stratégique. Il étend la définition des politiques basées sur la réécriture en fournissant la possibilité de définir la sémantique formelle de plusieurs genres de combinateurs de politiques. Nous discutons également la façon dont les propriétés des politiques combinées peuvent être préservées, en basant sur des résultats théoriques qui portent sur les propriétés modulaires en réécriture.
- Le chapitre 7 traite le problème de l'imposition des politiques expressives décrites en utilisant le cadre basé sur la réécriture du Chapitre 4. Il propose une méthode pour inclure des politiques dans des applications existantes, par la combinaison du système Tom et de la programmation orientée aspects. Ceci fournit les outils nécessaires pour le développement non invasif de politiques de sécurité, où les politiques peuvent être développées indépendamment des programmes où elles fonctionnent.
- En conclusion, le Chapitre 8 présente des perspectives des travaux futurs. Les travaux reliés aux développements décrits dans cette thèse sont discutés dans chaque chapitre indépendamment, avec une exposition concentrée sur le sujet correspondant.

Chapter 2

Introduction

2.1 Motivation

It is common sense that we live in the information age, and that the world's economy, communications, and entertainment depend on computers. Inestimably valuable information for many agents (companies, governments, etc) circulate on large information systems, often connected to other systems by networks. In this context, a primordial concern addressed by security policies is the protection of the data being processed from any undesirable use. According to [Bishop, 2004], a *security policy* is a statement of what is, and what is not, allowed. Consequently, policies are extremely important artifacts because the security of the huge amount of information handled by computers (or more abstractly, *resources*), depend on the clear statement of these security requirements.

Security requirements determine the situations in which a system may be exposed : (i) secret information may be revealed to agents who are not supposed to obtain it; (ii) the information itself can be corrupted, or modified in an unreliable manner, such that it cannot be trusted anymore, or even destroyed; and (iii) the information may be retained such that it becomes unavailable for other uses during some crucial period of time. These issues are known as *confidentiality*, *integrity* and *availability*, respectively. A security policy usually focuses on one of these issues, but the security goals in a given real world configuration may have an arbitrary combination of requirements involving these three aspects.

Several problems are related to this topic of policy specification. First, in order to cope with elaborated security requirements of current applications and systems, policies are described as computations towards a decision whether a given situation in a system is authorized. In a broad sense, given the current state of a system, a policy evaluates the data it contains in order to indicate which are possible secure transitions from this state. Therefore, policy definitions can be arbitrarily complex, and this demands an appropriate representation – they should be understood by people, but also be suitable to automated analysis. Automated policy analysis is highly desirable: as policies reflect real world constraints on how information can be accessed, thus being subject to frequent changes, it is very useful to dispose of tools to verify that a policy specification corresponds to the security requirements.

An example of property that policies should satisfy is to be unambiguous. The sharp distinction between allowed and prohibited states is hard to achieve. Due to the amount and diversity of data a policy has to evaluate, the security conditions are often partially specified. This happens because policies represent the interest of some agents in the system, that may be in contradiction with the interests of other agents. For an example, consider a medical system where there is

a policy regulating access to patient data. Health care providers and insurance companies are surely going to have competing requirements on which information can be revealed to whom.

It is natural to imagine that policies should not be authored and maintained in a single monolithic specification. Policies are independently created by different entities, which put forward their own priorities when defining the security goals. Assuming that large systems are used by several entities, a global policy must accommodate the different emphasis of each entity on the security requirements, that can even be stated using different languages.

Since information systems need to be integrated with each other, policy composition is a crucial problem in contemporary computer security. Policies specifications must coherently support different ways of defining a compound policy to govern different parts of a system by means of component policies.

The size and complexity of policies being treated mean that the policies themselves are interesting software artifacts in their own right. Ideally, policies are developed, analyzed, and maintained separately from the information systems where they run, with support of specialized tools. Unfortunately, a large part of the software being engineered today still defines policies in an ad-hoc manner. This creates room for compromising software faults. Policies are coded by hand inside the programs, making it difficult to understand how authorizations are computed.

As we will discuss in the next section, several approaches have been proposed for policy specification, and for the construction of mechanisms to impose them in computer systems. However, the problems we mentioned above remain to be solved, as more factors that influence how access is conceeded need to be taken into account, in the context of policy-based security nowadays.

2.2 Historical Background

Access control policies are a particular way to implement a security policy. It involves three main elements: the *resources* to be protected, the *actions* (or *access modes*, or *access privileges*) that can be performed over these resources, and the *subjects* (users or programs) in the system that may demand access over the existing resources.

In [Lampson, 1974], these elements are organized in an access control matrix, where cells contain a set of privileges. This data structure was reused in several systems, and has been improved to facilitate mediation for large collections of users and resources via access control lists, or by assigning privileges to *roles*, that can be assumed by an user for performing some tasks.

Since long time researchers in computer security have identified the need for mathematical rigor in policy specification. A rigorous specification of the security policy was called initially in the computer security literature a *security model*. The first models reflect the fact that computers were still very expensive in the late 70's, and that they were likely to be shared by several users.

In [Bell and LaPadula, 1974], the authors also introduced a formal model (known as the BLP model, or simply BLP for short) to capture the needs of computerized military systems, which already had clearly specified security goals proved effective when only pencil and paper were used to communicate information. In their seminal work, Bell and LaPadula associated security labels to users and objects in the system. These labels are ordered according to a classification,

such that users with a lower clearance should not have the right to read information from a higher security class. Such conditions specify how information is allowed to flow in the system, and are often called *information flow* policies. Another precursor work appeared in [Biba, 1977] for dealing with integrity constraints. It can be understood as a dual of BLP, in the sense that integrity levels replace the security levels of BLP, and the resulting model defines that information cannot flow from a subject to an object whose integrity level is higher.

In [Goguen and Meseguer, 1982], the authors introduced the notion of non-interference policies, which intuitively says that the actions performed by some user of a system, cannot be inferred by another user by observing some external behavior of the system. Goguen and Meseguer elaborated an abstract formalization of non-interference in terms of the possible commands that can be executed in a system by a group of users, which can be seen as the way information enters the system, and on the visible outputs of the system.

These works are of great importance for computer security because they introduced principles for the formal specification of policies: the security properties must be correctly stated and proved. The problem is that the early security models (mainly for the mandatory ones) were too rigid to be used outside the military domain.

The research on security policies moved towards policy specification formalisms where it is possible to define policies with an underlying formal semantics, in a per-application basis. Such frameworks, in principle, allows one to perform reasoning about the security properties from an abstract point of view. This motivated rule-based approaches for policies, which we discuss in the following.

Rule-based approaches

Policies are expressed by rules in natural language: if some conditions are satisfied, then, grant or deny some request. Rule-based languages are well-suited for policy specification because they have an understandable, human-friendly syntax, and are based on a formal semantics; indeed, rules are a natural way of defining policies. Not surprisingly, rule-based policy specification is a major research domain in computer security.

There are a number of initiatives that have applied different rule-based formalisms for policies. The existing approaches adopt different semantic frameworks such as first-order logic [Halpern and Weissman, 2003, Jajodia et al., 2001, Becker and Sewell, 2004b], or Datalog [Bonatti and Samarati, 2002, Li and Mitchell, 2003, Bonatti and Olmedilla, 2005, Dougherty et al., 2006], temporal logic [Bertino et al., 1998, Cuppens et al., 2005], deontic logic [Cholvy and Cuppens, 1997, Kalam et al., 2003], and defeasible logic [Lee et al., 2006].

The common characteristic of these initiatives is that one disposes of different degrees of expressivity for authoring a policy. This expressivity allows one to finely specify the conditions under which access takes place. In access control, these conditions involve attributes related to subjects or resources. For instance, in a hospital system the policy must take into account whether the subject is a physician, or administrative staff, or whether the resource being consulted is patient related data, whose access is restricted to the physician in charge for this person. As a further example, such a policy should contain rules to govern access to patient data in case of an emergency, while other physicians should also be allowed to read a medical record of a patient, even they are not primarily responsible for them.

It is worth to mention alternative approaches for policy specification that also address authoring, which are declarative, but not exactly rule-based: policies are small programs that compute authorizations, with some accompanying facilities for policy reuse. Representative examples following this direction are Ponder [Damianou et al., 2001], Polymer [Bauer et al., 2005], and the Java Security API [Gong et al., 2003]. This last one also has support to rather "programmatic" policies which may use imperative instructions to define policies.

Originally, these formalisms for policies allow only for *closed* policies, which strictly allowed situations, or *open* policies which specify the cases to be denied, while all other unspecified requests are allowed. However, there is a lot of need for specifying explicitly both positive and negative rules, simply because in the case of closed or open policies it becomes quickly restrictive when there are numerous punctual exceptions for some rules. In such cases, policy specification becomes a really tedious activity.

Closed (resp. open) policies contain default rules that allow (resp. deny) a large class or requests not explicitly handled by the policy. This brings an extra difficulty to compose such policies: when the component policies overlap, it may be tricky to assume a default decision for some conflicting request. A way to provide more flexibility for composition is to assume that for some cases, a policy is allowed not to give a decision for some kind of access requests. Then, the policy can be coherently overridden in composing with another policy.

The need for multiple decisions is even greater than it seems, notably for policy composition. Such feature appears in the design of several recent policy frameworks [Moses, 2005, Bauer et al., 2005, Bruns et al., 2007].

We further detail classic (confidentiality) models in the beginning of Chapter 4, and present a more detailed comparison among the rule-based approaches in the related works part of the same chapter, Section 4.6. There are interesting surveys covering the topics briefly introduced in the previous paragraphs. The reader should refer to [di Vimercati et al., 2005, Ryan, 2000, Antoniou et al., 2007] for more information on security policies, models and rule-based initiatives.

2.3 Contributions

In this thesis, we tackle several issues related to expressive, rule-based policy specification and composition. The first main contribution is a general definition of security policies that is able to capture a broad range of applications. One of its novelties is to adopt term rewriting as the foundation of the formal policy framework. The first approaches for access control using term rewriting were introduced in [de Oliveira, 2007], independently of [Barker and Fernández, 2006]. The formalisation we adopt in this thesis, which appeared in [Dougherty et al., 2007b], is a revised and augumented version of the work in [de Oliveira, 2007] for incorporating the notion of rewriting strategies [Kirchner et al., 2008, Kirchner, 2005, Borovansky et al., 2001, Cirstea et al., 2003] in policy definitions. In the next chapters, we include examples of access control policies in general information systems (e.g a medical system, a conference management system, etc), firewall policies, multilevel security, role-based access control, etc.

Term rewriting is a very active research field in automated reasoning. It is a Turing complete computational model, applied mainly in theorem provers, constraint solvers, and

in the foundations of functional programming languages. Term rewriting has an associate logic, the rewriting logic, surveyed in [Martí-Oliet and Meseguer, 2002] and calculus, the ρ-calculus [Cirstea and Kirchner, 2001]. Both provide a deep understanding of the core mechanisms of rewriting: its reflexive properties, matching power, and rewrite strategies. The use of term rewriting for security policies is justified by its high expressivity, by the numerous available proof techniques for checking properties, and by the efficiency of the various available implementations of rewrite engines. In this thesis we take benefit from all these advantages to provide a solid formal framework for policies.

The second most important contribution lies on the field of rule-based policy composition. Policies in our formal framework may return multiple possible access decisions, then we provide a theory of policy composition that fits adequately to existing policy languages such as the industrial standard XACML [Moses, 2005]. In contrast to other approaches, it is able to deal with several concrete composition "scenarios" without introducing any contradiction to the underlying logic. Furthermore, our policy composition framework is extensible: one can define new forms of policy combiners, by handling explicitly how rules should interact, or by providing rules to disambiguate among possibly conflicting decisions produced by the sub-policies. This work is the core contribution presented in [Dougherty et al., 2007b].

In this context rewriting strategies are used for two main purposes: to control how the rewrite rules in a policy must be applied in order to derive an access decision, and to give the semantics of policy combining algorithms. The choice of a given rewriting strategy may also facilitate the preservation of individual policy properties, in the context of a compound policy.

Besides policy specification and composition, we also present an analysis algorithm for verifying information flow in multilevel access control systems. The algorithm was written in Tom [1] [Balland et al., 2007], an implementation of rewriting systems embedded in Java. This work appeared in [Morisset and de Oliveira, 2007]. Moreover, we also develop an algorithm for checking safety in access control models, by encoding them as rewrite-based policies and testing whether the reachable states of the model are safe.

The Tom language and system also made it possible for us to address the construction of mechanisms for policy enforcement. A design principle in access control is that policy specification and the mechanism to enforce it should be separated. This coincides with the concept of *formal islands* [Balland et al., 2006b, Balland et al., 2006c], which sets the foundations for embedding formally specified languages in general purpose programming environments, principles that guided the construction of the Tom system itself. We show how Aspect Oriented Programming can be used to achieve this separation as we deploy rewrite-based policies implemented in Tom over Java programs. We provide a methodology to help developers to construct reliable access control mechanisms for existing applications such that policy can be maitained and updated without need to modify the program code by hand. This work appeared in [de Oliveira et al., 2007].

[1] http://tom.loria.fr/

2.4 Overview of the Thesis

This thesis is organized as follows:

- Chapter 3 presents the preliminary notions on term rewriting and its extensions, such as conditional rewriting and rewriting modulo equational theories. Then, it introduces the definition of abstract strategies, and presents the strategy language we adopt in the rest of the thesis.
- Chapter 4 introduces the formal definition of rewrite-based policies. We motivate, define, and discuss policy properties through several examples. Since these properties are undecidable in general, we indicate restricted classes of rewrite system for which they become decidable.
- Chapter 5 builds techniques for the verification of more elaborate policy properties, such as checking information flow in multilevel security, as well as verifying safety in access control models. In both cases, we provide procedures using rewriting techniques.
- Chapter 6 defines policy composition through strategic rewriting. It extends the definition of rewrite-based policies providing the possibility to define the formal semantics for several kinds of policy combiners. In this setting, we also discuss how the properties of the policies being combined can be preserved, based on results on modular properties from the rewriting theory.
- Chapter 7 addresses the problem of enforcing expressive policies using the rewrite-based framework of Chapter 4. It proposes a method for embedding policies in existing applications, by combining the Tom system and aspect oriented programming. This provides the necessary tools for the non-invasive policy development, where policies can be developed independently of the programs where they run.
- Finally, Chapter 8 concludes with perspectives for future work. Related works are discussed in each chapter, with an exposition focused on their corresponding topics.

Chapter 3

Background on Rewriting and Strategies

In this chapter we introduce the necessary formal background concerning term rewriting and strategies.

3.1 Preliminary Notions

3.1.1 Term Algebra

This section contains the basic notions on first-order term algebra.

Signatures A many-sorted signature is a set of sorts $\mathcal{S}$ and a set of function symbols $\mathcal{F}$. Each $f \in \mathcal{F}$ has a profile $f : S_1 \times \ldots \times S_n \rightarrow S$, where $S_1, \ldots S_n, S \in \mathcal{S}$, and is associated to a natural number by the arity function (ar : $\mathcal{F} \rightarrow \mathbb{N}$). When $ar(f) = 0$, the function symbol f is called a constant. For defining a signature we enumerate the function symbols it contains with their respective profiles.

Terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of well-sorted terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. Variables are also sorted, and $x : S$ means that the variable x in $\mathcal{X}$ has sort S. The set $\mathcal{X}_S$ denotes a set of variables of sort S, and $\mathcal{X} = \bigcup_{S \in \mathcal{S}} X_S$. Terms are classified according to their sorts as follows.

Definition 8. *The set of terms of sort S, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})_S$ is the smallest set containing $\mathcal{X}_S$ and any constant $a : S$ such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})_S$ whenever $f : S_1 \times \ldots \times S_n \rightarrow S$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})_{S_i}$ for $1 \leq i \leq n$.*

The top symbol of a term is denoted $Head(t)$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* (we may also call *closed term*). $\mathcal{T}(\mathcal{F})$ is the set of all ground terms. For $f : S_1 \times \ldots \times S_n \rightarrow S \in \mathcal{F}$, $f(\mathcal{T}(\mathcal{F}_{S_1}), \ldots, \mathcal{T}(\mathcal{F}_{S_n}))$ denotes the set of ground terms with f as top symbol. We may ommit sort names when they are clear from the context. A term t is said to be *linear* if each variable in t occurs at most once.

Positions Let $\mathbb{N}$ be the set of natural numbers, $\mathbb{N}_+$ the set of non-zero naturals. The set of finite sequences of non-zero natural numbers $\mathbb{N}_+^*$ is defined as:

$$p = \epsilon | n | p.p$$

where $\epsilon \in \mathbb{N}_+^*$ represents the empty sequence and $n \in \mathbb{N}_+$. For all $p, q \in \mathbb{N}_+^*$, p is prefix of q if there is $r \in \mathbb{N}_+^*$ such that $q = p.r$.

Definition 9 (Position)**.** *The set of positions $\mathcal{P}os(t)$ of the term t is recursively defined as follows:*

- *$\epsilon \in \mathcal{P}os(t)$ is the head position of t.*
- *For all $p \in \mathcal{P}os(t)$ and all $i \in \mathbb{N}_+^*$, $p.i \in \mathcal{P}os(t)$ if and only if $1 \leq i \leq ar(f)$ where $f \in \mathcal{F}$ is the symbol at the position p of t.*

We call subterm of t at the position $p \in \mathcal{P}os(t)$, the term denoted $t_{|p}$ which satisfies the following condition:

$$\forall p.r \in \mathcal{P}os(t), r \in \mathcal{P}os(t_{|p}) : Head(t_{|p.r}) = Head((t_{|p})_{|r})$$

We denote $t[s]_p$ the term t where the subterm at the position p has been replaced by the term s.

Example 4. *The set of Peano integers can be described by the signature containing the function symbols $\mathcal{F} = \{s : Nat \rightarrow Nat, 0 : Nat, plus : Nat \times Nat \rightarrow Nat\}$. The set of positions of the term $t = plus(s(0), s(s(0)))$ is $\mathcal{P}os(t) = \{\epsilon, 1, 2, 1.1, 2.1, 2.1.1\}$ which corresponds respectively to the subterms $plus(s(0), s(s(0))), s(0), s(s(0)), 0, s(0)$ and 0.*

The *height* of a term t is 0 if it is a single variable or a constant, and $1 + max_{i \in \{1,\ldots,n\}}(height(s_i))$ if $t = f(s_1, \ldots, s_n)$. The *depth* at position p of a term t in a term s, *i.e.* $s = s[t]_p$ is the maximum length of the sequence p. A term t is said *shallow* if all variable positions in t are at depth 0 or 1. A term is *flat* if its height is at most 1.

Substitutions A *substitution* σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, with a finite domain $\{x_1, \ldots, x_k\}$ and written $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. For all $i \in \{1, \ldots, k\}$, x_i, t_i are of the same sort. Substitutions are denoted by lower-case Greek letters such as σ, μ, etc.

The application of a substitution σ to a term t, denoted $\sigma(t)$, or σt, simultaneously replaces all occurrences of variables by their respective σ-images. The composition of σ followed by μ is denoted $\sigma\mu$ (thus $(\sigma\mu)(t) = \sigma(\mu(t))$ for any term t). We say that σ instantiates x if $x \in \mathrm{Dom}(\sigma)$. A term t is called an instance of a term s iff there exists a substitution σ such that $t = \sigma(s)$

Definition 10. *A substitution σ is* more general *than a substitution σ' if there is a substitution δ such that $\sigma' = \delta\sigma$. In this case we write $\sigma \lesssim \sigma'$. We also say that σ' is an instance of σ.*

Example 5. *We consider again the Peano integers. We define a set of variables $\mathcal{X} = \{x, y\}$. Given the term $t = plus(s(x), s(y))$ and the substitution $\sigma = \{x \mapsto 0, y \mapsto s(0)\}$, we have: $\sigma(t) = plus(s(0), s(s(0)))$.*

Two fundamental operations on terms are matching and unification.

Definition 11 (Pattern Matching)**.** *A term t* matches *a term s, or s is an* instance *of t, if there is a substitution σ such that $s = \sigma(t)$.*

This matching problem is known as *syntactical matching*, which is distinct from matching modulo an equational theory, defined in Section 3.1.2. Syntactical matching is always decidable. It is linear on the size of the pattern, if this last one is a linear term. Otherwise, matching is linear on the size of the term being matched (the term s in the definition above).

Definition 12 (Unification)**.** *An equation is a formula of the form $s =^? t$ where s and t are terms. An unification problem is a conjunction of equations denoted $P = (s_1 =^? t_1 \wedge \ldots s_n =^? t_n)$.*

- Two terms s and t are *unifiable* if there is a substitution σ such that $\sigma(s) = \sigma(t)$. σ is called an *unifier*.
- A substitution σ is a solution for P if $\sigma s_i = \sigma t_i$ for $i = 1, \ldots, n$. The set of all unifiers of an unification problem P is denoted $\mathcal{U}(P)$.

Definition 13. *A substitution σ is a* most general unifier*(m.g.u) of P, if σ is a least element of $\mathcal{U}(P)$, i.e. for every solution $\sigma' \in \mathcal{U}(P)$, $\sigma \lesssim \sigma'$.*

This unification problem is also called *syntactic unification*. Syntactic unification is also decidable. Its complexity is linear if terms are represented by an appropriate structure such as *directed acyclic graphs* [Paterson and Wegman, 1978].

3.1.2 Equational Theories

Definition 14 (Equality or Axiom)**.** *An equality (axiom) is a pair of terms $\langle l, r \rangle$, denoted $l = r$, where l and r are terms of the same sort.*

A binary relation $\longrightarrow$ over terms is said

- *stable by context* if for all terms s, t, u and for all position $p \in \mathcal{P}os(u)$: if $s \longrightarrow t$ then $u[s]_p \longrightarrow u[t]_p$.
- *stable by instantiation* if for all terms s, t and all substitution σ: if $s \longrightarrow t$ then $\sigma(s) \longrightarrow \sigma(t)$.
- *compatible* if it has both properties above.

Definition 15. *Given a set of axioms $\mathcal{E}$, we denote $\longleftrightarrow_{\mathcal{E}}$ the symmetric binary relation over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ defined by $s \longleftrightarrow_{\mathcal{E}} t$ if there is an axiom $(l = r)$ in $\mathcal{E}$, a position p in s and a substitution σ such that $s|_p = \sigma(l)$ and $t = s[\sigma(r)]_p$.*

The reflexive and transitive closure of $\longleftrightarrow_{\mathcal{E}}$, denoted $\overset{*}{\longleftrightarrow}_{\mathcal{E}}$, is the equational theory generated by *$\mathcal{E}$, or briefly, the equational theory $\mathcal{E}$.*

This definition formalizes the "replacement of equals by equals", Assuming the classical definition of validity (which can be found for instance in [Baader and Nipkow, 1998]), we mention an important theorem, due to Birkhoff, that guarantees the correctness and completeness of equality proofs by replacements of equal by equal.

Theorem 1. *[Birkhoff, 1935] $\mathcal{E} \models s = t$ if and only if $s \overset{*}{\longleftrightarrow}_{\mathcal{E}} t$.*

Short Name	Property	Definition
A_f	Associativity	$f(f(x,y),z) = f(x,f(y,z))$
C_f	Commutativity	$f(x,y) = f(y,z)$
I_f	Idempotency	$f(x,x) = x$
$E_{(h,*)}$	Endomorphism	$h(x*y) = h(x)*h(y)$
$H_{(h,*,+)}$	Homomorphism	$h(x*y) = h(x)+h(y)$
$T_{(f,g)}$	Transitivity	$f(g(x,y),g(y,z)) = f(g(x,y),g(x,z))$
$Ur_{(*,e)}$	Right Unit	$x*e = x$
$Ul_{(*,e)}$	Left Unit	$e*x = x$
$U_{(*,e)}$	Unit	$Ur_{(*,e)} \cup Ul_{(*,e)}$

Table 3.1: Some equational axioms

Some theories we mention in this thesis are defined in Table 3.1. In addition, an equational theory $\mathcal{E}$ is called a *permutative theory* if for every equation $s =_\mathcal{E} t$, the number of occurrences of every symbol in s is the same as in t.

Deciding whether two arbitrary terms are equal in an equational theory is known as the *word problem* in this theory.

The notions of unification and matching can be generalized to take into account the fact that terms can be equal *modulo* a given equational theory.

Definition 16 (Matching and Unification modulo an equational theory)**.** *Let $\mathcal{E}$ be an equational theory. We say that*

- *A term t* matches modulo *$\mathcal{E}$ a term s if there exists a substitution σ such that $s \overset{*}{\longleftrightarrow}_\mathcal{E} \sigma(t)$.*
- *two terms s, t are* unifiable modulo *$\mathcal{E}$ if there exists a substitution σ such that $\sigma(s) \overset{*}{\longleftrightarrow}_\mathcal{E} \sigma(t)$.*

In contrast to syntactical matching and unification problems, matching and unification problems modulo an equational theory are undecidable in general. When they can be decided, the available algorithms may have a considerable complexity. Well-known examples are matching and unification modulo associativity and commutativity. The interested reader can refer to [Baader and Snyder, 2001] for a detailed presentation of these issues.

3.1.3 Term Orderings

Definition 17. *A binary relation $\succeq$ defined on a set A is a* partial order *on the set A if the following conditions hold for all elements of A:*

1. *$a \succeq a$ (reflexivity)*
2. *$a \succeq b$ and $b \succeq a$ imply $a = b$ (antisymmetry)*
3. *$a \succeq b$ and $b \succeq c$ imply $a \succeq c$ (transitivity)*

   If, in addition, for every a, b in A

4. $a \succeq b$ *or* $b \succeq a$

   then $\succeq$ *is a* total order *on A.*

A nonempty set with a partial order is called a *partially ordered set*, or more briefly a *poset*. If the order is a total order then we speak of a *totally ordered set*. In a poset A, $a \succ b$ if $a \succeq b$ and $a \neq b$ (called a *strict order*).

Lattices

We give the definition of lattice, which is useful for understanding some security models.

Definition 18. *Let A be a subset of poset P. An element p in P is an* upper bound *for A if $p \succeq a$ for every a in A.*

- *An element p in P is a the* least upper bound *of A (l.u.b of A), or* supremun *of A (sup A) if p is an upper bound of A, and $b \succeq a$ for every a in A implies $b \succeq p$ (i.e., p is the smallest among the upper bounds of A).*
- *Similarly we can define what it means for p to be a* lower bound *of A, and for p to be the* greatest lower bound *of A (g.l.b of A), also called* infimun *of A (inf A).*

Definition 19. *A poset L is a* lattice *if for every a, b in L both $sup\{a, b\}$ and $inf\{a, b\}$ exist in L.*

If L is totally ordered set, we speak of a particular lattice structure called a *chain*.

Definition 20 (Well-founded Order)**.** *An order $\succeq$ over the set E is* well-founded *if there is no infinite sequence $x_1 \succeq x_2 \succeq \ldots \succeq x_n \succeq \ldots$ where $x_i \in E$ ($i \in \mathbb{N}_+$).*

Definition 21 (Rewrite order)**.** *A* rewrite order *is a compatible order over the set of terms.*

Definition 22. *A* simplification order *is a rewrite order which contains the strict subterm relation. A* reduction order *is a well-founded rewrite order.*

Definition 23 (*Lexicographic Path Ordering*)**.** *Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ be a term algebra and $\succ$ a strict ordering over symbols $\mathcal{F}$ (called a precedence relation), the lexicographic path order which extends $\succ$, denoted $\succ_{lpo}$, is defined as :*

$$s \succ_{lpo} t$$

if one of the following conditions is verified:

1. $t \in \mathcal{V}ar(s)$ *and* $s \neq t$
2. $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$ *and one of the following conditions are true:*
   - *there is i, $1 \leq i \leq m$ such that $s_i \succ_{lpo} t$*
   - *$f \succ g$ and $s \succ_{lpo} t_j$ for all j such that $1 \leq j \leq n$*
   - *$f = g$, $s \succ_{lpo} t_j$ for all j such that $1 \leq j \leq n$ and there is i, $1 \leq i \leq m$ such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i \succ_{lpo} t_i$*

For finite signatures, a lexicographic path order is also a simplification order over terms [Ohlebusch, 2002a].

3.1.4 Abstract Reduction Systems

In order to correctly abstract the notion of strategy, it is necessary to adopt an appropriate definition for abstract reduction systems. The one adopted in [Kirchner et al., 2008] allows one to make the distinction between "being in relation with" and "being connected to", *e.g.* this allows one to describe different ways to derive a term t' from a term t.

Definition 24. *An abstract reduction system (ARS) is a labeled binary oriented graph $(\mathcal{O}, \mathcal{S})$. The nodes in $\mathcal{O}$ are called objects, the oriented edges in $\mathcal{S}$ are called steps.*

Definition 25 (Derivation)**.** *For a given ARS $\mathcal{A}$:*

1. *A* reduction step is a labeled edge ϕ together with its source a and target b. This is written $a \rightarrowtriangle^{\phi}_{\mathcal{A}} b$, or simply $a \rightarrowtriangle^{\phi} b$ when unambiguous.
2. An *$\mathcal{A}$-derivation or* $\mathcal{A}$-reduction sequence is a path π in the graph $\mathcal{A}$.
3. When it is finite, an $\mathcal{A}$-derivation π can be written $a_0 \rightarrowtriangle^{\phi_0} a_1 \rightarrowtriangle^{\phi_1} a_2 \ldots \rightarrowtriangle^{\phi_{n-1}} a_n$ and we say that a_0 reduces to a_n by the derivation $\pi = \phi_0\phi_1 \ldots \phi_{n-1}$; this is also denoted $a_0 \rightarrowtriangle^{\phi_0\phi_1\ldots\phi_{n-1}} a_n$ or simply $a_0 \rightarrowtriangle^{\pi} a_n$. We call n the *length of* π.
   a) *The* source of π is the object a_0 and its domain is defined as the singleton $\mathrm{Dom}(\pi) = \{a_0\}$.
   b) The *target of π is the object a_n and the application of the derivation π to a_0 is the singleton denoted $([\pi](\ a_0)) = \{a_n\}$. This is also denoted simply $[\pi](a_0)$ when there is no syntactic ambiguity.*
4. *The set of all derivations is denoted $\mathcal{D}(\mathcal{A})$.*
5. *A derivation is* empty when its length is zero, in which case its source and target are the same. The empty derivation issued from a is denoted id_a.
6. The *concatenation of two derivations π_1 and π_2 is defined when Dom$(\pi_1) = \{a\}$ and $[\pi_1](a) =$ Dom(π_2). Then $\pi_1;\pi_2$ denotes the new $\mathcal{A}$-derivation $a \rightarrowtriangle^{\pi_1}_{\mathcal{A}} b \rightarrowtriangle^{\pi_2}_{\mathcal{A}} c$ and $[\pi_1;\pi_2](a) = [\pi_2]([\pi_1](a)) = \{c\}$.*

Note that an $\mathcal{A}$-derivation is the concatenation of its reduction steps. The following definitions concern general properties of abstract reduction systems.

Definition 26 (Termination)**.** *For a given ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$:*

- *$\mathcal{A}$ is terminating (or* strongly normalizing *if all its derivations are of finite length.*
- *An object $a \in \mathcal{O}$ is normalized when the empty derivation is the only one with source a (e.g. a is the source with no edge).*
- *A derivation is* normalizing *when its target is normalized.*
- *$\mathcal{A}$ is* weakly terminating *if every object a is the source of a normalizing derivation.*

Definition 27 (Confluence)**.** *An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is* confluent *if for all objects a, b, c in $\mathcal{O}$, and all $\mathcal{A}$-derivations $a \rightarrowtriangle^{\pi_1} b$ and $a \rightarrowtriangle^{\pi_2} c$, there exists d in $\mathcal{O}$ and two $\mathcal{A}$-derivations π_3 and π_4 such that $c \rightarrowtriangle^{\pi_3} d$ and $b \rightarrowtriangle^{\pi_4} d$.*

3.2 Term Rewriting

We first present "plain" term rewriting before we discuss several extensions such as conditional rewriting, rewriting modulo an equational theory, and rewriting with user-defined strategies.

Definition 28 (Rewrite rule)**.** *Given a signature and a set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$, a rewrite rule is a oriented pair of terms denoted $l \to r$ where $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$. We call l and r respectively right-hand side and left-hand side of the rule.*

These restrictions are usually imposed on rewrite rules of the form $l \to r$:

- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$,
- $l \notin \mathcal{X}$, and
- l and r are of the same sort.

Definition 29. *Let R be a rewrite system over $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $l \to r$ a rewrite rule in R. We say that $l \to r$ is a:*

- *left-linear rule (respectively right-linear rule) if l (respectively r) is a linear term,*
- *linear rule if both l and r are linear terms,*
- *ground rule if $l, r \in \mathcal{T}(\mathcal{F})$,*
- *right-ground if $r \in \mathcal{T}(\mathcal{F})$,*
- *left-shallow rule (respectively right-shallow rule) if l (respectively r) is a shallow term,*
- *shallow rule if both l and r are shallow terms,*
- *left-flat rule (respectively right-flat rule) if l (respectively r) is a flat term,*
- *flat rule if both l and r are flat terms,*
- *collapsing rule if r is a variable,*
- *duplicating rule if there exists a variable that has more occurrences in r than in l.*

Definition 30 (Rewrite Relation)**.** *Given a signature $\mathcal{T}(\mathcal{F},\mathcal{X})$ and a rewrite system R over $\mathcal{T}(\mathcal{F},\mathcal{X})$. The rewrite relation associated to R over $\mathcal{T}(\mathcal{F},\mathcal{X})$ is denoted $\to_R$ and is defined as follows: $t \to_R s$ if there exists a position p in t, a rewrite rule $l \to r$ in R and a substitution σ such that $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The subterm $t|_p$ is an instance of the left-hand side l and it is called a* redex.

Example 6. *The operator $plus$ for Peano integers can be defined by the following rewrite system :*

$$R = \left\{ \begin{array}{lll} [r_1] & plus(0,y) & \to \quad y \\ [r_2] & plus(s(x),y) & \to \quad s(plus(x,y)) \end{array} \right.$$

The term $t = plus(s(0), s(s(0)))$ is normalized by the following derivation:

$$plus(s(0), s(s(0))) \to s(plus(0, s(s(0)))) \to s(s(s(0)))$$

A term s is said R-irreducible, or *irreducible* by R, if there is no term t such that $s \rightarrow_R t$. A term t is *reachable* from a term s with the rewrite system R if $s \xrightarrow{*}_R t$. Two terms s, t are *joinable* if there exists a term u reachable from s and t.

The properties of a rewrite system R are those of the relation $\rightarrow_R$. All these properties, in particular termination and confluence are undecidable in general. This is not surprising because term rewriting is at least as expressive as Turing machines. Indeed, Turing machines can be expressed as a single rewrite rule [Dauchet, 1992]. However, there are methods for deciding these properties for specific classes of rewrite systems. For example, termination of a rewrite system can be proved through the use of an appropriate simplification ordering thanks to the following theorem.

Theorem 2. *[Dershowitz, 1982] Let $\mathcal{F}$ be a signature with a finite set of symbols. A rewrite system R over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ terminates if there is a simplification order $\succ$ such that $l \succ r$ for each rule $l \rightarrow r \in R$.*

Confluence can be decided for terminating rewrite systems by applying the Newman's lemma which assures that local confluence implies the confluence for these systems. Local confluence can be decided by testing the joinability of *critical pairs* [Knuth and Bendix, 1970].

Definition 31 (Critical Pair)**.** *Let $l \rightarrow r$ and $g \rightarrow d$ be two rules with disjoint sets of variables. We call a critical pair in the rule $g \rightarrow d$ over $l \rightarrow r$ at the non variable position $p \in \mathcal{P}os(l)$, the pair $(\sigma(r), \sigma(l)[\sigma(d)]_p)$ such that σ is a mgu of g and $l|_p$.*

If every critical pair is joinable, the rewrite system is locally confluent. Since the number of critical pairs in a finite rewrite system is also finite, local confluence is decidable.

A left linear rewrite system which does not have any critical pair is *orthogonal*. For example, the system from example 6 is orthogonal. Orthogonal rewrite systems are not necessarily terminating, but they are always confluent [Huet, 1980]. Two rules *overlap* if they give rise to a critical pair.

A *constructor system* $(\mathcal{C}, \mathcal{D}, R)$ is defined by a set of constructors $\mathcal{C}$, a set of defined functions $\mathcal{D}$ and a set of rewrite rules R, such that every left-hand side of any rule in R is of the form $f(t_1, ..., t_n)$ with $f \in \mathcal{D}$ and $t_1, ..., t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Two constructor systems $(\mathcal{C}_1, \mathcal{D}_1, R_1)$ and $(\mathcal{C}_2, \mathcal{D}_2, R_2)$ share constructors if $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{C}_1 \cup \mathcal{C}_2$ are pairwise disjoint, and $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$.

Definition 32 (Sufficient Completeness)**.** *A constructor system $(\mathcal{C}, \mathcal{D}, R)$ is sufficiently complete if for all $t \in \mathcal{T}(\mathcal{D} \cup \mathcal{C})$ there exists $s \in \mathcal{T}(\mathcal{C})$ such that $t \xrightarrow{*}_R s$.*

3.2.1 Conditional rewriting

Conditional rewrite systems arise naturally in some of the specifications adopted in this thesis. We formally define them as follows (which corresponds to *Normal 1-CTRS* in the classification given in [Ohlebusch, 2002a]):

Definition 33 (Conditional Rewriting)**.** *A conditional rewrite system is a set of conditional rewrite rules R over a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Each rewrite rule is of the form $l \rightarrow r$ if $s_1 \rightarrow t_1, \ldots, s_k \rightarrow t_k$ with $l, r, s_1, \ldots, s_k, t_1, \ldots t_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.*

- *For all rules in R rewrite system $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$, where c is an abbreviation for the conditional part of the rule, $s_1 \rightarrow t_1, \ldots, s_k \rightarrow t_k$.*
- *Each t_j in c is a ground normal form with respect R_u, which contains all rules in R without their conditional part.*

Definition 34. *Given a conditional rewrite system R, a term t rewrites to a term t', which is denoted as usual $t \rightarrow_R t'$ if there exists:*

- *a conditional rewrite rule $l \rightarrow r$ if c,*
- *a position ω in t,*
- *a substitution σ satisfying $t_{|\omega} = \sigma(l)$, and $\sigma(s_1) \rightarrow_{R_u} t_1, \ldots, \sigma(s_k) \rightarrow_{R_u} t_k$.*

3.2.2 Rewriting Modulo an Equational Theory

In this section we introduce the notion of rewriting modulo a set of equations. When the axioms of an equational theory can be oriented into a canonical rewrite system, the rewrite rules can be used for solving the word problem in such theory. However, there are equalities that cannot be oriented without loosing the termination property. A typical example is the commutativity axiom. In this case, equational reasoning needs a different rewrite relation which works on term equivalence classes modulo these non-orientable equalities.

Definition 35 (Rewriting Modulo Equivalence Classes)**.** *Given a rewrite system R and a set of axioms $\mathcal{E}$, the term t rewrites into the term s by R modulo $\mathcal{E}$, denoted $t \longrightarrow_{R/\mathcal{E}} s$, if there is a rule $l \rightarrow r \in R$, a term u, a position p in u and a substitution σ, such that $t \stackrel{*}{\longleftrightarrow}_{\mathcal{E}} u[\sigma(l)]_p$ and $s \stackrel{*}{\longleftrightarrow}_{\mathcal{E}} u[\sigma(r)]_p$.*

The relation $\longrightarrow_{R/\mathcal{E}}$ is not satisfactory with respect to efficiency because in order to rewrite a term, it is necessary to search in the whole equivalence class modulo $\mathcal{E}$. Such a search is even harder in the case of infinite equivalence classes. In order to solve this problem, a weaker relation has been proposed by [Peterson and Stickel, 1981], and generalized by [Jouannaud and Kirchner, 1986], in which matching is replaced by matching modulo an equational theory. This relation is called *rewriting modulo an equational theory* and is denoted $\rightarrow_{R,\mathcal{E}}$.

In practice, the most used equational theory is associativity and commutativity. The relation $\rightarrow_{R,\mathcal{E}}$ is called in this case rewriting modulo associativity and commutativity (AC).

The efficiency of matching modulo AC is essential for the performance of rewriting modulo AC. However, matching modulo AC is know to the a NP-Hard problem [Benanav et al., 1987] and it can have an exponential number of solutions.

In actual implementations, in order to reduce the search space for matching modulo AC algorithms, terms are transformed into an ordered canonical representation [Hullot, 1979]. This transformation is made in two stages: *flattening* which consists in transforming a term in a *variadic term* (where the arity of some symbols is not fixed), and *sorting* the subterms of AC symbols. In the flattening phase, the AC symbols are used as variadic symbols.

Example 7. *Given the signature defined by the function symbol* $+ : Term \times Term \rightarrow Term$ *and the constants* $\{a, b, c\}$ *of sort* $Term$*. The variadic term corresponding to the term* $t = (a + c) + ((b + a) + a)$ *is* $a + c + b + a + a$*.*

3.3 Strategic Rewriting

The notion of strategy used in this thesis is fundamental in rewriting. We give here a general presentation of the main ideas.

A term rewrite system R generates an abstract reduction system $\mathcal{R} = (\mathcal{O}_R, \mathcal{S}_R)$, whose nodes are terms $\mathcal{O}_R = \mathcal{T}(\mathcal{F}, \mathcal{X})$, and whose oriented edges are rewriting steps: $\mathcal{S}_R = \{t \rightarrow t' | t \rightarrow_\omega^R t'$ for ω a position in $t\}$. An *$\mathcal{R}$-derivation* or *$\mathcal{R}$-reduction sequence* is a path π in the graph $\mathcal{R}$.

Note that an $\mathcal{R}$-derivation is the concatenation of its reduction steps. The concatenation of π_1 and π_2 when it exists, is a new $\mathcal{R}$-derivation.

It is common to identify a rewrite system (i.e., a set of rewrite rules) with the abstract reduction system it generates (i.e., the set of all derivations allowed by R). To a rewrite system corresponds directly a unique abstract reduction system that can be seen as a generic way to describe the set of all derivations. The converse is not true since from a set of derivations, the generating rewrite system is not in general uniquely determined.

3.3.1 Abstract Strategies

Definition 36 (Abstract Strategy)**.** *For a given ARS* $\mathcal{R}$*:*

1. *An* abstract strategy ζ *is a subset of the set of all derivations (finite or not) of* $\mathcal{R}$*.*

2. *Applying the strategy* ζ *on an object* a *is denoted* $[\zeta](a)$*. It denotes the set of all objects that can be reached from* a *using a derivation in* ζ*:*

$$[\zeta](a) = \{b \mid \exists \pi \in \zeta \; such \; that \; a \rightarrow^\pi b\} = \{[\pi](a) \mid \pi \in \zeta\}$$

   When no derivation in ζ *has source* a*, we say that the strategy application on* a *fails.*

3. *Applying the strategy* ζ *on a set of objects consists in applying* ζ *to each element* a *of the set. The result is the union of* $[\zeta](a)$ *for all* a *in the set of objects.*

4. *The* domain *of a strategy is the set of objects that are source of a derivation in* ζ*:*

$$Dom(\zeta) = \bigcup_{\delta \in \zeta} Dom(\delta)$$

5. *The strategy that contains all the empty derivations is denoted* Id*:*

$$Id = \{id_a \mid a \in \mathcal{O}\}$$

3.3.2 Properties under Strategies

Definition 37 (Termination under strategy)**.** *For a given abstract reduction system abstract reduction system $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ and a strategy ζ:*

- *$\mathcal{A}$ is terminating under ζ if all derivation in ζ are of finite length;*
- *An object a in $\mathcal{O}$ is ζ-normalized when the empty derivation is the only one in ζ with source a;*
- *A derivation is ζ-normalizing when its target is ζ-normalized;*
- *An ARS is weakly ζ-terminating if every object a is the source of a ζ-normalizing derivation.*

Definition 38 (Weak Confluence under Strategy)**.** *An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is weakly confluent under an strategy ζ if for all objects a, b, c in $\mathcal{O}$ and all derivations in $\mathcal{A}$-derivations π_1 and π_2 in ζ, when $a \rightarrow^{\pi_1} b$ and $a \rightarrow^{\pi_2} c$ there exists d in $\mathcal{O}$ and two derivations π'_3, π'_4 in ζ such that $\pi'_3 : a \rightarrow c \rightarrow d$ and $\pi'_4 : a \rightarrow b \rightarrow c$*

Definition 39 (Strong Confluence under Strategy)**.** *An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is strongly confluent under an strategy ζ if for all objects a, b, c in $\mathcal{O}$ and all $\mathcal{A}$-derivations π_1 and π_2 in ζ, when $a \rightarrow^{\pi_1} b$ and $a \rightarrow^{\pi_2} c$ there exists d in $\mathcal{O}$ and two derivations π_3 and π'_4 in ζ such that:*

1. *$c \rightarrow^{\pi_3} d$ and $b \rightarrow^{\pi_4} d$;*
2. *$\pi_4; \pi_1, \pi_3; \pi_2 \in \zeta$.*

It is now possible to give the definition of strategic rewriting:

Definition 40 (Strategic rewriting)**.** *Given an abstract reduction system $\mathcal{R} = (\mathcal{O}_R, \mathcal{S}_R)$ generated by a rewrite system R, and a strategy ζ of $\mathcal{R}$, a strategic rewriting derivation (or rewriting derivation under strategy ζ) is an element of ζ. A strategic rewriting step under ζ is a rewriting step $t \rightarrow^R_\omega t'$ that occurs in a derivation of ζ.*

A strategy can be described by enumerating all its elements or more suitably by a *strategy language*. Various approaches have been followed, yielding slightly different strategy languages such as ELAN [Kirchner et al., 1995], Stratego [Visser, 2001], Tom [1] [Balland et al., 2007, Balland et al., 2006a] or more recently Maude [Clavel et al., 2007, Martí-Oliet et al., 2005]. All these languages share the concern to provide abstract ways to express control of rule applications, by using reflexivity and the meta-level for Maude, or the notion of rewriting strategies for ELAN or Stratego. Strategies such as bottom-up, top-down or leftmost-innermost are higher-order features that describe how rewrite rules should be applied. Tom, ELAN and Stratego provide flexible and expressive strategy languages where high-level strategies are defined by combining low level primitives. We describe below the main elements of the Tom strategy language, whose semantics is naturally expressed in the rewriting calculus [Cirstea and Kirchner, 2001, Cirstea et al., 2003].

[1] `http://tom.loria.fr`

We can distinguish two classes of constructs in the strategy language: the first class allows construction of derivations from the basic elements, namely the rewrite rules. The second class corresponds to constructs that express the control of strategy application such as recursion.

3.3.3 Strategy Constructors

Given a rewrite system R over $\mathcal{T}(\mathcal{F}, \mathcal{X})$, rewrite rules in R are elementary or atomic strategies. An elementary strategy is either Identity which corresponds to the set Id of all empty derivations, Fail which denotes the empty set of derivations $Fail$, or a set of rewrite rules R which represents one-step derivations with rules in R at the root position. For instance, if a and b are constants, the application of the rewrite rule $a \to b$ to the term a is denoted $[a \to b](a)$ and evaluates to $\{b\}$.

A strategy expression ζ may take arguments $\zeta_1, \ldots, \zeta_n$, and the resulting expression is expressed functionally: $\zeta(\zeta_1, \ldots, \zeta_n)$. Notice that this is consistent with the notation $\zeta(R)$ as soon as the definition of ζ does not depend on its arguments order. When it is clear from the context, we identify the strategy expression and the strategy (i.e. the set of derivations it represents). In a consistent way, the application of a strategy expression to a term is defined as the application of the strategy it represents.

A simple strategy is the sequential application of two rules. It is described by the concatenation operator "`seq`". For instance $[\texttt{seq}(l_1 \to r_1, l_2 \to r_2)](t)$ denotes $[l_2 \to r_2]([l_1 \to r_1](t))$. Then $\texttt{seq}(\zeta_1, \zeta_2)$, which could also be denoted $\zeta_1; \zeta_2$, is defined as the concatenation of ζ_1 and ζ_2 whenever it exists. This strategy operator extends naturally to multiple arguments:

$$[\texttt{seq}(\zeta_1, \ldots, \zeta_n)](t) \;=\; [\zeta_n]([\zeta_{n-1}](\ldots[\zeta_1](t)))$$

Identity and failure are strategies are given below:

$$[\texttt{id}](t) = \{t\}$$

$$[\texttt{fail}](t) = \emptyset$$

The strategy computing all derivations obtained by application of a rewrite system R is called `universal` ; it takes as argument the set of rules under consideration:

$$[\texttt{universal}(R)](t) \;=\; \{t' \mid t \xrightarrow{*}_R t'\}$$

For instance, we have:

$$\begin{array}{lcl} [\texttt{universal}(a \to a)](a) & = & \{a\} \\ [\texttt{universal}(f(x) \to f(f(x))](f(a)) & = & \{f(a), f(f(a)), f(f(f(a))), \ldots\} \end{array}$$

One can successively try to apply several strategies using the `choice` operator: its first argument is applied if it does not fail, otherwise the second one is applied (and may fail as well).

$$\begin{array}{lcll} [\texttt{choice}(\zeta_1, \zeta_2)](t) & = & [\zeta_1](t) & \text{if } [\zeta_1](t) \neq \emptyset \\ [\texttt{choice}(\zeta_1, \zeta_2)](t) & = & [\zeta_2](t) & \text{if } [\zeta_1](t) = \emptyset \end{array}$$

Clearly `choice` is associative and therefore its syntax is extended to be applicable to a list of strategies:

$$\texttt{choice}(\zeta_1, \zeta_2, \ldots, \zeta_n) \;=\; \texttt{choice}(\zeta_1, \texttt{choice}(\zeta_2, \ldots, \zeta_n))$$

Other strategies allow controlling the application of rules over sub-terms of a term. The strategy `one` must succeed on at least one of the sub-terms of a term. Note that this operator is deterministic. On the other hand, `all` application must succeed on each sub-term, otherwise, the result is failure:

$$\begin{array}{rcl}
[\texttt{one}(\zeta)](f(t_1, \ldots, t_n)) & = & f(t_1, \ldots, [\zeta](t_i), \ldots, t_n), \text{ if } [\zeta](t_i) \neq \emptyset \\
[\texttt{all}(\zeta)](f(t_1, \ldots, t_n)) & = & f([\zeta](t_1), \ldots, [\zeta](t_n)), \text{ if } \forall i \in \{1, \ldots, n\}, [\zeta](t_i) \neq \emptyset
\end{array}$$

Using the above set of operators, we can define recursive ones which iterate the application of a strategy to a term, for example:

$$\texttt{try}(\zeta) = \texttt{choice}(\zeta, \texttt{id})$$

$$\texttt{repeat}(\zeta) = \texttt{try}(\texttt{seq}(\zeta, \texttt{repeat}(\zeta)))$$

It is worth noticing that `try` and `repeat` never fail. Other high-level strategies implement term traversal and normalization on terms and are well-known in the rewrite system literature:

$$\begin{array}{lcl}
\texttt{topDown}(\zeta) & = & \texttt{seq}(\zeta, \texttt{all}(\texttt{topDown}(\zeta))) \\
\texttt{bottomUp}(\zeta) & = & \texttt{seq}(\texttt{all}(\texttt{bottomUp}(\zeta)), \zeta) \\
\texttt{OnceTopDown}(\zeta) & = & \texttt{choice}(\zeta, \texttt{one}(\texttt{OnceTopDown}(\zeta))) \\
\texttt{OnceBottomUp}(\zeta) & = & \texttt{choice}(\texttt{one}(\texttt{OnceBottomUp}(\zeta)), \zeta) \\
\texttt{innermost}(\zeta) & = & \texttt{repeat}(\texttt{onceBottomUp}(\zeta)) \\
\texttt{outermost}(\zeta) & = & \texttt{repeat}(\texttt{onceTopDown}(\zeta))
\end{array}$$

Example 8. *Some examples of strategy application are:*

$$\begin{array}{lcl}
[\texttt{universal}(a \to b, a \to c)](a) & = & \{a, b, c\} \\
[\texttt{choice}(a \to b, a \to c)](a) & = & \{b\} \\
[\texttt{choice}(a \to c, a \to b)](b) & = & \emptyset \\
[\texttt{try}(b \to c)](a) & = & \{a\} \\
[\texttt{repeat}(\texttt{choice}(b \to c, a \to b))](a) & = & \{c\}
\end{array}$$

3.4 The Tom System: Term Rewriting Systems in Java

Tom [Balland et al., 2007][2] is a language extension which adds strategic rewriting capabilities to Java. A Tom program is the combination of a host program in Java with code fragments delimited by some special purpose constructs to define rewrite systems.

The functionalities introduced by Tom syntactic matching, associative matching with unit axioms, anti-patterns [Kirchner et al., 2007], conditional rewriting, support for built-in data-types,

[2] `http://tom.loria.fr`

XML transformation facilities, and a modular strategy library. Tom has been used to implement various applications but one of the biggest applications is the Tom compiler itself, written in Tom and Java. The concept of making formal developments available on top of an existing language is called *Formal Island* [Balland et al., 2006b, Balland et al., 2006c].

The constructs of the Tom language useful for our purposes are the following ones:

- `%match` corresponds to an extension of `switch/case` construct in functional programming languages, which allows discriminating among a list of patterns.
- ` (backquote construct) is used to build terms from Java values.
- `%strategy` groups rules to form the basic building blocks for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

Therefore, a program is a list of Tom constructs (in the formal island metaphor, the island language) interleaved with some sequences of characters (from the ocean language). During the compilation process, all Tom constructs are translated and replaced by instructions of the host-language.

The following example shows how the addition of Peano integers can be defined. This supposes the existence of a data-structure and a mapping (defined using `%op`) where Peano integers are represented by *zero* and *successor*: the integer 3 is denoted by *suc(suc(suc(zero)))* for example.

```
public class PeanoExample {
  %op Term zero() { ... }
  %op Term suc(Term) { ... }
  ...
  Term plus(Term t1, Term t2) {
    %match(t1, t2) {
      x,zero   -> { return `x; }
      x,suc(y) -> { return `suc(plus(x,y)); }
    }
  }
  void run() {
   System.out.println("plus(1,2) = " +plus(`suc(zero),`suc(suc(zero))));
  }
}
```

In this example, given two terms t_1 and t_2 (that represent Peano integers), the evaluation of `plus` returns the sum of t_1 and t_2. This is implemented by pattern matching: t_1 is matched by x, t_2 is possibly matched by the two patterns $zero$ and $suc(y)$. When $zero$ matches t_2, the result of the addition is x (with $x = t_1$, instantiated by matching). When $suc(y)$ matches t_2, this means that t_2 is rooted by a suc symbol: the subterm y is added to x and the successor of this number is returned, using the ` construct. The definition of `plus` is given in a functional programming style, but the `plus` function can be used in Java to perform computations. This example illustrates how the `%match` construct can be used in conjunction with the considered native language.

In order to transform terms, it is necessary to state a relation between a Tom signature ($\mathcal{F}$ in the corresponding rewrite system) and Java objects. This relation is a mapping, either defined by hand or through an auxiliary tool, called Gom [Reilles, 2007], which is distributed with the Tom environment. Gom automatically generates the data structure implementation for a given term signature.

It is possible to define preferred canonical forms for algebraic terms in Gom through the *hook* mechanism [Reilles, 2007]. Hooks in Gom can contain Tom and Java code, which is executed every time a new term is created. Hooks are executed until a normal form for the term created is reached. In the code that follows we show an example of the use of hooks. The code implements *constant folding*: every time a term rooted by `Add` and `Mul` has subterms of sort `Nat`, in a small language for arithmetic expressions.

```
module Expressions
  imports String int
  abstract syntax
  Bool = True()
       | False()
       | Eq(lhs:Expr, rhs:Expr)
  Expr = Id(stringValue:String)
       | Nat(intValue:int)
       | Add(lhs:Expr, rhs:Expr)
       | Mul(lhs:Expr, rhs:Expr)
  Add:make(l,r) {
    %match(Expr l, Expr r) {
      Nat(lvalue), Nat(rvalue) -> {
        return `Nat(lvalue + rvalue);
      }
    }
  }
  Mul:make(l,r) {
    %match(Expr l, Expr r) {
      Nat(lvalue), Nat(rvalue) -> {
        return `Nat(lvalue * rvalue);
      }
    }
  }
```

The use of strategies in Tom is illustrated by the following example:

```
import eval.mydsl.types.*;
import tom.library.sl.*;
  public class Eval {
    %include { sl.tom }
     %gom {
         module mydsl
           imports int String
           abstract syntax
               Expr = val(v:int)
                    | var(n:String)
                    | plus(Expr*)
```

```
                    | mult(Expr*)
    }

  %strategy EvalExpr() extends `Identity() {
    visit Expr {
      plus(l1*,val(x),l2*,val(y),l3*) ->
        { return `plus(l1*,l2*,l3*,val(x + y)); }
      mult(l1*,val(x),l2*,val(y),l3*) ->
        { return `mult(l1*,l2*,l3*,val(x * y)); }
      plus(v@val(_)) -> { return `v; }
      mult(v@val(_)) -> { return `v; }
    }
   }

  public static void main(String[] args) throws VisitFailure {
    Expr e = `plus(val(2),var("x"),mult(val(3),val(4)),
                   var("y"),val(5));
    Expr res = (Expr) `Innermost(EvalExpr()).visit(e);
    System.out.println(e + "\n" + res);
   }
}
```

Above, symbols defined in the `%gom` signature followed by $\star$ are "list" symbol. They are associative with unit functions, where the empty list is the neutral element. The declaration `EvalExpr` corresponds to a rewrite system which simplifies expressions. Variables followed by $\star$ match lists of elements. It is meant to be applied on any sub-expression but not in top positions. We choose to apply the rewrite system in an innermost way .

We introduced several features used in the implementations we giving much detail about the exact syntax for Tom programs, which can be found on its user manual [Balland et al., 2006a].

Chapter 4

Policy Specification

Policy specification is an arduous activity: the precise specification of the security requirements in computerized systems demand a careful understanding of all the elements that influence how access control decisions are produced. It is essential that formal frameworks for policies provide both expressivity, to capture complex requirements, and reliable techniques to check policy properties.

In this chapter, we argue that term rewriting is an appropriate paradigm for expressing and reasoning about policies, in particular, about access control. The main advantage of term rewriting is to provide a formal semantics for modelling flexible policies. Such semantics is grounded on a logical and proof theory, where important policy properties can be stated and proved. Rewrite-based specifications are executable, so policies can be easily prototyped in one of the several languages and systems available today, such as ELAN [Borovanský et al., 1998], Maude [Clavel et al., 2007, Clavel et al., 2002], Tom [Balland et al., 2007].

In order to understand how the design of our framework was influenced, we present in Section 4.1 the progress from classical security models of access control to the most recent rule-based approaches for policy specification. Then, in Section 4.2 we give our definition of security policy, illustrate it through several examples, and discuss the formal properties concerning policies. Because most of the desirable policy properties are undecidable in general, in Section 4.4 we present two distinct approaches for defining a decidable rewriting sub-language for policies.

4.1 State of the Art

A *security model* is a specification of a system's requirements for confidentiality, integrity, or availability, that does not indicate any particular mechanism for achieving them. The word "model" may seem inappropriate, but it was adopted early in the computer security literature and kept since then. Essentially, a model establishes restrictions on the inputs and outputs of a system, which should be sufficient to ensure a given security requirement. In the following we will concentrate on confidentiality, which is the main focus of our work.

A main form of security policy consist in access control policies, which determine whose actions the principals of a system are or not allowed to perform over its resources, in a way some security objective can be achieved.

Several methods have been proposed to encode access control policies since Lampson [Lampson, 1974]. In his work, subjects, objects, and privileges are arranged in an access control matrix, where each cell contains the exact privileges of the user in its row, over the resource in its column. Therefore given a finite set of subjects S, a finite set of actions A, and a

finite set of objects O, we can represent an entry in the access control matrix (denoted M), is represented by

$$M[s, o] = \{a_1, \ldots, a_n\}, \text{where } s \in S, o \in O, a_1, \ldots, a_n \in A$$

The global state of a system in this model is a triple (S, O, M), where transitions occur via requests to alter the current set of privileges in the matrix. This model is still in use today with some improvements to accelerate the mediation to resources in operational systems, like in the most frequent of its variations the *Access Control List* (ACL), which has a more efficient data representation.

The work of Lampson was refined by Harrison, Ruzzo, and Ullman in [Harrison et al., 1976], who proposed the abstract model known today as the HRU model. Their goal was to analyze complexity issues in access control. The HRU model defines *protection systems*, which consist of an access control matrix and a set of small "programs" (*commands*) describing how to alter the matrix using some simple instructions following the syntax below:

$$\begin{array}{lll} \textbf{Command} & & name\ (X_1, \ldots, X_k) \\ & \textbf{if} & a_1 \in M[X_{s1}, X_{o1}] \text{ and} \\ & & a_2 \in M[X_{s2}, X_{o2}] \text{ and} \\ & & \ldots \\ & & a_m \in M[X_{si}, X_{oj}] \\ & \textbf{then} & \\ & & op_1 \\ & & \ldots \\ & & op_n \end{array}$$

where parameter names $X_1, \ldots, X_k$ can be instantiated only by subject or object identifiers, each a_i is an access right, and each op_k is one of the following operations, whose semantics corresponds exactly to their descriptions:

- **enter** $a \in A$ **into** $M[s, o]$,
- **delete** $a \in A$ **from** $M[s, o]$,
- **create subject** $s \in S$,
- **create object** $o \in O$,
- **destroy subject** $s \in S$,
- **destroy object** $o \in O$.

Therefore, it is possible to write small scripts which tell how to manipulate the permissions in a system.

Example 9. *The fact that a process running in an operating system owns the files it creates is exemplified in the following program:*

$$\begin{array}{ll} \textbf{Command} & CREATE\ (process, file) \\ & \textbf{create object}\ file \\ & \textbf{enter}\ own\ \textbf{into}\ [process, file] \end{array}$$

where $process$ *and* $file$ *are parameters. A subject can confer read rights to friends through the following conditional script:*

$$\begin{array}{ll} \textbf{Command}\ CONFER_{read}\ (owner, friend, file) & \\ \textbf{if} & own\ \in\ M[owner, file] \\ \textbf{then} & \textbf{enter}\ read\ \textbf{into}\ [friend, file] \end{array}$$

with parameters $owner$, $file$ *and* $friend$*.*

A configuration for a protection system is a matrix with sets of subjects, objects and rights; a new configuration is obtained through the execution of a sequence of commands. HRU introduces the following property:

Definition 41. *Given a particular protection system and a generic access privilege* a*, we say that the initial configuration* S_0 *is* unsafe for $a \in A$ *(or leaks* a*), if there is a sequence of commands that inserts the right* a *into a cell, if it did not already contained* a *in state* S_0*.*

Correspondingly, a state is safe for a right r if it is not unsafe for r. An important result proved in [Harrison et al., 1976] is that this safety problem is decidable only for *mono-operational* systems (*i.e.* every command contains a single primitive operation), the general problem being undecidable.

One could argue that this property is not of much interest in practice, when for example, one wants to share a file with a restricted group of principals: it is necessary to add access rights for these principals on that file, which violates the safety condition according to the HRU model. However, the safety property guarantees that a second set of (untrusted) subjects will not obtain such privileges in other ways. The pioneering work in the HRU model shows the hardness of the problem, even though we have a complete understanding of the operations that propagate rights in a system. We will further discuss this model in Chapter 5.

We call *Discretionary Access Control* (DAC), the models where subjects can freely pass rights they posses to other users. In general, DAC has the weakness of not offering any contingency mechanism in the event of *Trojan Horse* attacks. When subjects run third party programs, they deliver them all the privileges on their behalf, thus, no guarantees exist that such programs will not execute any malicious action on the user's files, then confidentiality is seriously compromised. *Mandatory Access Control* (MAC) offers an alternative to reduce the damage caused by such attacks, by labeling subjects and objects with a security level. In MAC we consider that an ordering for the security levels exists, and the observation of the ordering avoids the transmission of rights to access information from a high level security object, by any means, to a low level security subject.

The most studied MAC model is the BLP model [Bell and LaPadula, 1974], mainly used to implement military policies. In such model, the security levels are structured in a lattice [McLean, 1988, Sandhu, 1993], L, containing the elements $top\ secret \succeq secret \succeq confidential \succeq unclassified$. Sometimes, levels are combined with an extra label, to create different "compartments" (*e.g.* $[secret, NATO]$, $[secret, Navy]$, which are incomparable to each other).

In addition to security levels, Bell and Lapadula reuse the notion of access matrix, to build a series of definitions to come up with a characterization of secure system. Let $f : S \cup O \rightarrow L$, be the function that, when applied to a subject or an object, returns its corresponding security level. Also consider a set of states E, where each state is a pair (f, M), where M is an access control matrix, and a transition function $T : E \times Q \rightarrow E$ taking a state, E, a request Q to read or write, and yielding a new state, then:

- A state (f, M) is read secure (called *simple security* property) if and only if $\forall s \in S, o \in O, read \in M[s, o] \implies f(s) \geq f(o)$.
- A state (f, M) is write secure (called "star-property") if and only if $\forall s \in S, o \in O, write \in M[s, o] \implies f(o) \geq f(s)$.
- A state is secure if and only if it is read and write secure, and finally,
- A system (S_0, Q, T) is secure if and only if S_0 is state secure and every state reachable from S_0 by a finite sequence of requests from Q is state secure.

Equipped with these definitions, Bell and Lapadula could prove an important theorem about their model, which can be informally stated as "no reads-up and no writes-down". In other words, a subject whose security level is $secret$ can read $confidential$ files, but cannot write fresh information to them. On the other hand, the same subject can add data to $top\ secret$ files, but cannot obtain information from higher level objects. We further detail the BLP model in Chapter 5, where we show how rewriting can be applied to verify information flow policies.

Following the same principles as Bell and Lapadula, other models for confidentiality have been proposed, such as the *Chinese Wall* model (CW) [Brewer and Nash, 1989], whose main purpose is to control access to data from competing companies in a commercial services scenario. For instance, suppose a company that offers financial consulting for other companies, then a consultant of such company should not work for different clients that have conflicts of interest.

The CW model introduces other concepts than BLP. Each object, which can be thought of as a file, concerns only one company at a time. Company groups (cg) are a collection of files of one particular company, and conflict classes (cc) cluster competing companies. The principle of the security policy is the following: a subject can access any information from a new company as long as it has never seen information from another company in the same conflict class.

Then, the security condition is stated as follows. Let $prec(s)$ be the set of objects that a subject s has read. Then s can read an object o if one of the following is true:

- $\exists o' \in O$, such that $o' \in prec(s)$ and $cg(o) = cg(o)'$ (both are in the same company group),
- $\forall o' \in prec(S) \implies cc(o') \neq cc(o)$.

Then, similarly to BLP, a system is secure if, starting from a sanitized[1], secure state, all transitions preserve the security condition.

The problem of these general models is that they fail to capture certain requirements that may compromise their security properties in more concrete settings. For example, the possibility of downgrading the security levels of objects is omitted from the original formulation of BLP, therefore, as remarked in [McLean, 1985], it is possible to prove that an insecure system satisfies the security properties of the BLP model, given that security levels of objects or subjects can change along the system's execution. BLP also relies on the intuitive notion that the *read* and *write* actions imply that information goes in only one direction, from objects to subjects, and from subjects to objects, respectively. This is not the case in actual computer systems.

Role-based Access Control

Role-based Access Control (RBAC) models [Sandhu et al., 1996] have emerged as a solution to manage access control in large organizations. The concept of role captures the idea of having temporarily some privileges for executing a task. Access rights are assigned to roles, and users can activate one role at a time, provided that they are associated to the role they require to enable. This indirection level allows one to manage large databases of users and simplifies the work of the system administrator.

The amount of management activities can be further decreased by the creation of role hierarchies. This is done via an order relation between roles which establishes that a role inherits the privileges assigned to another. In addition to hierarchies, roles can also belong to a relation stating that they are mutually exclusive [Sandhu et al., 2000], which is actually a manner to implement separation of duty constraints – a principal cannot assume a given pair of roles declared conflicted with one another.

A lot of work on RBAC has been carried out by several authors [Sandhu et al., 1996, Sandhu et al., 2000, Gavrila and Barkley, 1998, Cuppens and Miège, 2004, Sandhu et al., 1999, Pavlich-Mariscal et al., 2005, Barker and Fernández, 2006], the model is still of much interest nowadays, and is actually used in network policies, web-based systems, operating systems, etc. Numerous extensions have been proposed, to cope with various kinds of constraints on role hierarchies, and to add some "structure" to subjects and objects, called attributes.

XACML

The XACML specification [Moses, 2005] is an initiative from the Oasis Consortium[2] to create a standard markup language for access control. The language provides basic constructs for

[1] Sanitized information can be accessed by any subject, since it should not concern any particular company.

[2] `http://www.oasis-open.org/`

selecting an access decision based on the subjetcs, actions, and objects in the system and their attributes.

In XACML it is possible to specify attributes for principals (and/or roles) and resources, defining the "context" in which access requests are made. Rules have a target (the resource being protected), an effect (an authorization decision, which ranges over $permit, deny, not\ applicable$, or $indeterminate$), and a condition. Conditions are expressions built from a predefined set of predicates over the attributes of subjects and objects.

Access is permitted whenever the conditions in the set of rules are matched against the values provided in the current access request, and the environment of the policy. In the case the conditions are not satisfied, access is denied. However, if the request does not match the target of a rule, the evaluation result is $not\ applicable$[3]. In the case of any errors, the result is set to $indeterminate$.

One of the strong points of XACML is its set of composition primitives, further detailled in Chapter 6. The XACML standard is described informally in a series of normative documents [Moses, 2005]. A candidate formal specification was proposed, which uses the functional programming language Haskell to give an operational semantics to the language [Humenn, 2003].

The XML syntax allows developpers to communicate policies across different access control platforms. Indeed, the specification also covers the architecture for enforcing access control in distributed environments. It is worth to mention two of its components, the Policy Decision Point (PDP), which centralizes the capability of producing an access control decision given a request and the current context, and the Policy Enforcement Point (PEP), which intercepts requests for access to resources.

XACML provides support for very interesting policies, used in practice in several organizations, where authorization decisions depend on the current values of attributes forming the policy environment. However, we will see in the next section that the explicit support for rules furnishes much more flexibility in policy specification, because there is the possibility for the user to define his own predicates important for access control, and sometimes, to define functions that influence how decisions are produced.

Rule-Based Approaches

Rule-based formalisms are well-adapted for expressing various kinds of problems in a declarative way – see the ever growing interest in all kinds of rule-based paradigms like production systems, functional and logic programming, and rule-based approaches for the semantic web[4]. This is also the case for security policies, where prohibitions, permissions, and obligations are naturally stated in the form of rules. Rules bring several advantages: they are intelligible to people, even non-specialists in logic can fairly well state some policy rules; rule-based languages have formal semantics (model-theoretic, operational, or denotational).

The evolution from the classic models we presented above to rule-based policy specifications was a natural choice, mainly because the latter fulfills the necessity of clearly expressing application specific policies. Policies are then understood as sets of rules which capture complex

[3] Rules cannot declare $not\ applicable$ as effect explicitly.

[4] `www.rewerse.net`

dynamic conditions that need to be evaluated in the access control process. Moreover, this rule-based approach allows one to suitably state and check policy properties.

Different application domains have peculiarities that need to be taken into account by the underlying policy formal model. These diverse constraints gave raise to several rule-based frameworks for access control in the recent years [Jajodia et al., 2001, Halpern and Weissman, 2003, Kalam et al., 2003, Becker and Sewell, 2004b], to cite a few. These works propose formal, rule-based languages for policies. A detailed discussion of these works is presented in Section 4.6. The design of such frameworks is influenced by additional requirements: in some cases, the main requirement a given framework addresses is to deal with time constraints – thus representing time, and reasoning about durations of permissions and time intervals is a must. In other cases, the capacity to rapidly answering to access control requests over huge databases is more important. Such considerations guide the choice of the appropriate logic for access control, and this choice has to be balanced with the properties which can be verified in the formal model.

We can state the following desiderata for rule-based access control languages:

- the framework must provide a formal semantics, such that it is possible to rely on the properties of the underlying logical framework in order to prove policy properties,
- the policy model must allow one to flexibly capture policies that depend on a given application context, thus to be able to represent the data environment of the application and to correctly associate decisions to access requests,
- the language must have an efficient implementation,
- a rule-based framework for policies must support policy composition, and allow its users to define how policies should be combined adequately, *i.e.* allow them to define their own composition operators, and
- the formal model must be suitable for policy verification, techniques and tools must be available to check policy properties.

The formal framework we present in this chapter covers all these issues in a convenient and powerful way. First, it is possible to build application-specific security policies, yet being endowed by a theory that allows for general reasoning about the properties expected for a given policy model. In the next section, we introduce rewrite-based policies. Term rewriting was applied for policy specification in [de Oliveira, 2007]. The section that presents an extended framework with respect to [de Oliveira, 2007], which appeared in [Dougherty et al., 2007b], addressing a much larger range of policies. Policy compositions using rewriting strategies, as well as related works on policy composition are the main topic of Chapter 6.

4.2 A Rewrite-Based Framework

Recent developments in access control aim at expressing various constraints on the environment where policies run, in order to capture real world requirements from policy authors. In such

scenarios, it is necessary that the policy definition incorporates enough expressive power, such that one can fully define the behavior of policies as a computation towards an access decision.

This section presents a formalization of access control policies based on term rewriting. Here, policies are defined by a set of rewrite rules, governed by a rewrite strategy. The access requests and the environment where policies are enforced are represented as algebraic terms. When applied to an access request the rewrite rules induce an evaluation process which eventually returns an authorization decision. This formalization allows us to capture many dynamic aspects which are important for policy enforcement, as we may see in several examples of this section.

Definition 42 (Security Policy)**.** *A security policy, $\wp$, is a 5-tuple $(\mathcal{F}, D, R, Q, \zeta)$ where:*

1. *$\mathcal{F}$ is a signature;*
2. *D is a non-empty set of closed terms called decisions: $D \subseteq \mathcal{T}(\mathcal{F})$;*
3. *R is a set of rewrite rules over $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
4. *Q is a set of closed terms from $\mathcal{T}(\mathcal{F})$, $\mathcal{Q} \subseteq \mathcal{T}(\mathcal{F})$, called access requests;*
5. *ζ is a rewrite strategy for R.*

In the following, we explain the elements in the definition above.

- First we consider that the policy specification and its environment are described as terms built over a signature $\mathcal{F}$.
- The set of possible decisions generated by the policy is denoted by D. In most access control languages, D is a set containing only the constants $permit$ and $deny$. However, as we will see later, it is crucial to model policies that explicitly exempt themselves from taking a decision. Therefore, it is useful to dispose of additional elements in the set of decisions, such as $not\ applicable$, which expresses the fact that a given policy cannot decide about a given access request under the current context (this aspect is well motivated in [Bruns et al., 2007]). Moreover, the result returned by a policy can be more elaborated than just a constant, thus it can be a term containing further information like the time and duration an access is granted, for example.
- The rewrite system R describes the behavior of the policy, and if necessary, any computations which help to explain how the policy environment evolves.
- The access requests are closed terms. They typically express questions of the form: should the system make a transition to another state, given the current configuration of the policy environment.
- The last component is the strategy which allows one to finely specify how the policy rules must be applied. Thus, an appropriate strategy selects derivations in R that lead to access decisions.

One of the main consequences of this approach, in addition to expressivity, which we illustrate on the examples below, is that it allows us to take advantage of available results from the rewriting theory. Amongst such results, we investigate in Section 4.3 confluence, termination and sufficient completeness.

Example 10. *As already suggested in the introduction, we can model a policy for a medical system (this example is adapted from the XACML specification [Moses, 2005], and first presented in the rewrite-based formalism in [de Oliveira, 2007]).*

- *The policy signature $\mathcal{F}$ contains the following symbols, with their corresponding profile declarations:*

$$\begin{array}{lll}
accs & : Request \times Attribute & \rightarrow Decision \\
req & : Subject \times Action \times Object & \rightarrow Request \\
read,\ write & : & \rightarrow Action \\
permit,\ deny,\ na & : & \rightarrow Decision \\
patient,\ phy & : Number & \rightarrow Subject \\
admin,\ per & : Number & \rightarrow Subject \\
record & : Number & \rightarrow Object \\
guard & : Subject \times Subject & \rightarrow Attribute \\
respPhy & : Subject \times Subject & \rightarrow Attribute
\end{array}$$

  Note that in the signature above the return sort of the function $accs$ is $Decision$, which is the same for the constants $permit$ and $deny$. With order-sorted *signatures we could give a more natural representation for these symbols, as we proposed in [de Oliveira, 2007]. However, in this thesis, we preferred to keep the compatibility with the Tom system, which deals only with many-sorted signatures currently.*

- *The set of decisions is $D = \{permit, deny, na\}$.*

- *$R = R' \cup R''$ is the following set of rules, where variables are $x, y : Number$; $r : Object$; $a : Attribute$:*

$$R' = \left\{ \begin{array}{ll}
accs(req(patient(x), read, record(x)), a) & \rightarrow permit \\
accs(req(per(x), read, record(y)), guard(per(x), patient(y)))) & \rightarrow permit \\
accs(req(phy(x), read, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\
accs(req(phy(x), write, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\
accs(req(admin(x), read, r), a) & \rightarrow deny \\
accs(req(admin(x), write, r), a) & \rightarrow deny.
\end{array} \right.$$

$$R'' = \left\{ accs(q, a) \rightarrow na. \right.$$

  In the order of occurrence, these rules state that: a patient can read his own record, the guardian of a person can read the record for that person, the responsible physician for a patient can read or write data for this patient's record, the last two rules deny any access from administrators to patient records. The last rule is a default case for this policy, where $q : Request$.

- *The set of requests Q is the set of all well-sorted terms with $accs$ as top symbol.*
- *One can adopt the strategy $\zeta = \texttt{choice}(R', R'')$. The terms in Q which are not reduced by the rules from R' will be rewritten into na by R''.*

The example given below was taken from the NetFilter how-to[5].

Example 11. *Suppose an Internet user wants to set his firewall to block any traffic coming from the exterior to his local network. Since the interface associated to Internet connections is usually* `ppp0`*, a simple method is to reject all new packets coming from this source. In order to demonstrate the fact that it might be convenient for a policy to contain rules beyond those which* directly *compute decisions, we also give some additional rules which allow two different local computers to share the same external IP address: for each outgoing packet whose origin is a local machine, its head is rewritten to a single address.*

- *Let the policy signature be:*

$$\begin{array}{lll} pckt & : Address \times Address \times State & \rightarrow Packet \\ filter & : Packet & \rightarrow Decision \\ new, established & : & \rightarrow State \\ drop, accept & : & \rightarrow Decision \\ eth0, ppp0 & : & \rightarrow Address \end{array}$$

  The constructor for packets takes three arguments: a source address, a destination address and the state of the connection, which can be either $established$ or new. In addition to the constants $eth0$ and $ppp0$, any valid IP address in the standard notation is of sort $Address$.

- *The set of of constant symbols representing decisions is $D = \{accept, drop\}$.*
- *Consider R as the following rules, where $src, dst : Address$, and $s : State$ are variables:*

$$\begin{array}{lll} filter(pckt(src, dst, established)) & \rightarrow & accept \\ filter(pckt(eth0, dst, new)) & \rightarrow & accept \\ filter(pckt(ppp0, dst, new)) & \rightarrow & drop \\ filter(pckt(123.123.1.1, dst, new)) & \rightarrow & accept \\ pckt(10.1.1.1, ppp0, s) & \rightarrow & pckt(123.123.1.1, ppp0, s) \\ pckt(10.1.1.2, ppp0, s) & \rightarrow & pckt(123.123.1.1, ppp0, s) \end{array}$$

  The first rule says that packets concerning established connections have to be accepted. The second rule matches any packet whose origin is the local network, which are accepted. The third rule rejects any new packet coming from the external network ($ppp0$). The last two rules match the IP address of the local machines and rewrites them to a single visible IP address visible from the external network, and that will be accepted according to the forth rule.

[5] `http://www.netfilter.org`

- *The set Q contains ground terms with top symbol $filter$.*
- *A possible strategy for this policy, among other possibilities which guarantee a normalization process, is $\zeta = \texttt{innermost}(R)$.*

An example of request reduction for this example is as follows:

$$filter(\underline{pckt(10.1.1.2, ppp0, established)}) \rightarrow \underline{filter(pckt(123.123.1.1, ppp0, established))} \rightarrow accept$$

This defines a security policy as all conditions of Definition 42 are satisfied. What is important to notice in this example is the presence of recursive rules, which are important for the policy definition, but that do not directly derive permissions.

The next example illustrates a policy where decisions can have some structure, instead of being only constants.

Example 12. *In Nancy, France, the public transportation system adopts a magnetic ticket which stores the number of remaining trips and the time and date of its last validation. Each passenger must validate the ticket once he/she gets inside the bus or tramway. The user has the right to take as many correspondences as necessary within the delay of one hour, since the last validation of the ticket. This policy can be stated in our framework as follows:*

- *The signature $\mathcal{F}$ contains the symbols*

$$\begin{array}{ll} ticket : Nat \times Time & \rightarrow Decision \\ q : Ticket \times Time & \rightarrow Decision \\ deny & \rightarrow Decision \end{array}$$

- *The set of decisions D =* $\{deny, ticket(\mathcal{T}(\mathcal{F}), \mathcal{T}(\mathcal{F}))\}$
- *R is the following set of conditional rules:*

$$\begin{cases} q(ticket(n, time), current\ time) \rightarrow ticket(n-1, current\ time) \\ \textit{if } (current\ time > time + 60) \rightarrow true \\ \\ q(ticket(n, time), current\ time) \rightarrow ticket(n, time) \\ \textit{if } (current\ time \leq time + 60) \rightarrow true \\ \\ q(ticket(0, time), current\ time) \rightarrow deny \end{cases}$$

- *The requests are ground terms of the form $q : Ticket \times Time \rightarrow Decision$*
- *The strategy is $\zeta = \texttt{universal}(R)$*

We assume that the number of trips in a ticket is a natural number, thus the subtraction operation, $-$, is defined as usual. The variable n is of type Nat. We also assume that a theory for dealing with time exists, defining the symbols $<, +$ with the expected semantics. How this theory is defined is not relevant for this example. The variable $current\ time$ and $time$ have sort $Time$ and stand respectively to the current clock date and time, and for the date and time of the last validation for a given ticket.

The first rule decreases the remaining trips in the ticket if more than one hour has passed since the last time it was validated. The second rule allows a passenger to continue the trip without modification to the ticket. And the last rule denies access to the transportation if there are no more remaining trips in the ticket, in this case, the passenger must buy a new ticket.

When requests are closed terms, they represent some true fact about the policy environment. In our model, we assume that additional mechanisms are employed by a reference monitor to verify the authenticity of requests. Therefore the source of requests are properly identified, and we can take for granted that they were not forged by a malicious entity. Then, for closed requests, the semantics of a policy is given as a function that evaluates request terms into decisions by applying the rewrite rules (following some strategy) until a normal form is reached – and taken as the final decision given by the policy. This interpretation may not be suitable if the requests contain variables (or parameters), because in this case, the result itself may also contain variables. Then one can imagine that there must be some theory explaining how to instantiate decisions with variables. Of course term rewriting with strategies can be used to describe such "parameterized" policies, but the proof theory has to be extended to handle such specifications, like in languages as OBJ [Goguen and Malcolm, 2000] or Maude [Clavel et al., 2002], such that, by their turn, parameterized decisions can be safely instantiated.

In contrast, the semantics for policies containing variables in requests should have a somewhat different interpretation. Consider a request from a subject x to read the file `/etc/passwd`, written as $q(x, read, /\texttt{etc/passwd})$. This shall be understood as a query to the policy, which asks who are the principals that can read this file. This view is close to logic programming: requests are open formulas and a solving process must be used to find all ground values that satisfy this request/formula. Such an approach can rely on narrowing [Hullot, 1979, Antoy et al., 2000]. The narrowing relation substitutes the pattern matching mechanism by unification. Thus it is possible to answer queries with the use of the rewrite rules defining a rewrite-based policy, but we do not explore this research direction in this thesis.

4.3 Policy Properties

In this section we discuss some crucial policy properties. We start by defining *consistency*, which is a recurrent problem in several rule-based approaches for policy specification which support permissions and prohibitions. Next, we introduce *termination*, which assures that every request evaluation from a policy is finite, then we define *decision completeness* which means that for every access request, there exists a corresponding access decision.

Consistency

A security policy is consistent if it computes at most one access decision for a given input request. Nevertheless, policy rules are intrinsically partial or conflicting. It seems that people have a really hard time when they need to express what is allowed and what is forbidden. Consider for example the frequent ambiguities that we encounter when reading all kinds of regulations in natural language. Hence, it is extremely important to dispose of robust reasoning techniques for searching for inconsistencies among the policy rules.

Definition 43 (Consistency)**.** *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is* consistent *if for every query $q \in Q$, ζ applied to q returns at most one result: $\forall q \in Q$, the cardinality of $[\zeta](q) \cap D$ is less than or equal to* 1*.*

This means that for every query evaluation, a deterministic result is computed by the application of ζ on the terms of Q. One must be careful when providing a strategy for the application of the rewrite rules of a given system, such that the strategy effectively leads to access decisions. Let us examine a few possible situations.

In the case the strategy produces only infinite derivations with source q (as in Figure 4.1(a)), then the cardinality of $[\zeta](q)$ is 0, and the policy is consistent. If all derivations are finite, but the normal forms do not belong to the set of decisions (as in Figure 4.1(b), with $t, t' \notin D$), the policy is still considered consistent with respect to our definition. When the policy has for instance an infinite derivation with source q, while all others lead to one decision d in D (as in Figure 4.1(c)), the policy is consistent.

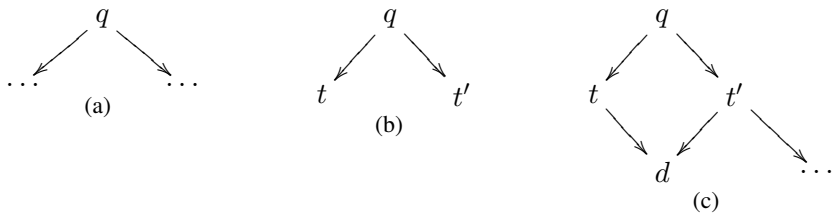


Figure 4.1: Examples of consistent policies.

Example 13. *Consider a car driver that is about to cross a traffic light (tl). Then the following policy must be respected:*

$$
\begin{array}{lrl}
\wp_1 = (& \mathcal{F}_1 & = \{tl : Color \rightarrow Decision, red, green, amber : Color, stop, go : Decision\}, \\
& D_1 & = \{stop, go\}, \\
& R_1 & = \left\{ \begin{array}{ll} tl(red) & \rightarrow stop, \\ tl(green) & \rightarrow go, \\ tl(amber) & \rightarrow go, \\ tl(amber) & \rightarrow stop \end{array} \right. \\
& Q_1 & = tl(\mathcal{T}(\mathcal{F}_1)), \\
& \zeta_1 & = \texttt{universal}(R))
\end{array}
$$

Then $\wp_1$ is not a consistent policy, since $[\zeta_1](tl(amber)) = \{stop, go\}$. However, an alternative strategy, ζ_2, yielding a consistent policy is to execute the rules in the order they appear. In this case, $[\zeta_2](tl(amber)) = go$, which may be an strategy for drivers in a hurry. Prudent drivers would prefer the strategy ζ_3, which gives priority to the last rule, obtaining a consistent policy where $[\zeta_3](tl(amber)) = stop$.

If we consider the `universal` strategy, confluence under strategy (Definition 39) reduces to the usual notion (Definition 27). Therefore:

Proposition 1. *A policy $(\mathcal{F}, D, R, Q, \texttt{universal}(R))$ is consistent if R is confluent on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, and all elements in D are irreducible.*

Proof. Assume R confluent and that terms in the set of decisions are irreducible. Let us show that the associated policy is consistent: if q rewrites to two *distinct* decisions d_1 and d_2, then from the hypothesis that R is confluent, d_1 and d_2 must reduce to a same term, which is impossible because they are different and irreducible by R. □

If we relax the hypothesis of the irreducibility of the terms in the set of decisions, it is possible to build policies whose underlying rewrite system is confluent, but which can generate inconsistent policies. This situation is illustrated in Figure 4.2. If d_1 and d_2 are decisions, it is possible to provide an strategy for which there are two possible decisions for the request q, for instance the strategy whose set of derivations is $\{q \twoheadrightarrow d_1, q \twoheadrightarrow d_2\}$. This would be an unusual situation involving both non-termination and reducible decision terms.

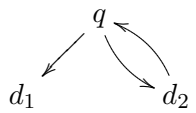


Figure 4.2: A confluent, but inconsistent policy.

Our definition of consistency is equivalent to the notion of *consistent regulation* in [Cholvy and Cuppens, 1997], and to the definition of the deterministic access control models in [Tschantz and Krishnamurthi, 2006]. However, in this latter work, an access control language is considered deterministic if all policies defined in such language are consistent, thus the model we propose here is not itself deterministic, because it allows one to define inconsistent policies. Our goal is to let the user specify the access control rules and then to use well known techniques in term rewriting, like testing the joinability of critical pairs, to prove the confluence of the rewrite system.

Termination

An essential analysis question is to answer whether a policy will ever return a decision for a given access request. This is an extremely important question, since policy rules often involve some computation (or deduction) prior to taking a decision and such process should be finite. We state the following fundamental property:

Definition 44 (Termination)**.** *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is* terminating *if for every q in Q, all derivations of source q in ζ are finite.*

Termination is surely an issue for rewriting based policies when compared to classic Datalog (without negation or constraints), for which all programs terminate. However, rewriting empowered with user-defined strategies provide a lot of flexibility in policy specification, where terms can have a "deep" structure, in contrast with Datalog. In the case of logic programming, as in Prolog, the termination is a problem to be worried about, where more and more techniques developed for checking termination in term rewriting are being applied to check termination of logic programs, see [Schneider-Kamp et al., 2007], for example. Other policy languages can have loops, such as Polymer [Bauer et al., 2005], thus termination analysis is needed.

Example 14. *The policies from Examples 10 and 11 are terminating and confluent, which can be easily checked by analyzing the rules in each policy. This guarantees that the evaluation of any request will return a unique decision.*

Example 15. *Consider the policy:*

$$\begin{array}{lll} \wp_2 = (& \mathcal{F}_2 & = \{a : Decision, permit : Decision, deny : Decision\}, \\ & D_2 & = \{permit, deny\}, \\ & R_2 & = \left\{ \begin{array}{l} a \rightarrow a \\ a \rightarrow deny \end{array} \right. \\ & Q_2 & = \{a\}, \\ & \zeta_2 & = \texttt{universal}(R) \\ ) & & \end{array}$$

$\wp_2$ is a security policy. But, in contrast to the previous examples, this policy is consistent (since the corresponding rewrite relation is confluent and decisions are in normal form), but it is not terminating.

Some simple sufficient conditions allow us to apply termination results from rewrite theory:

Proposition 2. *A policy $(\mathcal{F}, D, R, Q, \zeta)$ terminates provided that all derivations in ζ are finite.*

Proof. Trivial. If all derivations in ζ are finite for any term source of a derivation, then all derivations whose source is some q in Q are also finite. □

A similar result can be stablished when the problem reduces to the general termination problem, *i.e.*, no restrictions on derivations are imposed.

Proposition 3. *A policy $(\mathcal{F}, D, R, Q, \texttt{universal}(R))$ is terminating if R is strongly terminating.*

Proof. Trivial. If all derivations in R are finite, then all derivations starting with some q in Q are also finite. □

Proposition 3 permits to apply quite powerful techniques to check termination, such as recursive path orderings [Dershowitz, 1987], semantic labeling [Zantema, 1995], dependency pairs [Arts and Giesl, 2000], etc. These and other techniques have been implemented by several tools that can verify termination for a large set of rewrite systems: for example AProVe [Giesl et al., 2004], TTT [Hirokawa and Middeldorp, 2007], and CiME [Marché and Urbain, 2004], to cite a few. All these verification methods and tools check termination statically, which is very attractive for the specification of flexible policies.

Proving termination brings yet another advantage: for terminating rewrite systems, local confluence implies confluence (this result is known as Newmann's lemma). What is interesting for checking the consistency of policies is that in this case, the process can be mechanized via the completion algorithm [Knuth and Bendix, 1970]. When we consider policies with no specific strategy, we can inherit these sufficient conditions for the confluence and termination properties. However, we may use the finer notion of user defined strategies, which opens new research perspectives to establish sufficient conditions in the case of richer strategies.

Termination and confluence of a given term rewriting system assure the uniqueness of normal forms. Such property tells us that the evaluation of any request by a terminating and consistent policy will return an unique decision.

Decision Completeness

A further important policy property is the capacity to evaluate every incoming request into at least one decision. We call this property "decision completeness" for avoiding any confusion with the term "completeness" from a logical standpoint, or "sufficient completeness" in rewriting. This property is called totality in [Tschantz and Krishnamurthi, 2006] and [Barker and Fernández, 2006]. We did not adopt this word either because it reminds the notion of total functions.

Definition 45 (Decision Completeness)**.** *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is decision complete if $\forall q \in Q, \exists d \in D$, such that $[\zeta](q) \xrightarrow{*} d$.*

In other words, for any term in the set of requests, the evaluation of this term following the rewrite strategy defined by the policy will result in a term in the set of decisions.

Decision completeness in rewrite-based policies corresponds to the classical notion of sufficient completeness of a rewrite system. It states that every ground term evaluates to a term exclusively built with constructors (and possibly variables) [Comon, 1986, Kapur et al., 1991]. Several algorithms have been developed to check sufficient completeness, or to complete a set of patterns as to ensure this property [Bouhoula, 1994]. In [Gnaedig and Kirchner, 2006] authors used an alternative method to ensure completeness, notably it is possible to reason about weakly terminating systems.

Proposition 4. *A policy $(\mathcal{F}, D, R, Q, \texttt{universal}(R))$ is decision complete if R is sufficiently complete with respect to a set of constructors $\mathcal{C}$ and if the set of constructor terms $\mathcal{T}(\mathcal{C})$ is contained in the set of decisions D.*

Proof. Assume that $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$, where $\mathcal{D}$ is a set of defined symbols and $\mathcal{C}$ is a set of construtors. Then, by definition we have that $\mathcal{D} \cap \mathcal{C} = \emptyset$. If a request is a term containing defined symbols, then it is reducible by R to a constructor term in $\mathcal{T}(\mathcal{C})$. Since $\mathcal{T}(\mathcal{C}) \subseteq D$, the policy is complete.

□

Our definition of decision completeness is a particular case for the general sufficient completeness problem. In our case, it is necessary to check whether the reached normal forms are from the set of decisions.

The example below is decision complete. All input requests consist in terms rooted by the defined symbol $auth$. They are evaluated into access decisions, which are just constants (constructors). It suffices to examine the rewrite rules to infer that all input terms are reducible, and that the output term is either $permit$, $deny$, or na.

Example 16. *A simple example, inspired from [Barker and Fernández, 2006], illustrates the assignment of authorizations based on a "user id" which is represented by a natural number: all requests from a user whose "id" is greater than* 3 *are denied.*

- *Let the policy signature be:* $\mathcal{F} = \{0 : Nat, s : Nat \rightarrow Nat, + : Nat \times Nat \rightarrow Nat, auth : Nat \rightarrow A, permit : A, na : A, deny : A\}$
- *The set of of constant symbols representing decisions is* $D = \{permit, na, deny\}$
- *Consider* R *as the following set of rules (the operator* s *gives the successor of a number,* $+$ *is the usual sum operator,* x, y *are variables of sort* Nat*):*

$$\begin{cases} x + s(y) & \rightarrow & s(x + y) \\ x + 0 & \rightarrow & x \\ auth(0) & \rightarrow & permit \\ auth(s(0)) & \rightarrow & permit \\ auth(s(s(0))) & \rightarrow & na \\ auth(s(s(s(x)))) & \rightarrow & deny \end{cases}$$

- *the set* Q *contains ground terms with top symbol* $auth$*;*
- $\zeta = \texttt{universal}(R)$.

An example of request evaluation is: $[\zeta](auth(s(0) + s(s(s(0))))) = \{deny\}$

Remark that returning the na decision does not mean that the policy is incomplete: it does provide an answer, which can be clearly overridden in the context of a compound policy, as we will see later in Chapter 6. We can consider assigning the na decision automatically in the case policy fails to reduce a given term, which is the approach followed in the XACML specification, but in our framework we prefer to include such "default" rules in the set of policy rules, such that we can readily apply the existing verification techniques [Bouhoula and Jacquemard, 2006].

Furthermore, rewrite strategies facilitate the task of specifying decision complete policies. For instance, in Example 10 the `choice` strategy guarantees that the default rule, which captures all unspecified cases, is applied only when all other rules fail.

The main goals of the work we presented in this section were to provide a formal semantics for an expressive policy language, able to support dynamic constraints, recursive policies, and

to have requests and decisions with structure; to provide a design, where it is possible to have maintainable enforcement mechanisms, as we will see in Chapter 6; and also, to characterize policies properties by associating properties of the corresponding rewrite system that implements a given security policy – consistent, terminating and complete specifications provide a high degree of confidence to the policy author.

Unfortunately, all these properties are undecidable in general. We give in the next section some alternatives for considering restricted classes of rewrite systems with decidable properties.

4.4 Decidable Classes

One of the advantages of using rewrite systems in policy specification is the possibility to describe complex (recursive) functions that may be involved in the computation of access decisions. On the other hand, when we use the full expressive power that rewriting and strategies deliver, we have to take into account that almost all important properties for policies become undecidable. Therefore, it is important to identify some suitable settings for which decision procedures exist to check consistency, decision completeness, and termination. The specification of policies following some restricted syntax for the rewrite rules would give some guarantees in this sense, rather than in the general framework presented in the previous section.

The rest of the section is organized as follows. We survey in the first place decidability results for rewrite systems, where we will see that syntactical classes of rewrite systems (like, shallow, flat, or linear systems) may have nice properties, in particular, under some fixed strategy, *i.e.*, innermost, outermost, etc. Then we present an alternative approach for finding a decidable setting for rewrite-based policies, by creating a correspondence between Datalog programs and rewrite systems, which have the same properties as their Datalog equivalents.

Syntactical Classes of Term Rewriting Systems

We present a collection of decidability results (and also complexity, when available) of the termination, confluence, reachability, and sufficient completeness problems for rewrite systems. Several works present simple, syntactically restricted classes of rewrite systems with decidable properties. These restrictions are close to those adopted for Datalog programs, like shallow rewrite systems, for which variables appear at maximal depth of one in a term. Decidability results for other properties are summarized in the end of the section.

The results listed below are surveyed in [Godoy et al., 2007]. Termination is decidable for:

- ground rewrite systems in polynomial time,
- right-ground rewrite systems,
- rewrite systems with right-variable rules (rules whose right hand side is a single variable),
- flat rewrite systems under an innermost strategy, and
- right-shallow linear rewrite systems, as well under the innermost strategy.

Therefore, it is a positive fact that in some of our policy examples, *e.g.* Examples 10 and 13, all rewrite rules have right-hand sides composed of simple constants, $permit$, $deny$, etc.

Decidability results for other properties are summarized as follows:

- Confluence is decidable in the following cases (surveyed in [Godoy and Tiwari, 2005a]):
  - for ground rewrite systems,
  - for strongly terminating rewrite systems,
  - for shallow linear rewrite systems (both right-hand side and left-hand side are linear), and
  - for shallow and right-linear rewrite systems, more general than the previous item.
- Reachability and joinability can be decided
  - for linear rewrite systems [Godoy et al., 2004],
  - for shallow rewrite systems assuming that the innermost strategy is used [Godoy and Huntingford, 2007], and
  - for strongly terminating systems, given a strongly terminating system R and two terms s and t, can we decide whether $s \rightarrow_R^* t$ by rewriting a term s into all its possible reduced forms and comparing them to t.
- Sufficient completeness is decidable for terminating and confluent term rewriting systems [Bouhoula and Jacquemard, 2006], and the complexity of the decision procedures varies with the kind of rewrite system in question, ranging from NP-complete for free constructor systems to exp-time on linear systems [Kapur et al., 1991].

We grouped these known results in Table 4.1. The "-" mark means that the result is not known, as far a we know. From Table 4.1 we can conclude that an interesting syntactical class of rewrite systems, to be applied in security policy specification (and maybe in other domains as well), are ground rewrite systems.

Clearly this is too restrictive for policies in general. Nevertheless, ground rewrite systems can be very useful for verifying systems that enforce a certain policy, deployed in an ad-hoc manner. For example, consider an administrator of a UNIX environment. There are several combinations of groups, roles, and access control lists in the file system that can lead to inconsistencies: for instance, one can explicitly deny the access for a given group of users, but a member of that group can log in using a role for which the access is granted. This kind of verification can be achieved by abstracting the existing system configuration as a ground rewrite system (with a ground rule for each permission), and then, checking confluence over it.

Although Table 4.1 shows that most properties are decidable only for syntactically restricted classes of rewrite system, it does not mean that one will not be able to prove properties for a given rewrite-based policy outside of these classes.

Class	Property	Decidability	Complexity
Ground	Termination	*decidable*	*polynomial*
	Confluence	*decidable* [Dauchet et al., 1990]	*polynomial*
	Reachability	*decidable*	-
	Joinability	*decidable*	-
Flat	Termination	*undecidable* (*decidable* under innermost [Godoy et al., 2007])	-
	Confluence	*undecidable* [Jacquemard, 2003]	-
	Reachability	*undecidable* [Jacquemard, 2003]	-
	Joinability	*undecidable* [Jacquemard, 2003]	-
Shallow	Termination	*undecidable* [Godoy et al., 2007]	-
	Confluence	*decidable* if linear [Godoy et al., 2004]	*polynomial*
	Reachability	*decidable* under innermost [Godoy and Huntingford, 2007]	-
	Joinability	*decidable* under innermost [Godoy and Huntingford, 2007]	-
Right Shallow	Termination	*undecidable* (*decidable* if linear [Godoy and Tiwari, 2005b, Wang and Sakai, 2006, Godoy et al., 2007])	-
or Left Shallow	Confluence	-	-
	Reachability	-	-
	Joinability	-	-
General	Termination	*undecidable*	-
	Confluence	*undecidable*	-
	Reachability	*undecidable*	-
	Joinability	*undecidable*	-

Table 4.1: Decidable properties for rewrite systems

4.5 Relating Datalog and Term Rewriting

Datalog is a logic programming language originally used for performing deductive queries over relational databases. It has emerged during the eighties, and has been studied mainly by the community on database research. The main interest of Datalog is its flexibility and low complexity. It has been massively used in security for specifying access control [Fournet et al., 2007, Dougherty et al., 2006, Naldurg et al., 2006] and trust management [Winslett et al., 2005, Li and Mitchell, 2003]. The exposition about Datalog is an excerpt of [Ceri et al., 1989, Ceri et al., 1990]. In this section, we give a transformation from Datalog programs into an equivalent rewrite system. This translation preserves the advantages of Datalog: low complexity and termination. Our goal is not to simply simulate a Datalog program, but rather understand the characteristics of a class of rewrite programs that always terminate.

Preliminaries about Datalog

A *Datalog program* is a set of ground facts and rules, both represented as Horn clauses. The general form of a *Datalog rule* or *clause*, is the following:

$$H :- L_1, \ldots, L_n$$

each L_i is a literal of the form $p_i(t_1, \ldots, t_k)$, where p_i is a predicate symbol and all $t_1, \ldots, t_k$ are either constants or variables. The left hand side of a rule is called its *head*, whilst the right hand side is called its *body*. A clause with an empty body is called a *fact*. The set of literals that occur in the head of rules is called *Intentional Data Base* (IDB), while the set of ground facts is referred to as *Extensional Data Base* (EDB).

Datalog programs must fullfil the following restrictions:

- All facts are ground.
- Variables occuring in the head of a rule must also occur in its body.

These conditions ensure a very important property of such programs: given a finite set of ground clauses, the set of all facts that can be inferred by a Datalog program is finite.

Example 17. *The fact "John is the father of Bob" is represented as* $father(Bob, John)$*. The rule "if* X *is the father of* Y *and* Y *is the father of* Z*, then* X *is a grandparent of* Z*" is written as:*

$$gp(Z, X) :- par(Y, X), par(Z, Y)$$

The evaluation of a Datalog program corresponds to computing a fix-point on the set of new facts derived by the application of the rules. The evaluation stops when no new fact is added to the set of results. The semantics of Datalog can be defined as follows:

Definition 46 (Elementary Production Principle (EPP))**.** *Consider a Datalog rule the form* $H :- L_1, \ldots, L_n$ *and a set of ground facts* I*. If there exists a substitution* σ *such that for each* $1 \leq i \leq n$*,* $\sigma(L_i) = F_j$*, for some* F_j *in* I*, then we can infer in one step the ground fact* $\sigma(H)$*.*

The inferred fact can be either new or already known. Note that the definition above corresponds to matching each literal L_i modulo associativity and commutativity (See definitions 11 and 16) against the set of facts I.

Let Γ be a Datalog program. A ground fact F can be inferred from Γ, denoted $\Gamma \vdash F$, iff either $F \in \Gamma$, or F can be obtained by applying EPP a finite number of times. We formally define the relation $\vdash$ below.

Definition 47 (Datalog Semantics)**.** *The relation* $\vdash$ *is recursively defined as follows:*

- $\Gamma \vdash F$ *if* $F \in \Gamma$*;*
- $\Gamma \vdash F$ *if a rule* $c \in \Gamma$ *and ground facts* $F_1, \ldots, F_n$ *exist in* Γ*, such that* $\forall i, 1 \leq i \leq n$*,* $\Gamma \vdash F_i$*, and* F *can be inferred in one step by the application of EPP to* c *and* $F_1, \ldots, F_n$*.*

In order to distinguish among applications of EPP using distinct rules from a program, we may denote $\Gamma \vdash_c F$. Where c is a label identifying a given rule in Γ, and F is a ground fact derived from that application.

Since Datalog uses matching and not unification, its semantics differs from that of Prolog. The evaluation of Datalog programs does not depend on the order of the literals in a rule, nor on the order of the rules. Therefore, some valid Datalog programs would not terminate when run by a Prolog interpreter (see the case of the recursive program in Example 18).

Example 18. *In this example we show a program for computing the* same generation cousin *(sgc) from the following set of facts:*

$$\begin{aligned} person = \ & \{ann, bertrand, charles, dorothy, evelyn, fred, george, hilary\} \\ par = \ & \{(dorothy, george), (evelyn, george), (bertrand, dorothy) \\ & (ann, dorothy), (ann, hilary), (charles, evelyn)\} \end{aligned}$$

And the rules:

$$\begin{aligned} r_1 &: sgc(X, X) \ :- \ person(X) \\ r_2 &: sgc(X, Y) \ :- \ par(X, X1), sgc(X1, Y1), par(Y, Y1) \end{aligned}$$

Some facts generated by this program are:

$$sgc(dorothy, evelyn), sgc(charles, ann), sgc(ann, charles)$$

[Ceri et al., 1989] brings a simple saturation procedure to compute the set of infered facts from a Datalog program, which we reproduce below. The input is a finite set Γ of Datalog clauses and the output is the set of infered facts from Γ:

Algorithm 1.

```
Infer(Γ)=
BEGIN
    W←Γ;
       WHILE
          EPP applies to some rule and facts of W
          producting a new ground fact F ∉ W
       DO W←W ∪ {F};
       RETURN all facts of W, but not the rules
END
```

It is possible to input a Datalog program with a goal (a single literal with at least one variable). For example, one can query the result of $sgc(ann, X)$. Instead of computing the set of all derived facts, it is necessary to compute only the solutions that instantiate the variables in the goal. Thus the algorithm above is not efficient in this respect.

Transforming Datalog programs into TRSs

In this section we provide a transformation from Datalog programs into rewrite systems using a simple algorithm.

Definition 48 (Transformation)**.** *Let Γ be a set of Datalog clauses, it is transformed into the rewrite system R_Γ over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ through the function ϕ recursively defined as:*

- *For each constant c occurring in a ground fact in Γ, there is a constant $\phi(c) = c'$ where $c' \in \mathcal{F}$,*
- *For each variable x in a literal of Γ, $\phi(x) = x'$ where x' in $\mathcal{X}$,*
- *For each predicate p in a literal of Γ, there is a function symbol in p' in $\mathcal{F}$ associated by $\phi(p) = p'$, with the same arity,*
- *For each literal $p(t_1, \ldots, t_n)$ in Γ, $\phi(p(t_1, \ldots, t_n)) = \phi(p)(\phi(t), \ldots, \phi(t_n))$ where $t_1, \ldots, t_n$ are either constants or variables,*
- *For each set of literals appearing in a rule body of Γ of the form $\{L_1, \ldots, L_n\}$ associate a term $\phi(\{L_1, \ldots, L_n\}) = \phi(L_1) + \ldots + \phi(L_n)$, where $+$ is a new associative and commutative operator from $\mathcal{F}$,*
- *For each rule $H :- L_1, \ldots, L_n$ of Γ there is a rule in R_Γ:*

$$\begin{aligned} \phi(\{L_1, \ldots, L_n\}) + l \quad &\rightarrow \phi(H) + \phi(\{L_1, \ldots, L_n\}) + l \\ &\text{if } (\phi(H) \notin l) \rightarrow true \end{aligned}$$

  Where l is an extension variable,

- *The set of ground facts in Γ, $\{F_1, \ldots, F_n\}$ is transformed into a ground AC term with the operator $+$: $\phi(\{F_1, \ldots, F_n\}) = \phi(F_1) + \ldots + \phi(F_n)$.*

Then, given a Datalog program Γ containing a set of ground facts I, the transformation ϕ builds a conditional rewrite system R_Γ and a term $\phi(I)$ containing a list of ground facts, such that R_Γ implements the Algorithm 1 for Γ. When the rules of R_Γ are applied to the term containing the transformation of the ground facts of Γ, the output is the set of inferred facts for Γ.

Definition 49. *Let $\sigma = \{x_1 \mapsto c_1, \ldots, x_k \mapsto c_k\}$, be a substitution and $c_1, \ldots, c_n$ a set of constants. The substitution $\delta = \phi(\sigma)$ is defined as $\{\phi(x_1) \mapsto \phi(c_1), \ldots, \phi(x_k) \mapsto \phi(c_k)\}$.*

Proposition 5. *Consider a set of ground facts $\{F_1, \ldots, F_m\}$ and a set of literals $\{L_1, \ldots, L_n\}$. If there exists a substitution $\sigma = \{x_1 \mapsto c_1, \ldots, x_k \mapsto c_k\}$, such that of for each $L_i \in \{L_1, \ldots, L_n\}$, $\sigma(L_i) = F_j$ for some $F_j \in \{F_1, \ldots, F_m\}$. Then, the substitution $\delta = \phi(\sigma)$ is a solution for matching $\phi(L_1) + \ldots + \phi(L_n) + l$ against $F_1 + \ldots + F_m$.*

Proof. Follows from the associativity and commutativity of $+$. □

Theorem 3 (Correctness)**.** *Let Γ be a Datalog program, containing a set of ground facts I. Then, for the transformation $\phi(\Gamma)$, obtaining the rewrite system R_Γ and the term $\phi(I)$, the following diagram commutes:*

$$\begin{array}{ccc} I & \xrightarrow{\vdash_c} & I \cup \sigma(H) \\ \phi \downarrow & \Updownarrow & \phi \downarrow \\ \phi(I) & \xrightarrow{\rightarrow_{\phi(c)}} & \phi(I) + \phi(\sigma(H)) \end{array}$$

Proof. Given a Datalog program Γ with ground facts I. We show that the rewrite relation $\rightarrow_{R_\Gamma}$ induced by the rewrite system R_Γ, obtained by the transformation $\phi(\Gamma)$ is equivalent to $\vdash$.

Consider a rule c from Γ of the form $H :- L_1, \ldots, L_n$. If there is a substitution σ such that the new ground fact $\sigma(H)$ is derived from the application of EPP to c and I. Then, we have from Proposition 5 that the corresponding rewrite rule, $\phi(c)$ will reduce the term $\phi(I)$ with the substitution $\phi(\sigma)$, producing the result $\phi(\sigma(H)) + \phi(I)$. From the condition in each rule we have that $\phi(\sigma(H))$ will be produced only once. Therefore, there is a rewrite step in a derivation of R_Γ over $\phi(I)$ with the rule $\phi(c)$ iff there is an application of EPP to a rule c in Γ and facts in I. □

Theorem 4 (Termination)**.** *Let Γ be a Datalog program. The rewrite system R_Γ obtained by $\phi(\Gamma)$ terminates.*

Proof. From Proposition 5 and Theorem 3 we have that from the termination R_Γ follows from the termination of Γ .

□

In the following we show the result of the transformation of some Datalog programs into rewrite systems using the transformation described above.

Example 19. *The transformation of the Datalog program from Example 17 results in the rewrite system below:*

$$\begin{array}{l} par(Y,X) + par(Z,Y) + l \rightarrow gp(Z,X) + par(Y,X) + par(Z,Y) + l \\ \textit{if } gp(Z,X) \in l \rightarrow false \end{array}$$

Example 20. *The transformation of the "same generation cousin" program from Example 18 results in the following rewrite system :*

$$\begin{array}{ll} person(X) + l & \rightarrow sgc(X,X) + person(X) + l \\ \textit{if } sgc(X,X) \in l \rightarrow false & \\ & \\ par(X,X1) + sgc(X1,Y1) + par(Y,Y1) + l & \rightarrow sgc(X,Y) + par(X,X1) + sgc(X1,Y1) \\ & \quad + par(Y,Y1) + l \\ \textit{if } sgc(X,Y) \in l \rightarrow false & \end{array}$$

Example 21. *The security policy from [Dougherty et al., 2006] regulates access for a conference management system, where the rules state that:*

1. *During the submission phase, an author may submit a paper;*
2. *During the review phase, a reviewer r may submit a review for paper p if r is assigned to review p;*
3. *During the meeting phase, a reviewer r can read the scores for paper p if r has submitted a review for p;*
4. *Authors may never read scores.*

It is defined by the following rules in Datalog :

$$
\begin{array}{lll}
permit(A, sbmtPaper, P) & :- & paper(P), author(A) \\
permit(R, sbmtRvw, P) & :- & rev(P), paper(P), assigned(R, P), phase(review) \\
permit(R, readScores, P) & :- & rev(P), paper(P), hasRevwed(R, P), phase(meeting) \\
deny(A, readScores, P) & :- & paper(P), author(A)
\end{array}
$$

This program is tranformed by ϕ in the following rewrite system :

$$
\begin{array}{ll}
paper(P) + author(A) + l & \rightarrow permit(A, sbmtPaper, P) \\
 & \quad + paper(P) + author(A) + l \\
\text{if } permit(A, sbmtPaper, P) \in l \rightarrow false & \\
 & \\
paper(P) + author(A) + l & \rightarrow deny(A, readScores, P) \\
 & \quad + paper(P) + author(A) + l \\
\text{if } deny(A, readScores, P) \in l \rightarrow false & \\
 & \\
rev(P) + paper(P) + assigned(R, P) + phase(review) + l & \rightarrow permit(R, sbmtRvw, P) \\
 & \quad + rev(P) + paper(P) \\
 & \quad + assigned(R, P) \\
 & \quad + phase(review) + l \\
\text{if } permit(R, sbmtRvw, P) \in l \rightarrow false & \\
 & \\
rev(P) + paper(P) + hasRevwed(R, P) + phase(review) + l & \rightarrow permit(R, readScores, P) \\
 & \quad + rev(P) + paper(P) \\
 & \quad + hasRevwed(R, P) \\
 & \quad + phase(review) + l \\
\text{if } permit(R, readScores, P) \in l \rightarrow false &
\end{array}
$$

Starting from a database of facts such as the one below:

$$I = \{paper(1), author(a), author(b), phase(submission)\}$$

the execution of these rewrite rules will produce:

$$
\begin{array}{ll}
I' = & \{paper(1), author(a), author(b), phase(submission), deny(a, readScores, 1), \\
 & deny(b, readScores, 1), permit(a, sbmtPaper, 1), permit(b, sbmtPaper, 1)\}
\end{array}
$$

The advantages of such a translation are: it computes a fix-point over an AC term, in the same manner as in Datalog. The results obtained are the same ground facts that would be generated by the corresponding Datalog program. The transformation we have given does not take goal evaluation into account. This can be achieved with the application of a transformation-based method to compute goals in Datalog, such as *magic sets* [Bancilhon et al., 1986], prior to the translation of the program into a rewrite system.

The complexity of the execution of the resulting rewrite system may be equivalent to that of the original Datalog program. The condition in each rule correspond to the condition in Algorithm 1, thus we do not introduce an extra overhead, since the same problem exists in the execution of Datalog programs: their evaluation can actually derive facts that already exist in the EDB.

The concatenation of all facts as a very big term does not generate a performance problem either. The complexity analysis of Datalog programs does not take into account the cost of retrieving large database instances into the working memory. On the other hand, in rewrite systems implementations, like Tom [Balland et al., 2007], which we actually used to implement the examples in the current section, we make use of the maximal term sharing representation [Reilles, 2007] for improving efficiency.

4.6 Related Works

There are a lot of related works with respect to policy specification. In this section we will mainly concentrate on the rule-based ones. We start from other works that have employed rewriting in policy specification, and then we will move our attention to other rule-based approaches, which may adopt some variation of first-order logic, Datalog, or temporal logic, etc.

Rewrite-Based Initiatives

The works more closely related with the one described in this chapter are recent initiatives that introduce term rewriting into the specification of security policies. In [Echahed and Prost, 2005], term rewriting is used in the definition of the functions that control the confidentiality level of data, more specifically by describing how security levels are downgraded. The work concerns concurrent programs, whose formal model relies on a variant of process calculus.

In [Barker and Fernández, 2006], authors model access control lists and role based access control as term rewrite systems. They give an interpretation to the properties of a RBAC policy in the same direction as presented in [de Oliveira, 2007, Dougherty et al., 2007b], by characterizing termination, consistency, and completeness of policies with respect to the properties of the rewriting systems defining them. On the other hand, the model we presented here is more general, and allows the user to specify a reduction strategy. A second interesting approach for coding RBAC policies using constraint logic programming is given in [Barker and Stuckey, 2003].

In a recent paper [Bertolissi et al., 2007], rewrite systems are used to describe distributed policies, and presents a new model that generalizes RBAC by considering the concept events as the source of action control requests. Events embed the time notion, and can be formed by a sequence of other events. For dealing with distributed systems, they introduce function

annotations to distinguish in which node of a distributed system the function is defined.

Rule-Based Policy Specification

The process of creating rule-based languages for access control is, firstly, to choose a formal setting to base the new language. Usually this formal basis is a logic suitable to express the dynamic access control conditions for a given application domain, we may cite fragments of first-order logic [Halpern and Weissman, 2003], Datalog [Dougherty et al., 2006], deontic logic [Cholvy and Cuppens, 1997], defeasible logic [Lee et al., 2006], etc. Second, it is necessary to identify some properties of the chosen formal logic that will be important for policies in the given context, for example, to check decidability and complexity results. Third, to provide tools to support the development, and reasoning about policy specifications in the chosen underlying formal framework.

The first framework we describe is the one from Halpern and Weissmann [Halpern and Weissman, 2003]. In their work, they have proposed a fragment of first-order logic as policy specification language. The main point of their approach is that one is able to represent real world facts that affect policy decisions. In [Halpern and Weissman, 2003] a policy is a closed formula

$$\forall x_1, \ldots, \forall x_m (f \implies [\neg] Permitted(t, t'))$$

where f is a first-order formula, t and t' are terms of sort $Subject$ and $Action$, and where $[\neg]$ denotes an optional negation. Therefore, formulas are written in a vocabulary of sorted terms with some fixed elements, such as the predicate $Permitted$. It is possible, for example, to write access rules of the following form

$$\forall x, Permitted(x, sing) \implies Permitted(x, dance)$$

Remark that denials are expressed as negated permissions, which has important consequences for policy composition, as we will see in Chapter 6. In our work, we do not treat the failure to derive permission as denial. This is a crucial advantage for merging rules, since in purely logic-based works, there is no way to handle in the theory what happens when a policy which derives $deny$ for a request q is merged with another which derives $permit$ explicitly, for the same q.

Halpern and Weismann define the policy environment to be the set of true facts in a given instant of the system's execution, containing all kinds of simple statements like $Librarian(Alice)$. The problem in such model is that checking the validity of a formula is undecidable, even for function-free ones, which contain only the $Permitted$ predicate. However, they have identified a tractable subclass of first-order logic to deal with this question, which consists of a language with slightly different restrictions from those of Datalog. These constraints do not allow the use of equality, all variables appearing in a formula must appear in an occurrence of a $Permitted$ predicate, and policies should only have negated or non-negated occurrences of $Permitted$, not both at the same time. Respecting these restrictions comes with the advantage of getting rid of inconsistencies, when conflicting access decisions can be produced for the same access request. Yet, this fragment of first-order logic is expressive enough to capture a wide-range of typically dynamic policies.

The language designed by Jajodia and colleagues in [Jajodia et al., 2001], known as Flexible Authorization Framework, FAF, which tries to balance expressivity and performance. They proposed a staged architecture that combines an access matrix, to grasp static privileges; propagation policies, which are rules governing how access decisions are derived; and conflict resolution operators, to disambiguate among conflicting policy decisions. The distinction of the policy specification phases in FAF suggests a methodology for policy development. In addition, respecting these separation in stages improves the performance of the system. Policies are first materialized, creating instances of each predicate which are used on the later phases. By using this schema, FAF guarantees a unique stable model semantics and polynomial time complexity.

Jajodia et al. [Jajodia et al., 2001] define different predicates for explicit access (from the static part) and for derived access, both with positive and negative authorizations. They are respectively

$$cando(o, s, (\pm)a)$$
$$dercando(o, s, (\pm)a)$$

with $s \in S, o \in O$, and $a \in A$, where actions are preceded by a sign $+$ or $-$ to assign a permission or a denial. Finally, there is a third predicate used to produce a final decision for the policy do with the same signature as the previous ones, with the distinction that the vocabulary for defining rules with do is extended with some special purpose predicates, such as *denials-takes-precedence, permissions-take-precedence, etc*. Therefore, users need to foresee possible conflicts and define an overriding policy to deal with them.

One of the main originalities of the language proposed in [Becker and Sewell, 2004a, Becker and Sewell, 2004b] is the ability to set different constraint domains, for the same base language (a variant of Datalog), such that the complexity of the language can be tuned for a given application. In addition to RBAC, the language also supports distributed policies. However, the language does not include prohibitions and there is no explicit support for policy composition.

The policy framework of [Kalam et al., 2003] combines concepts like roles, temporal authorizations, obligations, and recommendations. It introduces a new dimension to policy specification (in addition to roles and groups) that is useful for the exchange of policies between different entities. The concept of *organization* provides scopes to policies. This model is called Or-BAC (for Organization-Based Access Control). It is based on first-order logic, and its decidable subset based on Datalog [Miège, 2005]. However, one has to give up from decidability provided by Datalog, for using one of the several extensions of Or-BAC with theories (to deal with time, location, and other user defined constraints). Conflicts are solved by a combination of approaches [Cuppens et al., 2007, Miège, 2005]: it is possible in the model to assign priorities to decisions, and also one can adhere to conditions (or guidelines) which help to avoid conflicts. Or-BAC comes with an administrator model to determine the roles who can grant and revoke authorizations.

As we see in the summarized descriptions of the above initiatives, there are some strong candidates for general-purpose policy specification languages like, FAF and Or-BAC. They provide expressive rule-based formalisms for building policies, and are well adapted to solve most of the problems related to reasoning about policies.

In [Jaume and Morisset, 2006b, Morisset, 2007] authors propose a formalism to describe security policies as functions on state transitions. Systems must ensure that a given security pred-

icate defined by the policy is true along all their execution. In such framework, it is possible to distinguish the policy statement from its implementation, which is particularly interesting for comparing different implementations of the same policy. In [Tschantz and Krishnamurthi, 2006] a similar enterprise is taken. The work proposes a set of suitable properties a policy model or policy specification language should satisfy, such as determinism, totality, compositionality, etc. They compare different approaches following their formal model of policies, including XACML and the logic language of [Halpern and Weissman, 2003].

An important issue that is not covered yet in our work is distributed access control. The core problem in access control for distributed systems is that a principal may require access in any node of a network, and he/she needs to present adequate credentials. The difficulty is that policy definitions may be decentralized, and decisions may differ from one server to another in a given network [Abadi et al., 1993]. Moreover, access control decisions may also depend on the kind of channels used for communication, on the protocol used, etc. A second important problem in distributed policies is to correctly model delegation; it means to provide mechanisms for one principal to generate a credential for a second principal to act in his behalf, at the same time to assure that such credentials cannot be forged, what can be achieved through digitally signed proofs [Bauer et al., 2007].

Some policy models put forward the causes and consequences of authorizations with respect to the reachable states in a system [Becker and Nanz, 2007, Dougherty et al., 2006]. In our work we do not directly offer support to describe the interactions of policies and system states, or how the access control rules influence the updates of such states, although our model can support the specification of such rules. The work presented in [Becker and Nanz, 2007] goes beyond and brings a logic and a proof system to reason about the history of the allowed requests, whereas in [Dougherty et al., 2007b] it is shown that obligations also have states, and discusses how they are related to program runs, with suitable analysis concerning obligations.

There are other frameworks that were left out of this survey, in particular those dedicated to trust negotiation on distributed, open environments, like the work presented in [Winslett et al., 2005]. These frameworks can involve some functionalities for access control, but, their main purpose is to describe protocols where it is possible to specify how much information can be released in a communication session according to the trust that client and server assign to each other. Other works not mentioned here include frameworks dedicated to Digital Rights Management (DRM), or usage control [Park and Sandhu, 2004].

As we have seen in the discussion above, several initiatives adopt some logic to specify access control. Therefore a natural question is to compare the expressive power of (first-order) rewriting and (first-order) logic. Since both paradigms are Turing complete, from the theoretical point of view, decidability results are a concern in all both approaches. In the logic-based setting, authors point out some characterizations of decidable subsets of first order logic, for example in [Halpern and Weissman, 2003]. An advantage of term rewriting systems is the number of available efficient implementations, which allows fast prototyping, and the tools and methods that allows one to check properties like termination, for example.

As a matter of fact, rewriting is an interesting approach for defining policies in a declarative manner, and can suitably be viewed as an intermediate language for policies, and we hope our work can influence the design of new paradigms for security policies.

4.7 Conclusions

In this chapter we have introduced a framework to formally describe security policies based on term rewriting with strategies. The framework is general enough to capture various kinds of policies as we saw in the examples we presented. The originality of the approach lies on the freedom to identify the core elements that form policies — decisions, requests, access rules, etc — and on the ability to perform formal reasoning about access control, based on the properties of the underlying rewrite system, such as the absence of conflicts, completeness, and so on. The main concept used here is that of rewrite strategy: because strategies allow to have a fine control on rule application, we can easily model real world policies, with priorities among rules, default rules, etc. In addition, in Section 4.4 we have identified some necessary conditions where important properties are decidable. This is crucial for improving the confidence in a formal policy model, but which can be restrictive, since syntactical limitations are imposed.

This chapter represents the foundational part of the thesis. In the next chapters, we will rely on the formal model for policies introduced here to explain the remaining contributions of this thesis. In addition to expressivity, the adoption of strategies is useful for preserving properties under composition, and for the definition of the policy composition operators themselves, which will be subject of Chapter 6. The implementation of rewrite-based reference monitors in Chapter 7 also relies on this framework, and for verification of information flow properties in Chapter 5.

Future Perspectives

To conclude this chapter, we would like to point some future research directions related to the work presented here.

- It is necessary to instantiate our policy model to deal explicitly with distributed environments. Since term rewriting provides a lot of expressivity for policy specification, together with fast implementations, this can be a promising application for rewrite-based policies.
- We shall adapt our model for reasoning about integrity and availability policies. There are several subtleties concerning these security objectives. Integrity makes intensive use of cryptography, whereas the notion of time is essential to availability. Of course, the combination of term rewriting with the existing approaches may be fruitful.
- An interesting research direction is to apply narrowing to solve queries over rewrite-based policy specifications. Narrowing can tell the possible instantiations for a term that will fire some rule, thus helps to answer questions of the kind “what rules apply in this situation” or “what-if” questions for some kinds of request like, “what if a given user is assigned an additional role”, for example.
- We shall work towards a policy integrated development environment. In such environment it is possible to translate other models into the rewrite-based one and to perform uniform analysis for policies, term rewriting can do the job of intermediary language for policies.

Chapter 5

Policy Verification

We have seen in Chapter 4 that term rewriting allows us to provide a uniform semantics to policies, and to reason about some important properties, such as consistency and completeness. However, some policy models may require more specialized analysis. The security properties they require go beyond the general consistency and completeness goals, because having these two properties would not be enough to assure a given security objective.

An interesting case of such goals appears in information flow policies, where confidentiality is preserved through constraints on how information can be disseminated. Information flow is usually controlled by assigning every object a security class, also called a security label. Whenever information is copied from object x to object y, there is a corresponding information flow from the security class of x to the security class of y. A second interesting problem is safety in classic access control models, which can be encoded as a rewrite system and solved through a simple procedure, as we will see in this chapter.

This chapter is divided in two main parts. First, we develop a technique to detect information leakage in multilevel security policies in Section 5.1. In Section 5.2, we study the safety problem in the HRU model. We close the chapter with related works and conclusions, respectively in Sections 5.3 and 5.4.

5.1 Verifying Information Flow Policies

In this chapter we will consider an "object" as an abstract concept, defined informally as a container of information. Subjects can be understood as human users, but it is more meaningful to think of them as programs. Typical examples of objects are files and directories in an operating system. The main principle in information flow policies is to avoid information from higher security levels to be stored in objects with lower security levels. More generally, security-sensitive information should not be accessible to lower-privileged subjects. For instance, an information flow policy states that a subject must be denied to copy a sensitive file, such as /etc/shadow in a UNIX file system (which contains all encrypted passwords), to a file with less restrictive rights, even if a program run by root is trying to perform such operation.

The mechanisms employed in order to enforce this kind of policy involve discretionary access control, and a system-wide mandatory policy, stipulating that resources can be owned by individual users, but that the system is able to override any user permissions, under certain circumstances. More precisely, when principals try to violate the information-flow property. This is the approach followed by Bell and LaPadula in their seminal work on mandatory access control [Bell and LaPadula, 1974]. Nevertheless, it is very hard to prove by analyzing the access

control rules that a system starting on a valid configuration will reach only secure or valid states. This is in part due to the fact that typical information flow properties are expressed in terms of desirable read-write and write-read traces. Roughly speaking, since systems are abstracted as Turing machines, checking an information flow property corresponds to checking whether the machine enters a given state, which is in general undecidable.

In this section, we show that it is not enough to assume that the security policy specifying the authorized accesses is correctly defined based on these traces, because some models can allow some sequences of requests that will indirectly violate the policy. We propose a method to make explicit such flows in a given state of the system. We use rewrite rules both to describe the access control policy and to explore information flow over the reachable states of a system. We propose an analysis technique that can automatically identify information leakages which may invalidate an implementation of an information flow policy.

We apply our technique on well-known models in the computer security literature. We consider the security model proposed by McLean in [McLean, 1988], whose intention is to generalize the Bell and LaPadula model for dealing with joint access. Confidentiality is compromised when some simplifying assumptions are taken, which were made in order to relate the general framework for joint access with BLP, as shown in [Jaume and Morisset, 2006a].

This section is organized as follows. Sect. 5.1.1 presents the BLP model in more detail, accompanied by its specification in our framework. Sect. 5.1.2 introduces the McLean's model, together with the proof that, in this model, the relation among the security levels is not a total order. Then, in Sect. 5.1.3 we define an analysis algorithm to detect information leakages. The work described here appeared in [Morisset and de Oliveira, 2007].

5.1.1 The Bell and LaPadula Model

Bell and LaPadula have formalized the concept of mandatory access controls [Bell and LaPadula, 1974, Bell and LaPadula, 1996] originally targeted to the military domain. The essence of BLP is to augment discretionary access controls, represented in an access control matrix, with mandatory access controls to enforce information flow policies. Without loss of generality, we focus only on the mandatory part in this chapter, which is really specific to the Bell and LaPadula model. Therefore we do not describe how the contents of the access control matrix can be modified by subjects, but we consider that the current state of the system is exactly the set of access tuples the matrix contains. Also, the notation and precise formulation of the rules of BLP used here are substantially different from those of the original Bell-LaPadula work, since we use a more up-to-date interpretation of these concepts in line with our formalism.

In BLP the relation among security levels is a total ordering, where the most common classes are: $top\ secret \succeq secret \succeq confidential \succeq unclassified$. In the following, S is a set of subjects, O is a set of objects, and A is the set of actions containing $\{read, write\}$. The functions f_s and f_o give the corresponding security clearance for subjects and classification for objects. Their profile is

$$\begin{array}{lll} f_s: & Subject & \rightarrow Level \\ f_o: & Object & \rightarrow Level \end{array}$$

M is an access control matrix containing access tuples of the form $m(S, O, A)$. Then, the mandatory rules for the BLP model are:

$$\forall s \in S \; \forall o \in O \quad m(s, o, read) \in M \Rightarrow f_s(s) \succeq f_o(o) \tag{5.1}$$

$$\forall s \in S \; \forall o_1, o_2 \in O \quad m(s, o_1, read) \in M \wedge m(s, o_2, write) \in M \Rightarrow f_o(o_2) \succeq f_o(o_1) \tag{5.2}$$

Write access is assumed here as write only. The properties 5.1 and 5.2 are known respectively as *simple security property* and $\star$*-property*. Write access is interpreted here as write only.

From a practical point of view, the $\star$-property would not be applied to human users, but rather to programs. Human users are trusted not to leak information. A secret user can write an unclassified document because we assume that he or she will put only unclassified information in it. Programs, on the other hand, are not trusted because they can have embedded Trojan horses. The $\star$-property prohibits a program running at the secret level from writing to unclassified objects, even if it is permitted to do so by discretionary access controls. A user labeled secret who wishes to write an unclassified object must log in as an unclassified subject.

The specification of the Bell and LaPadula policy in our framework is the access control policy $P_{BLP} = \{\mathcal{F}, D, R, Q, \zeta\}$ where:

- $\mathcal{F}$ is the signature:

req	:	$Query \times Matrix$	$\rightarrow Decision$
s	:	$Id \times Level$	$\rightarrow S$
o	:	$Id \times Level$	$\rightarrow O$
q	:	$S \times O \times A$	$\rightarrow Query$
m	:	$S \times O \times A$	$\rightarrow Privilege$
$+$	:	$Privilege \times Privilege$	$\rightarrow Matrix$
$read,\ write$	:		$\rightarrow A$
$\top, I, \bot$	:		$\rightarrow Level$
$permit, deny$	:		$\rightarrow Decision$

- the set of decisions is $D = \{permit, deny\}$;
- the set of requests is any term rooted by the function symbol req;
- R is the set of conditional rewrite rules below, with the variables i_1, i_2, i_3 of sort *Id*, l_1, l_2, l_3 of sort $Level$, x of sort $Privilege$ and y of sort $Matrix$. The operator $+$ is assumed to be associative and commutative;

$$
\begin{array}{lll}
r_1: & req(q(s(i_1,l_1),o(i_2,l_2),read),x+m(s(i_1,l_1),o(i_3,l_3),write)) & \rightarrow deny \\
 & \text{if } (l_3 \succeq l_2) \rightarrow false & \\
 & & \\
r_2: & req(q(s(i_1,l_1),o(i_2,l_2),read),y) & \rightarrow deny \\
 & \text{if } (l_1 \succeq l_2) \rightarrow false & \\
 & & \\
r_3: & req(q(s(i_1,l_1),o(i_2,l_2),read),y) & \rightarrow permit \\
 & & \\
r_4: & req(q(s(i_1,l_1),o(i_2,l_2),write),x+m(s(i_1,l_1),o(i_3,l_3),read)) & \rightarrow deny \\
 & \text{if } (l_2 \succeq l_3) \rightarrow false & \\
 & & \\
r_5: & req(q(s(i_1,l_1),o(i_2,l_2),write),y) & \rightarrow deny \\
 & \text{if } (l_2 \succeq l_1) \rightarrow false & \\
 & & \\
r_6: & req(q(s(i_1,l_1),o(i_2,l_2),write),y) & \rightarrow permit
\end{array}
$$

- ζ is the strategy which applies the rules in the order they appear above.

The rule r_1 states that if a subject requests read access over an object $o(i_2, l_2)$, but such subject already has write access to another object in the system $o(i_3, l_3)$, then such request should be denied if the security level of l_2 is higher that the level of l_3. Rule r_2 denies access when read access over an object is requested by a subject with insufficient clearance. Rule r_3 allows read access if the previous two rules were not applied. Rules r_4 to r_6 handle write requests in a similar way. Whenever a request is permitted, the corresponding access tuple is added to the matrix.

5.1.2 The McLean Model

The algebra of security of McLean [McLean, 1988] is seen as a generalization of the Bell and LaPadula model. One of the concepts introduced in this algebra is *joint access*, which is the capability to express that a given task can be accomplished by a set of subjects only concurrently. In the military field, this allows to capture the idea that two individuals need to command at the same time a missile launch, for instance. Privileges are then represented as tuples of the form (S, o, x), where S is a non-empty set of subjects, o is an object and, x is an access mode.

In [McLean, 1988], the $\star$-security property was modified to take joint access into account. It is stated as follows, where M is an access control matrix:

> "*a state is $\star$-secure if for any (sets of) subjects S_1, S_2 and objects o_1, o_2, if $m(S_1, o_1, read) \in M$ and $m(S_2, o_2, write) \in M$ and the classification of o_1 dominates that of o_2, then $S_1 \cap S_2 = \emptyset$* "

A direct translation of this sentence leads to the following logical formula:

$$\begin{array}{l} \forall S_1, S_2\ \forall o_1, o_2 \in O \\ \qquad (m(S_1, o_1, read) \in M \wedge m(S_2, o_2, write) \in M \wedge f_o(o_1) \succ f_o(o_2)) \\ \Rightarrow \quad S_1 \cap S_2 = \emptyset \end{array}$$

note the use of the strict comparison above. It is necessary because $S_1 \neq S_2$. Such expression is logically equivalent (by contraposition) to:

$$\begin{array}{l} \forall S_1, S_2\ \forall o_1, o_2 \in O \\ \qquad (m(S_1, o_1, read) \in M \wedge m(S_2, o_2, write) \in M \wedge S_1 \cap S_2 \neq \emptyset) \\ \Rightarrow \quad \neg(f_o(o_1) \succ f_o(o_2)). \end{array} \tag{5.3}$$

When McLean generalized BLP by considering it as a particular case of the joint access model where every set of subjects S is reduced to a singleton $\{s\}$, he introduced incomparable elements in the lattice of security levels, where the property (5.3) is equivalent to:

$$\begin{array}{l} \forall s\ \forall o_1, o_2 \in O \\ m(\{s\}, o_1, read) \in M + m(\{s\}, o_2, write) \in M \Rightarrow \neg(f_o(o_1) \succ f_o(o_2)) \end{array} \tag{5.4}$$

Therefore, this property would be equivalent to the $\star$-property of BLP (5.2) only when the order among security levels is total, since only in this situation we have the following result:

$$\neg(f_o(o_1) \succ f_o(o_2)) \Leftrightarrow (f_o(o_2) \succ f_o(o_1))$$

In general, actual implementations of multilevel security models adopt partial orders to compare security levels, having non-linear lattice structures. For example, military systems often employ extra labels to annotate the security levels. This practice introduces "compartments", *e.g.* $\{secret, Navy\}$ and $\{secret, Army\}$. Thus, subjects with clearance $\{secret, Army\}$ may access objects with the same classification, but not those from the $\{secret, Navy\}$ class. Thus, the corresponding lattice of security levels has incomparable elements [Sandhu, 1993].

Therefore, in the McLean's interpretation of BLP, a subject will be able not only to write objects with a superior security classification than his, but also those objects which cannot be compared to its security clearance. We show that this formulation is not only less restrictive than the original BLP model, but may also allow information leaks to happen. We illustrate such situation with an example.

Example 22. *Consider a lattice $L = \{min, \bot, I, \top, max\}$, illustrated in Figure 5.1. We have that $\top \succeq \bot$; the element I is not comparable to $\bot$ or $\top$; and for all x in L, $max \succeq x \succeq min$. Then, let O be the set of objects $\{o_1, o_2, o_3\}$ and f_o the security function such that $f_o(o_1) = \bot$, $f_o(o_2) = I$, and $f_o(o_3) = \top$. Let S be the set of subjects $\{s_1, s_2\}$ and f_s the security function such that $f_s(s_1) = \top$ and $f_s(s_2) = I$.*

A state M_1, represented in Figure 5.2, containing the access tuples

$$M_1 = \{(s_1, o_3, read), (s_1, o_1, write)\}$$

would be clearly not secure, since $f_o(o_3) \succeq f_o(o_1)$, and so the high-level information contained in o_3 cannot flow to o_1.

Now consider a state M_2 as the access control matrix represented in Figure 5.3. It contains the following set of accesses tuples:

$$M_2 = \{(s_1, o_3, read), (s_1, o_2, write), (s_2, o_2, read), (s_2, o_1, write)\}$$

According to the McLean's policy, this state is secure since $\neg(f_o(o_3) \succeq f_o(o_2))$ and $\neg(f_o(o_2) \succeq f_o(o_1))$. This reveals a leak of information, since the state M_2 is secure despite the fact that the information contained in o_3 can be copied to o_1, through o_2. Such a weakness can be easily exploited by Trojan horses.

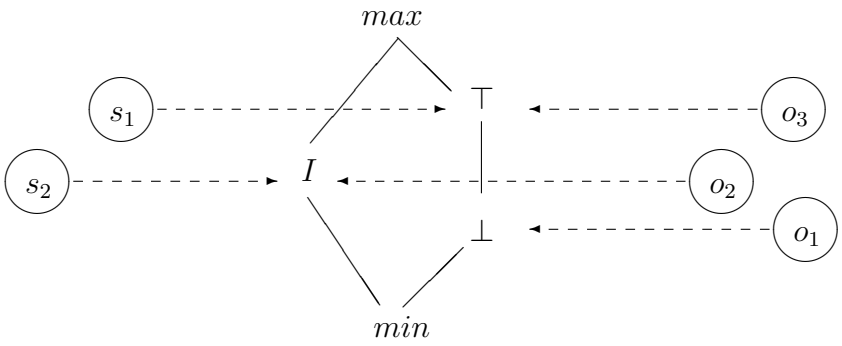


Figure 5.1: A lattice with a partial ordering.

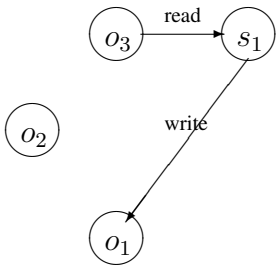


Figure 5.2: An insecure flow

With this new definition of the $\star$-property, the rewrite-based implementation of the McLean's model can be obtained from P_{BLP} , by replacing the rule (r_1) by

$$r_7: \quad req(q(s(i_1, l_1), o(i_2, l_2), read), x + m(s(i_1, l_1), o(i_3, l_3), write)) \quad \rightarrow deny$$
$$\text{if } (l_2 \succeq l_3) \rightarrow true$$

and the rule (r_4) by

$$r_8: \quad req(q(s(i_1, l_1), o(i_2, l_2), write), x + m(s(i_1, l_1), o(i_3, l_3), read)) \quad \rightarrow deny$$
$$\text{if } (l_3 \succeq l_2) \rightarrow true$$

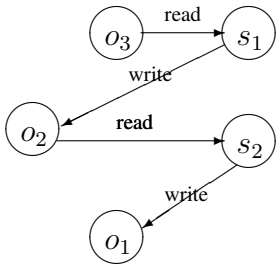


Figure 5.3: A secure flow

The next section presents a method for detecting such information leakages in rewrite-based policies.

5.1.3 An Analysis Algorithm for Information Flow

In this section we provide an algorithm, based on previous work on security protocol analysis [Cirstea et al., 2005]. The kind of vulnerability we search for in this work relies on the fact that some sequences of allowed $write$-$read$ actions may provide counter-examples for the overall security property.

An information flow is implicit when a subject is able to read an object, but the corresponding access tuple is not in the current access control matrix. An example of implicit information flow is the following situation: if a subject s_1 reads an object o_1 and writes to an object o_2, and if a distinct subject s_2 reads the object o_2, then one can consider that s_2 is also able to read the object o_1. In other words, if the accesses $(s_1, o_1, read)$, $(s_1, o_2, write)$ and $(s_2, o_2, read)$ belong to the set all privileges in the system, then the access tuple $(s_2, o_1, read)$ should also belong to this set.

The algorithm we propose here consists in making explicit every information flow in the configuration of a system, and to check whether this information flow is permitted by the security policy in place. When the policy denies this kind of access, then an information flow is characterized. If the policy allows the implicit access, then, according to its security model, there is no harm in this kind of flow.

Derived access tuples are made explicit by the application of a rewrite rule, which we call R', shown below. The variable M is an extension variable which matches all other element in a list of access tuples.

$$\begin{aligned} R' :\ & m(s_1, o_1, read) + m(s_1, o_2, write) + m(s_2, o_2, read) + M \quad \rightarrow \\ & m(s_1, o_1, read) + m(s_1, o_2, write) + m(s_2, o_2, read) + m(s_2, o_1, read) + M \\ & \text{if } m(s_2, o_1, read) \notin M \rightarrow true \end{aligned}$$

Note that the addition of a new access privilege in the access control matrix may lead to the creation of new implicit information flows. The process terminates because we only consider finite sets of accesses, and we avoid double entries in the matrix, thanks to the condition in the R' rule.

The algorithm we use to detect information leakages works as follows. Given an initial state, which is an access control matrix containing a number of access tuples, apply the rewrite system R', obtaining a set of derived privileges. For each new generated tuple – the implicit information flows – check whether the access control policy being tested allows it as a valid request. If the policy denies the request, an information flow is characterized. This is expressed in Algorithm 2 below.

Algorithm 2 (Verification of a set of accesses)**.**

```
Verification(M: set of accesses)=
    M_d←M;
    While [R'](M_d) ≠ fail Do
        M_d←[R'](M_d);
    End While;
    For each m(s,o,x) ∈ M_d \ M Do
        If req(q(s,o,x),M) →* deny Then
              Return false;
    End For;
    Return true;
End Verification;
```

The particularity to check the McLean's generalization of BLP lies in the different ways the initial configuration of the sets of accesses can be created given a set of subjects and objects. In fact, the input for our algorithm is simply the size of the instance to be checked – the number of subjects and objects. Then we generate different elements on each set, with different security levels – since it is uninteresting to check information flow when all elements in the system have the same labels. We reduce the search space in this manner, instead of giving all possible combinations of classifications to the different objects and subjects in the system and providing a breadth-first, or depth-first exploration strategy.

Algorithm 3 (Creation of accesses requests)**.**

```
Check(S: set of subjects,
      O: set of objects,
      R,R' : term rewriting systems)=
   M←∅;
   For each s ∈ S and each o ∈ O Do
       m_w←q(s,o,write);
       m_r←q(s,o,read);
       If req(m_w,M) →* permit Then
           M←m_w + M;
       If req(m_r,M) →* permit Then
```

```
            M←m_r + M;
         If Verification(M) = false Then
            Return "information leakage detected";
      End For;
      Return "no information leakage detected";
   End Check;
```

Algorithm 3 stops either because all combinations of requests that can lead to an information flow have been explored, or because the access control matrix contains an information flow. We start with an empty access control matrix and add access tuples from requests (to read and write) from high level subjects to low level objects. Each different order on the addition of access tuples in the matrix leads to a different set of reachable states to verify. This is a feature of the BLP model, since a permission received by a subject in the present restricts the possible actions it is allowed to perform in the future. Therefore, we also generate requests in the sequence that can lead to information leakage, as shown in Figure 5.3. We have implemented this algorithm in Tom (see Appendix A.1).

The different sets of accesses created by the Algorithm 3 depend on the "order" of the sets S and O. Indeed, let us consider for instance the sets $S = \{s_1, s_2, s_3\}$ and $O = \{o_1, o_2\}$ where subjects have distinct security levels from each other, and objects idem. If the loop `For` considers each subject and each object in this order, then the first requests created are $m_w = q(s_1, o_1, write)$ and $m_r = q(s_1, o_1, read)$. If these two requests are granted, then, in the following, the set of accesses M will contain the access tuples $m(s_1, o_1, write)$ and $m(s_1, o_1, read)$. Hence, the algorithm will not verify other sets of accesses which do not contain both of these tuples.

Since such sets of accesses can be reachable and respect the security policy, our algorithm does not explore the whole search space. Indeed, let us consider for instance that only the set of accesses:

$$m(s_2, o_1, read) + m(s_2, o_2, write) + m(s_3, o_2, read)$$

creates an information flow, and such set is secure and reachable. Let us also consider that the security policy is defined such that s_1 can read o_1, s_3 can read o_2 but these two accesses cannot appear at the same time. If the access $m(s_1, o_1, read)$ is first added, and never removed, then the access $m(s_3, o_2, read)$ will never be added, and so the information leakage is never detected. On the contrary, if the access $m(s_3, o_2, read)$ is added first, then the information leakage can be detected. In order to generate all possible instances of the problem, we need to call the function `Check` on every permutation of the sets S and O. For instance, if $S = \{s_1, s_2, s_3\}$ and $O = \{o_1, o_2\}$, we need to call the function `Check` as described in Figure 5.4.

Such method is clearly naive, because it leads to verifying several times the same set of accesses. In the case of the policy described in Section 5.1.2, information flow can be detected starting from two subjects and two objects. Provided that at least three different confidentiality levels exist, we can detect a leakage in four steps of the execution of the main loop, as depicted in Figure 5.5.

In this section we have proposed an analysis technique for detecting information flow violations in access control models. We have defined the state of the system as an access control

```
Check({s1, s2, s3}, {o1, o2});
Check({s1, s3, s2}, {o1, o2});
Check({s2, s1, s3}, {o1, o2});
Check({s2, s3, s1}, {o1, o2});
Check({s3, s1, s2}, {o1, o2});
Check({s3, s2, s1}, {o1, o2});
Check({s1, s2, s3}, {o2, o1});
Check({s1, s3, s2}, {o2, o1});
Check({s2, s1, s3}, {o2, o1});
Check({s2, s3, s1}, {o2, o1});
Check({s3, s1, s2}, {o2, o1});
Check({s3, s2, s1}, {o2, o1});
```

Figure 5.4: Calls to the `Check` function

matrix and the policy as a set of rewrite rules. The policy controls how new access tuples can be added to the matrix, like in the BLP model. The rule-based algorithm we have provided explores the space of possible requests in order to automatically identify information flow in these implementations. We illustrated the usefulness of our approach over the McLean's algebra of security for joint access, when generalized to the BLP model. We showed that it fails to prevent information leakage even on small configurations.

The technique we presented here can be adapted to check other access control models. It is only necessary to substitute the policy implementation, with the corresponding implementation for the new model. However, if this algorithm does not capture an information flow to a certain instance of a policy model, it does not mean that the implementation is secure with respect to the information flows.

5.2 Checking Safety in Rewriting-Based Policies

In this section we return to the issue of *safety* in access control models, originally raised by Harrison, Ruzzo and Ullman [Harrison et al., 1976]. The question is of great interest in access control because it concerns whether a certain configuration (a state of a system) leaks a privilege to an untrusted subject. In their seminal paper [Harrison et al., 1976], the authors provide a simplified set of operations, with limited computational complexity, to control how rights are assigned and revoked in an access control matrix. They have shown that safety is undecidable in general. But this does not mean that we cannot provide an algorithm for checking safety of a specific set of commands built from these operations and a given system configuration. In this section, we encode the HRU model in our rewrite-based framework, and analyze safety for some of its instances.

Our objectives are:

- to illustrate how this model can be encoded in our framework,
- to provide a running implementation of HRU that can be reused with other purposes,

Initialization.
Generation of the sets of $S = \{s(0, \top), s(1, I)\}$, and $O = \{o(0, \top), o(1, I)\}$.
The set of current accesses is empty: $M = \emptyset$
Step 1

Requests generated	Current accesses
$q(s(0, \top), o(0, \top), read)$	$m(s(0, \top), o(0, \top), read)$
$q(s(0, \top), o(0, \top), write)$.	$+ \quad m(s(0, \top), o(0, \top), write)$

$M_d = M \Rightarrow M_d \setminus M = \emptyset$ no information flow detected.

Step 2

Requests generated	Current accesses
$q(s(0, \top), o(1, I), read)$	$m(s(0, \top), o(1, I), write)$
$q(s(0, \top), o(1, I), write)$	$+ \quad m(s(0, \top), o(0, \top), read)$
	$+ \quad m(s(0, \top), o(0, \top), write)$

$M_d = M \Rightarrow M_d \setminus M = \emptyset$, no information flow detected.

Step 3

Requests generated	Current accesses
	$m(s(1, I), o(0, \top), write)$
$q(s(1, I), o(0, \top), read)$	$+ \quad m(s(0, \top), o(1, I), write)$
$q(s(1, I), o(0, \top), write)$	$+ \quad m(s(0, \top), o(0, \top), read)$
	$+ \quad m(s(0, \top), o(0, \top), write)$

$M_d = M \Rightarrow M_d \setminus M = \emptyset$, no information flow detected.

Step 4.

Requests generated	Current accesses
	$m(s(1, I), o(1, I), write)$
	$+ \quad m(s(1, I), o(1, I), read)$
$q(s(1, I), o(1, I), read)$	$+ \quad m(s(1, I), o(0, \top), write)$
$q(s(1, I), o(1, I), write)$	$+ \quad m(s(0, \top), o(1, I), write)$
	$+ \quad m(s(0, \top), o(0, \top), read)$
	$+ \quad m(s(0, \top), o(0, \top), write)$

$M_d = m(s(1, I), o(0, \top), read) + \ldots + m(s(0, \top), o(0, \top), write)$

$M_d \setminus M = m(s(1, I), o(0, \top), read)$. When we apply the transition rule to this request we obtain $req(q(s(1, I), o(0, \top), read), M) = deny$ which characterizes the information flow, then the execution stops.

Figure 5.5: Execution trace of the analysis algorithm

- to reason about elaborated properties such as safety using term rewriting techniques, and
- to identify the reasons for undecidability of the safety problem.

In the following we start with an exact specification of all capabilities of the HRU model using rewrite rules. Next we present a class of protection systems in the HRU model for which safety is decidable, and then we provide an algorithm to verify it on such class.

5.2.1 The HRU model as a Rewrite System

In HRU, a configuration of a system is an access control matrix with the particularity that every subject in the system is also an object (according to these assumptions, programs in operating systems are objects since they are also stored in files). Since matrices do not exactly have a term structure, we encoded it differently, as depicted in Figure 5.6.

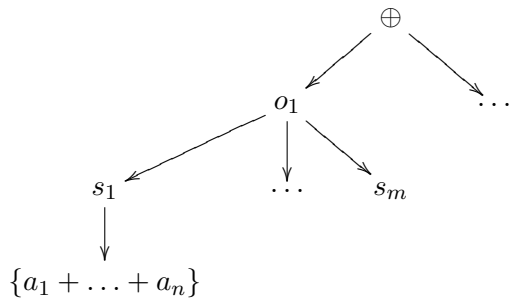


Figure 5.6: Term Representation of an access control matrix.

In our specification, a configuration is a term rooted by $\oplus$, which takes a list of objects. Objects have identifiers (a natural number, as in HRU) and a list of subjects, that by their turn also have an identifier. Each subject has a list of generic rights over the object they are attached to. The signature for these symbols is the following:

$$
\begin{array}{lll}
s & : Nat \times RightList & \rightarrow Subject \\
o & : Nat \times SubjectList & \rightarrow Resource \\
read, write, own, execute & : & \rightarrow Right \\
\oplus & : Resource \times Resource & \rightarrow ResourceList \\
\boxplus & : Subject \times Subject & \rightarrow SubjectList \\
+ & : Right \times Right & \rightarrow RightList
\end{array}
$$

A *protection system*, as defined in HRU, is composed of a set of generic rights and a finite set of commands, as we have briefly presented in Section 4.1. Each command is composed of a a list of conditions and a list of instructions. In our rewrite-based interpretation, commands are built from the following signature:

cmd	: $ConditionList \times InstructionList$	$\rightarrow Command$
$createSubject, createObject$	: Nat	$\rightarrow Instruction$
$destroySubject, destroyObject$	: Nat	$\rightarrow Instruction$
$enter, delete$	: $Right \times Nat \times Nat$	$\rightarrow Instruction$
in	: $Right \times Nat \times Nat$	$\rightarrow Condition$
$\wedge$	: $Condition \times Condition$	$\rightarrow ConditionList$
$\odot$	: $Instruction \times Instruction$	$\rightarrow InstructionList$
q	: $Command \times ResourceList$	$\rightarrow Configuration$
m	: $ResourceList$	$\rightarrow Configuration$

The operators $+, \oplus, \boxplus, \wedge, \odot$ are associative and commutative. Readers should refer to Appendix A.2 for the complete specification in Tom. The corresponding sorts for these operators also have a constant representing the empty list. In the rest of this section we overload the symbols for empty lists by representing them by the constant nil. As an example of how commands are encoded, consider the command $CONFER_{read}$ from Section 4.1:

Command $CONFER_{read}$ $(owner, friend, file)$
if $own \in M[owner, file]$
then **enter** $read$ **into** $[friend, file]$

Which specifies that a owner of a file can confer the read right over it to friends, is encoded as follows:

$$cmd((in(own, file, owner) \wedge nil), (enter(read, file, friend) \odot nil))$$

In the term above, $owner$, $friend$, and $file$ are simply variables of sort Nat representing the parameters in commands, as modeled in HRU. Parameter instantiation becomes an assignment of values of sort Nat to the variables contained in terms rooted by the symbol cmd. For instance suppose that the parameters are assigned the values $1, 2, 3$ in this order. Then, a call of this command $CONFER_{read}(1, 2, 3)$ is the term

$$cmd((in(own, 3, 1) \wedge nil), (enter(read, 3, 2) \odot nil))$$

We assume the instantiation of parameters to be a separate process, which needs to be performed prior to the execution of a command on a given configuration. As stated by HRU in their paper, all commands are ground instructions at run-time.

In the encoding we provide requests correspond to command calls, whose execution is performed by a set of rewrite rules that transform the current access control matrix. The rules implement the expected effect of primitive instructions (to create a new subject, or object, to enter a right in a given position of the matrix, etc), provided that the conditions on the command are satisfied. We separate the specification of the command execution in two parts. The first part tells us how to consume an instruction from the list of instructions of a command. The second part controls how the access control matrix is to be altered. The rules for the first part are given below, where the convention we use for naming variables is that x, y, i, j are of sort Nat, variables

named cl, sl, ol, etc stand for list variables of sort $ConditionList, SubjectList, ResourceList$ etc, respectively.

$$
\begin{array}{ll}
q(cmd(cl, il), ol) & \rightarrow m(ol) \\
\text{if } check(cl, ol) \rightarrow NotOK & \\
 & \\
q(cmd(cl, nil), ol) & \rightarrow m(ol) \\
 & \\
q(cmd(cl, createSubject(i) \odot il), ol) & \rightarrow q(cmd(OK, il), insSub(i, ol)) \\
\text{if } check(cl, ol) \rightarrow OK & \\
 & \\
q(cmd(cl, destroySubject(i) \odot il), ol) & \rightarrow q(cmd(cl, il), delSub(i, ol)) \\
\text{if } check(cl, ol) \rightarrow OK & \\
 & \\
q(cmd(cl, createObject(i) \odot il), ol) & \rightarrow q(cmd(cl, il), insObj(i, ol)) \\
\text{if } check(cl, ol) \rightarrow OK & \\
 & \\
q(cmd(cl, destroyObject(i) \odot il), ol) & \rightarrow q(cmd(cl, il), delObj(i, ol)) \\
\text{if } check(cl, ol) \rightarrow OK & \\
 & \\
q(cmd(cl, enter(r, i, j) \odot il), ol) & \rightarrow q(cmd(cl, il), insRight(r, i, j, ol)) \\
\text{if } check(cl, ol) \rightarrow OK & \\
 & \\
q(cmd(cl, delete(r, i, j) \odot il), ol) & \rightarrow q(cmd(cl, il), delRight(r, i, j, ol)) \\
\text{if } check(cl, ol) \rightarrow OK &
\end{array}
$$

The function $check$ performs a logical "and" on all elements of a list of conditions. A list is OK when all conditions in the list are true (correspondingly $NotOK$ when at least one condition in the list is false). Then, given a (ground) command c and a configuration ol, we obtain a new configuration ol' by applying the rules above, that is $q(c, ol) \xrightarrow{*} ol'$.

The first rule states that a configuration is not altered when at list one of the conditions in the list of the conditions is false. An application of the second rule will result in an configuration if all instructions in a command have been consumed. The remaining rules will execute some modification on the list of objects which depends on the current instruction.

The last set of rules in our specification manages how the matrix is to be modified by primitive operations. For illustration, we present a part of this set, with some functions to manage deletion of subjects from matrices. The rules for managing the creation and removal of objects are similar. The full set of rules can be found in the Tom implementation of the model, in Appendix A.2.

$$
\begin{array}{ll}
delSub(i, nil) & \rightarrow nil \\
delSub(i, o(i, sl) \oplus ol) & \rightarrow delSub(i, ol) \\
delSub(i, o(j, sl) \oplus ol) & \rightarrow o(j, delSub2(i, sl)) \oplus delSub(i, ol) \text{ if } i \neq j \rightarrow true
\end{array}
$$

$$
\begin{array}{ll}
delSub2(i, nil) & \rightarrow nil \\
delSub2(i, s(i, rl) \boxplus sl) & \rightarrow sl
\end{array}
$$

The rules above are explained as follows. The first rule returns an empty list if the list of objects is empty. The second rule states that when a subject is removed from the matrix, the corresponding object is also deleted, which is the behavior of this operation as defined in HRU. The function $delSub2$ is defined over lists of subjects, and deletes a subject with the index given in its first argument from the list. It is called in the third rule when the identifier of the subject being deleted is different from that of the current object.

5.2.2 Verifying Safety

In HRU, safety is the property assuring that given a protection system and a configuration, there is no future reachable state, obtained by the execution of a given command, that will leak a right. In our encoding, this property can be expressed in the following manner.

Suppose a configuration m and a command c, and that m' is obtained by rewriting the request

$$q(cmd(cl, i \odot il), m) \xrightarrow{*} q(cmd(cl, il), m')$$

where i is an primitive operation, cl is a list of conditions, and il is a list of instructions. We say that m' *leaks a right* r, if there is an object $o(x, sl)$ in m', where sl is a subject list, and a subject $s(y, r + rl))$ such that $check(in(r, x, y), m) \xrightarrow{*} NotOK$, for some r. Or, more simply, given a right $r \in M'[x, y]$ (in the standard notation) the question is to know whether it is in the same entry of the previous configuration of the matrix: $r \in M[x, y]$?

Given two configurations the function $safe(m', m)$, defined below, checks whether a right was leaked from a configuration m to a configuration m':

$$
\begin{array}{lll}
r_1: & safe(nil, ol) & \rightarrow OK \\
r_2: & safe(o(x, nil) \oplus ol, ol_2) & \rightarrow OK \\
r_3: & safe(o(x, s(y, nil) \boxplus sl) \oplus ol, ol_2) & \rightarrow safe(o(x, sl) \oplus ol, ol_2) \\
r_4: & safe(o(x, s(y, r + rl) \boxplus sl) \oplus ol, ol_2) & \rightarrow check(in(r, x, y), ol_2) \wedge \\
 & & \quad safe(o(x, s(y, rl) \boxplus sl) \oplus ol, ol_2)
\end{array}
$$

The rule r_1 says that if the configuration obtained from the execution of an instruction is empty, then it is safe. The rule r_2 establishes that if for a given object, there are no subjects that have some right over it, then for that object there is no leakage, and the safety test does not need

to check the remainder objects because when a subject is deleted from the matrix, it is suppressed from all resource lists. Whereas rule r_3 discards subjects with empty lists of rights. The rule r_4 builds a list of conditions to test whether each of the rights in the current configuration are present on the previous configuration of the matrix. When the result of the conjunction of the conditions in the list is false, then the configuration is not safe for one of the rights in a protection system.

Then, the algorithm for testing safety is: given a set of commands $\{c_1, \ldots, c_n\}$ and an initial configuration m_0, execute every permutation of the commands[1] in a given order, by reducing the term $q(c_i, m), 1 \leq i \leq n$; For every execution of a primitive operation, which result in a new matrix m', call to the function $safe(m', m)$. If the result of this call is $NotOK$ for a given pair (m', m) then stop with a leak. Otherwise, continue until all permutations have been tested. In this case, a protection system is safe for any given right r. Although costly, this algorithm allows to check safety for protection system in HRU. Note that in order to apply this algorithm, commands need to have their parameters instantiated because in our implementation, there are no variables in requests. This is the most important assumption we made in this development such that we are able to correctly compute the next configuration of a protection system at each step of the command execution. Then, verifying safety in this class of protection systems is possible.

The general problem in HRU is undecidable because it would be necessary to check every possible instantiation of the parameters in order to apply the procedure we presented above. Since subjects and object identifiers range over Nat, we have an infinite number of subjects and objects that can be created by different instantiations of the same command. Then, there is an infinite number of configurations for testing whether a right is leaked. In consequence, the procedure we gave above would not terminate.

5.3 Related Works

Reachability over term rewriting is a very general approach which constructs approximations of the set of normal forms with respect to a rewrite system, see for instance [Genet, 1998]. This is done through the use of tree automata. When verifying some security property, one tests for the reachability of terms representing unwanted situations. If such terms cannot be reached by an over-approximation of the set of derivations for a given rewrite system, then the security flaw will never occur in the actual system. This approach has been followed on the development of static analyzers [Boichut et al., 2007]. In our work, we use a simplified technique using the exact sets of normal forms, since computing the approximations is only necessary in a general framework for rewrite systems.

Static analysis techniques have also been applied to verify information flow inside programs. This consists in finding out every flow of data inside the code of a program, and then to avoid paths which violate the security policy. This was done, for instance, on some extension of λ-calculus [Heintze and Riecke, 1998], or on extensions of programming languages as in [Myers, 1999]. Most of these extensions include a sophisticated type system to cope with

[1] It is not necessary to change the order of the instructions inside commands.

how information can flow from protected variables with high security level to lower classified variables as surveyed in [Sabelfeld, 2003].

Some related works use similar verification techniques in the domain of access control policies. In [Naldurg et al., 2006] an approach close to ours is used for finding vulnerabilities through the use of rules to infer attacks from existing access control configurations in the Windows XP and SELinux operating systems. Several kinds of attacks are explored and the formalism used is a variation of Datalog with negation both to specify the policy and the vulnerabilities.

In [Guelev et al., 2004], the authors verify the satisfiability of a temporal logic formula in the policy model they presented, but it is not clear how they could address information flow. In [Dougherty et al., 2006], the authors suggest to perform goal reachability over Datalog programs as a way to compare access control policies, but information flow is not addressed as well.

In [Jaeger and Tidswell, 2001], access control is defined via binary relations on graphs, creating a model which is basically a derivation of RBAC. The complexity of the specifications is controlled with the help of some additional constraint types, such that the safety problem can be verified. Whereas in [Koch et al., 2002], safety is shown decidable for an access control model based on graphs, with some restrictions on the form of the graph transformation rules. The decidability results rely on existing theorems for the decidability properties for graph transformations. Authors show the usability of their approach on RBAC. In the work we have presented here, we have encoded the HRU model as a rewrite system and given an algorithm for checking safety on instantiated protection systems. The same approach can be developed without additional difficulties for other DAC models or RBAC.

5.4 Conclusions

Besides proving more fundamental properties like consistency, termination and completeness, security models often require to analyze properties related to the correctness of a given policy implementation. For information flow policies, this corresponds to checking whether there is any possible flow, implicit, or explicit, that violates the security principles of the model being considered. For access control, this is related to verifying whether there are other ways for obtaining privileges in the system without being given these rights explicitly, which characterizes the safety problem.

In this chapter, we have given rewrite-based procedures that can be applied to the verification of both of these problems. First we have developed a rule-based technique to verify information leakage in mandatory access control policies. We have shown in Section 5.1.3 that different implementations of a policy can allow for prohibited flows to happen. In Section 5.2.1, we have presented an approach for checking safety in the classic HRU model that allows one to verify this property at run-time.

A further development would be to consider new techniques for answering general queries over rewrite-based policies. For instance, one would ask whether access is always denied for a given resource. Further questions can be made to challenge a policy, including administrative queries, such as in change-impact analysis of dynamic policies, in the same line of other works like [Liu, 2007] and [Fisler et al., 2005].

Chapter 6

Policy Composition

It is natural to admit that rules describing the various policies in large organizations should not be authored and maintained in a single monolithic policy. It is useful to break down the complexity of large specifications in smaller rule sets. Smaller policies, which usually cover a few kinds of access requests, can be better understood and the proofs of their properties become easier. The need for modular policies appear in several practical situations, for instance, when different organizations (or different departments of the same organization) need to cooperate. Very often, the entities involved in the system have different, incompatible, or contradictory, requirements for access control, since they put different emphases on the security goals (what needs to be protected from whom). For example, in a hospital, there are rules governing the access of patients to their health records, their financial records, and the like, while at the same time, there are rules for employee access to these same records as well as to resources quite different from health records. Meanwhile, other entities such as insurance carriers are subject to yet another set of rules for access to these data.

Imagine that rules for patient data access and rules for staff data access are composed in separate policies, $\wp_p$ and $\wp_s$ respectively. What should we say about the decision of $\wp_p$ in the context of a request by an administrator to write a health record? Assuming $\wp_p$ will not explicitly compute a decision ($permit$ or $deny$) upon such a request, we must uniformly assume a *default* decision, perhaps a decision called $deny_{default}$, for all requests not handled directly. But this immediately leads to the conclusion that composing policies is something more subtle than taking their union. Consider by contrast a request by an administrator to read the next-of-kin information for a patient. A $deny_{default}$ produced by $\wp_p$ for this request would mean that, when $\wp_p$ was combined with $\wp_s$, which may explicitly compute a $permit$ for this request, the resulting logical theory, taken in a naive sense, would be contradictory.

In this setting, a *theory of composition* of policies becomes crucially important, in particular, when it is necessary to know under which conditions the individual security objectives can be maintained under composition.

We have introduced in Chapter 4 a model for rich security policies based on strategic rewriting. In this chapter, we show that this framework is not only well-suited for the specification of single policies, but it also supports their modular construction.

The contributions described in the current chapter aim to solve two central problems in policy composition:

- to provide mechanisms for combining policies with an uniform formal semantics: users can define how policies are to be combined through the definition of new functions and strategies for the combined policy; and

- to allow reasoning about how the interaction among the access rules of multiple policies being combined affects the general behavior of the global policy, in particular, whether the properties of the component policies are preserved.

We will see in this chapter that strategic rewriting is an expressive paradigm for defining various kinds of policy combiners – thanks to the capacity provided by strategies to combine rewrite rules, through the selection of the derivations of interest from the component systems. We experiment with various scenarios for policy composition in Section 6.2, such as the XACML-like policy combiners, in addition to sequential and conservative extensions. We also take advantage from existing results on the modularity of rewrite systems, in particular for policy consistency and termination properties.

6.1 Definitions

The definition presented below allows us to rely on the extensive results concerning modular properties of rewrite systems. It is compatible with Definition 42, in the sense that every policy combination is itself a policy. Without loosing generality, unless otherwise specified, we consider the pairwise combination of policies.

Definition 50 (Policy Composition)**.** *A* composition *of two policies* $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_i)$ *(*$i = 1, 2$*) is any policy* $\wp = (\mathcal{F},\ D,\ R,\ Q,\ \zeta)$*, where:*

1. $\mathcal{F}_1 \cup \mathcal{F}_2 \subseteq \mathcal{F}$*;*
2. $D_1 \cup D_2 \subseteq D \subseteq \mathcal{T}(\mathcal{F})$*;*
3. $R_1 \cup R_2 \subseteq R$*;*
4. $Q_1 \cup Q_2 \subseteq Q \subseteq \mathcal{T}(\mathcal{F})$*;*
5. ζ *is a rewrite strategy for* R*.*

We can make the following observations about this definition:

- When combining policies, it may be convenient to introduce symbols that do not occur in the original policies. This is why the signature $\mathcal{F}$ can contain more symbols than $\mathcal{F}_1 \cup \mathcal{F}_2$.
- The set of requests for the combined policy contains terms of the form determined by its sub-policies, but may also contain additional well-formed terms which can be constructed from the combined policy signature. For example, suppose that $\mathcal{F}_1 = \{0, f\}$, $Q_1 = f(\mathcal{T}(\mathcal{F}_1))$ and $\mathcal{F}_2 = \{g\}$, $Q_2 = g(\mathcal{T}(\mathcal{F}_2))$, then a valid request in the context of the combined policy would be $g(f(0))$.
- The combination strategy defines how the composed policy reduces request terms. Intuitively, it can be expressed as a functional composition of component strategies by using any strategy operator. However, it may not be built in a modular way, such that the new strategy disregards the strategies originally used in the sub-policies. This is useful in several situations, where it is necessary to give a more appropriate evaluation function for the

new set of request terms. Therefore, we provide in this definition the choice to reuse only part of the rewrite rules available from the component policies, and to freely build a new strategy for their combination.

- We give the definition of policy composition in terms of two policies, but of course the definition can be extended without problems to n policies by combining them by pairs.

In the next section we describe several examples for policy combiners.

6.2 Policy Combiners

In this section, we explore the formal semantics and properties of interesting policy combinations by using the strategic rewriting formalism. Examples are introduced along.

6.2.1 Simple Union of Policy Rules

The union of two policies consists in the simple union of the set of rewrite rules of the component policies. This kind of combination is well understood from the term rewriting perspective, and several results concerning the preservation of the properties of the combined systems are available. In this section, we will see examples of consistent and terminating policies whose union no longer satisfies these properties. Nevertheless, we present some syntactical conditions on the union operation, and strategies that preserve these properties for this kind of policy composition.

The next example is due to [Toyama, 1987a]. Although it does not represent any realistic policy, it illustrates the fact that much care must be taken in policy composition, even if the underlying rewrite systems being composed are both terminating and/or consistent.

Example 23. *Consider the policies $\wp_1$ and $\wp_2$ below.*

$$\begin{array}{rcl}
\wp_1 = (\quad \mathcal{F}_1 & = & \{g : Decision \times Decision \rightarrow Decision,\ na : Decision\}, \\
D_1 & = & \{na\}, \\
R_1 & = & \left\{ \begin{array}{l} g(x,y) \rightarrow x \\ g(x,y) \rightarrow y \end{array} \right. \\
Q_1 & = & g(\mathcal{T}(\mathcal{F}), \mathcal{T}(\mathcal{F})), \\
\zeta_1 & = & \texttt{universal}(R))
\end{array}$$

$$\begin{array}{rcl}
\wp_2 = \quad (\quad \mathcal{F}_2 & = & \{\ f : Decision \times Decision \times Decision \rightarrow Decision, \\
 & & \quad permit, deny : Decision\} \\
D_2 & = & \{permit, deny\} \\
R_2 & = & \left\{ \begin{array}{l} f(permit, deny, x) \rightarrow f(x,x,x) \\ f(deny, permit, x) \rightarrow f(x,x,x) \\ f(x,x,x) \qquad\qquad \rightarrow x \end{array} \right. \\
Q_2 & = & f(\mathcal{T}(\mathcal{F}_2), \mathcal{T}(\mathcal{F}_2), \mathcal{T}(\mathcal{F}_2)), \\
\zeta_2 & = & \texttt{universal}(R_2)\ )
\end{array}$$

The composition of $\wp_1$ and $\wp_2$, denoted $\wp$, is defined as:

$$\begin{array}{rll} \wp = & (\mathcal{F} = & \mathcal{F}_1 \cup \mathcal{F}_2, \\ & D = & D_1 \cup D_2, \\ & R = & R_1 \cup R_2, \\ & Q = & \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \\ & \zeta = & \mathtt{universal}(R)) \end{array}$$

Note that the signature for these policies are disjoint ($\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$), and that both policies are terminating (the collapsing rules in R_1 and R_2 guarantee the finiteness of the derivations). Despite of these properties, the composed policy is not terminating, which is justified by the following derivation:

$$\begin{array}{ll} f(\underline{g(permit, deny)}, g(permit.deny), g(permit, deny)) & \rightarrow_{R_1} \\ f(permit, \underline{g(permit, deny)}, g(permit, deny)) & \rightarrow_{R_1} \\ \underline{f(permit, deny, g(permit, deny))} & \rightarrow_{R_2} \\ f(g(permit, deny), g(permit.deny), g(permit, deny)) & \rightarrow \ldots \end{array}$$

A property is said to be *modular* if it is preserved under composition. In general the confluence and termination properties are not modular. However, several results concerning modularity for rewrite systems have been produced. An extensive survey is found in [Ohlebusch, 2002a]. Thanks to the work we have done in Chapter 4 and in this chapter to define access control policies, all knowledge from the theory of term rewriting can be imported to the context of policy composition.

Confluence is modular for disjoint rewrite systems [Toyama, 1987b]. Therefore we can state the following proposition about the combination of *consistent* policies.

Proposition 6. *The union $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, Q_1 \cup Q_2, \mathtt{universal}(R_1 \cup R_2))$ of two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \mathtt{universal}(R_i))$ ($i = 1, 2$), is consistent if:*

- *$\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$,*
- *R_1 and R_2 are confluent,*
- *the elements in D_1 are irreducible with respect to R_1, and the elements in D_2 are irreducible with respect to R_2.*

Proof. From the modularity of the confluence property on disjoint signatures, $R_1 \cup R_2$ is confluent. Since $\mathcal{F}_1$ and $\mathcal{F}_2$ are disjoint, each request q belongs to one Q_i and is only reducible by the corresponding R_i. From Proposition 1, each $\wp_i$ is consistent, and so is $\wp$ □

In other words, the disjoint union of two consistent policies is consistent under the `universal` strategy. This is a very useful result for policy composition, since it is perfectly safe with respect to consistency, to combine a policy that states positive rules only (rules that derive only permissions) with strictly negative rules (rules that derive only prohibitions), given that the remainder of their signatures are also disjoint. Moreover, if two policy signatures are not disjoint, it is possible to provide an appropriate renaming for the shared symbols.

Termination, in turn, is not a modular property even for disjoint rewrite systems [Toyama, 1987a]. However, adding syntactic conditions on the rewrite rules, or the existence of a simplification ordering, allows getting positive results [Rusinowitch, 1987, Middeldorp, 1989, Gramlich, 1992, Kurihara and Ohuchi, 1990].

Proposition 7. *The combination of two terminating policies* $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \mathtt{universal}(R_i))$ *(*$i = 1, 2$*), with disjoint signatures, given by* $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \mathtt{universal}(R))$ *is terminating if:*

1. *neither* R_1 *nor* R_2 *contain collapsing rules (Definition 29), or*
2. *neither* R_1 *nor* R_2 *contain duplicating rules (Definition 29), or*
3. R_1 *or* R_2 *contains neither collapsing rules nor duplicating rules, or*
4. *termination of* R_1 *and of* R_2 *are proved by a simplification ordering (Definition 22).*

The disjointness of signatures assumption in the previous results can be relaxed for constructor-sharing systems [Kurihara and Ohuchi, 1992], composable systems [Middeldorp and Toyama, 1991] or hierarchical combinations [Ohlebusch, 2002b] of rewrite systems which generalizes the previous ones by allowing some sharing of defined symbols [Dershowitz, 1994].

The interest of rewriting strategies appears again in policy composition. For instance, in contrast to termination, innermost termination has a nice modular behavior, for disjoint unions, constructor-sharing systems, composable systems, and for some hierarchical combinations. Given the results on innermost termination from [Gramlich, 1996] we can state the following:

Proposition 8. *Let* $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \mathtt{innermost}(R_i))$ *(*$i = 1, 2$*) be two policies such that* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *are disjoint or share only constructors, then their composition* $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \mathtt{innermost}(R_1 \cup R_2))$ *is terminating as soon as* $\wp_1$ *and* $\wp_2$ *are.*

Example 24. *Let us consider again the policies* $\wp_1$ *and* $\wp_2$ *from Example 23, but now with different strategies* $\zeta_1' = \mathtt{innermost}(R_1)$ *and* $\zeta_2' = \mathtt{innermost}(R_2)$ *respectively. Their combination* $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \mathtt{innermost}(R))$ *is terminating according to Proposition 8.*

Up to date, we are not aware of modularity results for the termination, confluence, and sufficient completeness properties in the case of user-defined strategies in their generality. These "local" strategies can employ the full expressivity of strategic rewriting, in contrast to well-know strategies for term traversals like *innermost, outermost*, and so on. This is an important research direction that deserves to be explored in the future. In the next sections we explore other strategy operators that can be of great interest for policy composition.

6.2.2 Sequential Policy Combination

A sequential policy combination captures the need to further evaluate intermediate decisions generated by some policy, using the rules from a second policy, in order to produce a final access

control decision. The final decision will be in normal form with respect to the second policy, and belongs to its set of decisions. Therefore, in this case, the decisions generated by the first policy work as input for the next one. We illustrate this situation with the following example, which is actually a policy used at Disneyland to control the access to its main attractions.

Example 25. *In order to reduce the waiting time for the Big Thunder Mountain or the Space Mountain, visitors can request a "fast pass" ticket. A scenario for its usage is as follows: suppose it is midday and a visitor does not want to stand on the queue. So this visitor can use the fast pass automated distribution machine, which will print a ticket for the next available time frame, say from* $2\ pm$ *to* $2:30\ pm$*. There is a determined number of time slots for each different attraction from opening time to closing time. Each time slot has a precise initial time, and a final time. The visitor has time to have lunch, and maybe to visit some other less popular attraction, and then at* $2\ pm$ *go to a separate entrance for fast pass ticket holders for the attraction he has chosen. On that entrance the ticket is checked: a visitor cannot enter if it is earlier than the time slot specified in the ticket, nor if it is later.*

Therefore, the first policy computes the intermediate access token based on the current flow of people entering a given attraction and on the number of places available for the next time slot, and in the case all places in all next slots have been allocated, the ticket is denied.

- *The signature* $\mathcal{F}_1$ *contain the following set of symbols:*

$$\begin{array}{lll} slot & : Nat \times Time \times Time & \rightarrow Slot \\ nil & : & \rightarrow SlotList \\ + & : Slot \times SlotList & \rightarrow SlotList \\ q_1 & : Time \times SlotList & \rightarrow Decision \\ fp & : Time \times Time & \rightarrow Decision \\ deny & : & \rightarrow Decision \end{array}$$

  For illustration purposes, we consider that the constructor $slot$ *takes as argument the number of available places for a given time slot, an initial time, and a final time for that slot. The* $Time$ *sort stands for some theory for dealing with time with predicates* $>$ *and* $<$ *with the usual semantics. The details of the time theory in question do not influence the rest of the example.*

- *The set of requests* Q_1 *contains terms with* q_1 *as head symbol, takins as arguments the current time, and the current slot list for some attraction, which are the environment of the policy.*

- *The set of decisions* D_1 *contains terms of the form* $fp : Time \times Time$*, which means that a visitor is allowed to enter the given attraction during the time frame in the ticket, or simply* $deny$*.*

- *The set of rewrite rules* R_1 *is defined as follows*

$$R_1 = \begin{cases} q_1(time, nil) & \rightarrow deny & \\ q_1(time, slot(n,i,f) + L) & \rightarrow fp(i,f) & \text{if } ((n > 0) \wedge (time < f)) \rightarrow true \\ q_1(time, slot(n,i,f) + L) & \rightarrow q_1(time, L) & \text{if } ((n = 0) \vee (time > f)) \rightarrow true \end{cases}$$

The variables occurring in the rules above are $time, i, f : Time$, standing respectively for the current time, the initial time for an slot, and the final time for that slot; $n : Nat$ represents the remaining number of places in an slot; and $L : SlotList$ is a list of slots. These rules perform a search for the next time slot that has not yet been exhausted and generate a ticket with the appropriate time interval. The first rule states that if all slots have been looked at, then the decision is $deny$. The second rule creates a fast pass, fp, if there are still places in a slot not already passed (the current time is earlier than the slot's final time). The third rule takes the next slot if the current slot in the list does not satisfy these conditions. Some additional rules for this part of the system would manage the time slots and their update.

- *The strategy is ζ_1 applies the rules in the given order.*

The second policy is defined over the signature $\mathcal{F}_2$ given below:

$$\begin{array}{lll} q_2 & : FastPass \times Time & \rightarrow Decision \\ fp & : Time \times Time & \rightarrow FastPass \\ permit, deny & : & \rightarrow Decision \end{array}$$

Its set of decisions is $D_2 = \{permit, deny\}$ and the set of requests Q_2 contains all terms rooted by q_2. The set of rules R_2 is enforced at the separate entrance for visitors that already have fast pass ticket (where $time$ is the current system time:

$$R_2 = \begin{cases} q_2(fp(i,f), time) & \rightarrow permit \quad \textit{if } (i \leq time \leq f) \rightarrow true \\ q_2(fp(i,f), time) & \rightarrow deny \quad \textit{if } ((i < time) \vee (time > f)) \rightarrow true \end{cases}$$

Depending on the attraction, there can be more rules in the second policy, e.g. the ticket holder must have a given minimum height, or a minimum age, not to have heart conditions, etc. The strategy ζ_2 applies the rules in the order they are shown above.

The resulting policy is expressed as :

$$\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, \texttt{seq}(\zeta_1, \zeta_2))$$

This example shows the usefulness of the strategy operator `seq`. Sequential policy composition can capture several real world situations, where there is a need for achieving successive authorizations, from different entities, before being permitted to perform some action. Other examples of the use of the `seq` strategy for sequential combinations appear in our encoding for the XACML policy combiners in Section 6.2.4.

6.2.3 Conservative Policy Extension

A natural way of reusing a given policy is to add rules to it, such that the new policy can deal with a larger class of requests than it was able to treat before. A safe way to create such combined policy consists in building a *conservative extension* of the base policy. The idea is that the decisions produced for each request in the context of the original policy are not altered under composition. This property can be stated as follows:

Definition 51 (Conservative Extension)**.** *The composition $\wp = (\mathcal{F},\ D,\ R,\ Q,\ \zeta)$, of two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_i)$ $(i = 1, 2)$ is conservative if $\forall q_i \in Q_i$, then $[\zeta](q_i) = [\zeta_i](q_i)$.*

In other words, the result of the evaluation of a given access request under the combined policy strategy is the same as the result of its evaluation under the original component strategy. A suitable strategy for such combinations is `choice`, since it assures that the first set of rewrite rules will be applied to all input terms, and in the case of failure, a second set of rewrite rules is used. We can state the following theorem concerning this strategy:

Proposition 9. *The composition $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, Q_1 \cup Q_2, \texttt{choice}(\zeta_1, \zeta_2))$ of two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_I)$ $(i = 1, 2)$, with disjoint signatures is conservative provided that one of the component strategies is not the* `id` *strategy.*

Proof. From the definition of the `choice` strategy, the application of ζ_1 over a request term $q_2 \in Q_2$ always fails, because such term cannot be the source of any rewrite derivation contained in ζ_1, since the symbols in q_2 do not belong to $\mathcal{F}_1$. Then, q_2 evaluated by the second strategy, which will then return the same result as usual, then, $[\zeta](q_2) = [\zeta_2](q_2)$. The same is true if the order of the arguments in the `choice` strategy is exchanged. Therefore, the combination of two policies with disjoint signatures with the strategy $\texttt{choice}(\zeta_1, \zeta_2)$ is conservative. When the strategy of the first component is the `id` strategy, which always succeeds, the conservativity cannot be verified. □

Conservative combinations can ensure the validity of the principles of autonomy and security from [Cholvy and Cuppens, 1997]. Autonomy means that any access permitted in the individual system must also be permitted by the composed policy. The security principle means that access not permitted by a policy in isolation must also be denied under interoperation.

The combination of policies which share symbols may hardly be conservative extensions. Even with the use of a strategy such as `choice`, it is hard to assure that a first strategy will not reduce a request term that should be dealt with by the second strategy, and vice versa. Consider the following example:

Example 26. *The policy P_1 is defined as*

$$\begin{aligned} P_1 = (\quad \mathcal{F}_2 &= \{a : Term,\ b : Term\}, \\ D_1 &= \{b\}, \\ R_1 &= \{a \rightarrow b\}, \\ Q_1 &= \{a\}, \\ \zeta_1 &= \texttt{universal}(R)) \end{aligned}$$

and let the policy P_2 be

$$\begin{array}{rcl} P_2 = (\quad \mathcal{F}_2 & = & \{f : Term \rightarrow Term,\ a : Term\}, \\ D_2 & = & \{a\}, \\ R_2 & = & \{f(x) \rightarrow x\}, \\ Q_2 & = & f(\mathcal{T}(\mathcal{F})), \\ \zeta_2 & = & \texttt{universal}(R)) \end{array}$$

The combined policy be $(\mathcal{F}_1 \cup \mathcal{F}_2, D, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \texttt{choice}(\zeta_1, \zeta_2))$ is not conservative, since in the context of P_2, we have the following result $f(a) \xrightarrow{*}_{R_2} a$. Whereas for P_1, $a \xrightarrow{*}_{R_2} b$. However, for the compound policy, the request becomes a counter-example for conservativity:

$$[\texttt{choice}(\zeta_1, \zeta_2)](f(a)) \xrightarrow{*} b$$

Remark that both signatures share only the symbol a, and that R_1 and R_2 do not contain overlapping rules.

In the next section we introduce more "conventional" policy combiners, known as conflict resolution operators. Combiners in this category basically assume that the results of all sub-policies being combined need to be collected, and conflicting decisions will be disambiguated by the super-policy. This is the characteristic of the XACML policy combiners and of several other approaches.

6.2.4 XACML-like Policy Combiners.

In this section we show that the strategic rewriting approach can capture the behavior of conflict resolution policy combiners, such as those proposed in the industrial standard access-control language XACML [Moses, 2005]. The idea of disambiguating among possibly conflicting decisions appear in several previous initiatives, such as in [Jajodia et al., 2001, Kalam et al., 2003], which is exactly the goal of the XACML policy combiners, whose main ones are introduced below.

- *permit-overrides* : whenever one of the policies answers to a request with a *granting* decision, the final authorization for the composed policy is $permit$. The policy will generate a *denial* only in the case at least one of the sub-policies denies the request, and all others return *not-applicable* or $inderterminate$. When there is no sub-policy that returns neither $permit$ or $deny$, but explicitly *not-applicable*, the final outcome is *not-applicable*. The decision is $indeterminate$ if all sub-policies do not return any decision, *i.e.*, when unexpected errors occur in every evaluation attempt.

- *deny-overrides* : this combiner has a similar semantics to *permit-overrides*, with the difference that denials take precedence.

- *first-applicable* : the decision produced by the combined policy corresponds to the authorization determined by the first sub-policy that does not fail, and whose decision is different from *not-applicable*.

- *only-one-applicable* : the output decision is $indeterminate$ if more than one policy set returns a decision different from *not-applicable*. The resulting decision will be $permit$ or $deny$ if the single policy that applies to the request generates one of these decisions. The result will be *not-applicable* if all policies return such decision.

We give the semantics of these operators (and one more) below.

Permit-Overrides and Deny-Overrides

The *permit-overrides* algorithm specified in [Moses, 2005] is translated into the rewrite system R_{po} below:

$$\begin{array}{ll} po(permit, y) & \rightarrow permit \\ po(deny, na) & \rightarrow deny \\ po(deny, indeterminate) & \rightarrow deny \\ po(na, na) & \rightarrow na \\ po(na, indeterminate) & \rightarrow na \\ po(indeterminate, indeterminate) & \rightarrow indeterminate \end{array}$$

The decision $indeterminate$ denotes that an error occurred during the evaluation of a request, for instance, the policy did not respond. In order to apply these rules, it is necessary to first collect the results of the policies being combined. This can be performed through a strategy expression, which is defined as follows.

We provide an additional rule, R_{wrap}, to add a top symbol over incoming requests, that will be later submitted to each sub-policy for evaluation. The function symbol po is commutative and $q(x)$ illustrates request terms (the actual terms depend on the signature of the policies being combined).

$$R_{wrap} = \Big\{ q(x) \rightarrow po(q(x), q(x))$$

In order to encode the *permit-overrides* combiner over two sub-policies, we can define the following sequential combination:

$$[\zeta_{po}](q) = \texttt{seq}([R_{wrap}](q), \texttt{onceBottomUp}(\zeta_1), \texttt{onceBottomUp}(\zeta_2), R_{po})$$

It means that the global policy intercepts requests before they are evaluated by the sub-policies. This builds a new term (of the form $po(t, t)$), containing two subterms, which will be separately evaluated in a bottom-up fashion by the component sub-policy strategies ζ_1 and ζ_2, then the reductions with R_{po} occur in the top position of the wrapped term, generating a final decision.

Therefore, given two policies $\wp_i = (\mathcal{F}_i, \{permit, deny, na, indeterminate\}, R_i, Q_i, \zeta_i)(i = 1, 2)$ the *permit-overrides* combiner is defined as follows:

1. $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \ \cup$
   $\{po : Decision \times Decision \rightarrow Decision,\ q : Request \rightarrow Decision\}$;

2. $D = \{permit, deny, na, indeterminate\}$;
3. $R = R_1 \cup R_2 \cup R_{po} \cup R_{wrap}$;
4. $Q = \{q(t) \mid t \in Q_1 \cup Q_2\}$;
5. $\zeta = \zeta_{po}$.

Clearly, *deny-overrides* can be simulated in an analogous way. The XACML specification mentions ordered and unordered versions of these combiners, saying that the order in which the policies appear must be respected by the request evaluations, which in our encoding corresponds to the order of the arguments.

First-Applicable

In order to simulate the *first-applicable* combining algorithm, we need only to construct the strategy that schedules the rules in the order they appear in the policy specification. The fa function defined by the rewrite rules given below, which we call R_{fa}, encodes the *first-applicable* combiner in the same straightforward manner we encoded *permit-overrides*, given that the order of the arguments correspond to the order of the sub-policies in the combination defined by the user:

$$\begin{array}{ll} fa(permit, y) & \rightarrow permit \\ fa(deny, y) & \rightarrow deny \\ fa(na, na) & \rightarrow na \\ fa(na, indeterminate) & \rightarrow na \\ fa(indeterminate, indeterminate) & \rightarrow indeterminate \end{array}$$

The component policies can be easily scheduled with the application of a rewrite strategy:

$$[\zeta_{fa}](q) = \texttt{seq}([R'_{wrap}](q), \texttt{onceBottomUp}(\zeta_1), \texttt{onceBottomUp}(\zeta_2), R_{fa})$$

where R'_{wrap} is the following rewrite system :

$$q(x) \rightarrow fa(q(x), q(x))$$

Therefore, Given two policies $\wp_i = (\mathcal{F}_i, \{permit, deny, na, indeterminate\}, R_i, Q_i, \zeta_i)(i = 1, 2)$, the combined policy with the *first-applicable* operator is defined as follows:

1. $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \ \cup$
   $\{fa : Decision \times Decision \rightarrow Decision,\ q : Request \rightarrow Decision\}$;
2. $D = \{permit, deny, na, indeterminate\}$;
3. $R = R_1 \cup R_2 \cup R_{po} \cup R'_{wrap}$;
4. $Q = \{q(t) \mid t \in Q_1 \cup Q_2\}$;
5. $\zeta = \zeta_{fa}$.

Only-One-Applicable

In order to encode the *only-one-applicable* combining algorithm, we define the commutative function symbol oo through the rewrite system R_{oo} below:

$$\begin{array}{ll}
oo(permit, permit) & \rightarrow indeterminate \\
oo(permit, indeterminate) & \rightarrow permit \\
oo(permit, na) & \rightarrow permit \\
oo(deny, permit) & \rightarrow indeterminate \\
oo(deny, deny) & \rightarrow indeterminate \\
oo(deny, indeterminate) & \rightarrow deny \\
oo(deny, na) & \rightarrow deny \\
oo(na, indeterminate) & \rightarrow na \\
oo(na, na) & \rightarrow na \\
oo(indeterminate, indeterminate) & \rightarrow indeterminate
\end{array}$$

The component policies can be easily schedule with the application of a rewrite strategy:

$$[\zeta_{oo}](q) = \texttt{seq}([R'_{wrap}](q), \texttt{onceBottomUp}(\zeta_1), \texttt{onceBottomUp}(\zeta_2), R_{oo})$$

where R''_{wrap} is the following rewrite system :

$$q(x) \rightarrow oo(q(x), q(x))$$

Therefore, given two policies $\wp_i = (\mathcal{F}_i, \{permit, deny, na, indeterminate\}, R_i, Q_i, \zeta_i)(i = 1, 2)$, the combined policy with the *only-one-applicable* operator is defined as e follows:

1. $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \ \cup$
   $\{oo : Decision \times Decision \rightarrow Decision,\ q : Request \rightarrow Decision\}$;
2. $D = \{permit, deny, na, indeterminate\}$;
3. $R = R_1 \cup R_2 \cup R_{oo} \cup R''_{wrap}$;
4. $Q = \{q(t) \mid t \in Q_1 \cup Q_2\}$;
5. $\zeta = \zeta_{oo}$.

Majority

The use of rewrite rules and strategies allows to define other operators of the same nature as the XACML combiners, which do not appear in the standard specification [Moses, 2005]. Indeed, the standard determines that the set of combiners it proposes is not definitive, but can be extended by its users if necessary. For instance, it is interesting to have a policy combiner that takes the majority of the decisions produced by the sub-policies as the outcome of a policy combination.

Such combiner is defined by the rewrite system R_{maj} below:

$$
\begin{array}{ll}
maj(permit, permit, z) & \rightarrow\ permit \\
maj(deny, deny, z) & \rightarrow\ deny \\
maj(na, na, z) & \rightarrow\ z \\
maj(indeterminate, indeterminate, z) & \rightarrow\ z \\
maj(permit, deny, na) & \rightarrow\ permit \\
maj(permit, deny, indeterminate) & \rightarrow\ permit
\end{array}
$$

We assume that the maj function symbol is permutative. Remark that we are interpreting na and $indeterminate$ as abstentions. This situations may provoke a "tie", which is solved by the last two rules in R_{maj} – the policy author may assign a different default value in these cases. Here we have illustrated them by assigning the value $permit$. Although the idea of this "voting" policy is very intuitive, up to our knowledge, it has not been proposed in previous works.

We define this policy combination in the same way as we did for the previous operators. Given three policies $\wp_i = (\mathcal{F}_i, \{permit, deny, na, indeterminate\}, R_i, Q_i, \zeta_i)(i = 1, 2, 3)$, their combination is:

1. $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \ \cup \mathcal{F}_3$
   $\cup\{maj : Decision \times Decision \times Decision \rightarrow Decision,\ q : Request \rightarrow Decision\}$;
2. $D = \{permit, deny, na, indeterminate\}$;
3. $R = R_1 \cup R_2 \cup R_3 \cup R_{oo} \cup R'''_{wrap}$;
4. $Q = \{q(t) \mid t \in Q_1 \cup Q_2 \cup Q_3\}$;
5. $\zeta = \zeta_{maj}$.

where R'''_{wrap} is

$$q(x) \rightarrow maj(q(x), q(x), q(x))$$

and ζ_{maj} is

$$\texttt{seq}([R'''_{wrap}](q), \texttt{onceBottomUp}(\zeta_1), \texttt{onceBottomUp}(\zeta_2), \texttt{onceBottomUp}(\zeta_3), R_{maj})$$

The combiner above was over three components, but other arrangements can be defined for a different number of arguments. For example, if we are combining only two sub policies, we may demand that both agree on the same result, otherwise, the final decision would be na (below we only sketch the rewrite rules instead of the full policy definition):

$$
\begin{array}{ll}
maj'(permit, permit) & \rightarrow\ permit \\
maj'(deny, deny) & \rightarrow\ deny \\
maj'(permit, deny) & \rightarrow\ na \\
maj'(x, indeterminate) & \rightarrow\ x
\end{array}
$$

For other number of arguments the encoding will be similar to the previous version, maj.

As we have seen, there are several ways of combining policies by using rewrite rules and strategies. We preferred not to propose a set of composition operators, but rather to present how they can be defined in our framework. In the next section we discuss related works in policy composition.

6.3 Related Works

With respect to policy composition, a number of works have a close relationship with the formalization introduced in this thesis. Policy composition is addressed in [Bonatti et al., 2002] through an algebra of composition operators that is able to define union, intersection, policy templates, among other operations. The operator definitions can be adapted to several languages and situations, since their definition is orthogonal to the underlying authorization language. The definition of the policy closure operator in [Bonatti et al., 2002] opens the possibility to define policy combiners in the same way we proposed the majority combiner: for instance, given a policy (a set of ground authorizations) and set of inference rules, the generated policy will be closed with respect to these rules.

The work presented in [Wijesekera and Jajodia, 2003] extended Bonatti et al.'s algebra with negative authorizations and non-determinism, features that our rewrite-based framework supports naturally (even more, since we do not restrict the set of decisions). [Wijesekera and Jajodia, 2003] also includes an operator for sequential composition, but no examples of its use are given. They do not cover the template operator from Bonatti et al.'s algebra.

The policy composition algebra of [Bonatti et al., 2002] is a representative work in the computer security literature, that is why we give below how their policy operators can be encoded in our framework. The reader should refer to the original article for example of use for each operator. Here we restrict ourselves to the explanation of their semantics.

Example 27. *Policies are interpreted as sets of ground authorizations of the form $permit(S, O, A)$, which means that a subject from the set S of subjects is permitted to execute an action from the set A of actions over a resource from the set O of objects. Therefore only positive authorizations can be expressed in [Bonatti et al., 2002]. In [Wijesekera and Jajodia, 2003] prohibitions and permissions are supported through the addition of the symbol $\pm$ before the symbols representing actions. Below we encode sets as lists of terms. The signature contains some operators for list concatenation, and for checking whether an element is in the list. The rules are given in Table 6.1.*

The composition operators from [Bonatti et al., 2002] expressed in our framework is the policy $P_{alg} = (\mathcal{F}, Q, D, R, \zeta)$ defined as follows

- *The signature $\mathcal{F}$ contains the binary operators $+, -, \&, *, \hat{}$ whose domain and co-domain are lists of authorizations, the ternary symbol o (for override), the constant nil for the empty lists, $:$ for adding an element to the list, and $++$ for list concatenation.*
- *Decisions are sets of authorizations,*
- *Requests are expressions containing symbols $+, -, \&, , \hat{}, o$. They are reduced until only lists of permission terms are obtained,*
- *The set of rewrite rules R is defined in Table 6.1. The addition operator $+$ merges two policies, by taking the union of the sets of authorizations. The policy conjunction operator $\&$ takes the intersection of the sets of permissions. The subtraction $-$, stands for the difference of sets of decisions. The overriding operator substitutes part of a policy, specified by a set of authorizations, with a third set of authorizations. The closure under rules operator*

$*$ is defined with the help of an external strategy, which reduces the current term using the given rules. This is more general than the original framework, but the way to implement the more restricted version is similar. The scoping operator selects elements from a set of permissions which satisfy a given set of constraints (in our approach given by a set of predicates over permissions, but not specified here). Explicit rules for policy templates are not given. We assume that input expressions with variables can be seen as templates. In Table 6.1 the variables have the following sorts: $x, y : Permission, L, L_1, L_2 : List$.

- *The strategy ζ is any strategy for evaluating expressions containing the operators from the set Q which respects the precedence among the operators in Q as defined in [Bonatti et al., 2002].*

The goal of this example was to give an executable specification of the algebraic operators for access control composition proposed in [Bonatti et al., 2002]. A subset the operators of [Wijesekera and Jajodia, 2003], which can also be encoded in strategic rewriting in a similar way. The code for this example appears in Appendix A.3.

The algebras for policy composition are very well suited for composing policies that can be easily abstracted as triples of the kind $permit(S, O, A)$, from more classical perspectives for access control like RBAC. Our framework would not be more attractive than the approaches of [Bonatti et al., 2002] or [Wijesekera and Jajodia, 2003], in these cases. On the other hand, the use of rewrite rules can suitably tackle the definition of combiners for policies with multiple decisions, in an approach more in line with the current needs of policy languages, such as the XACML industrial standard. We can handle inconsistencies in several ways, either by checking whether the combination will preserve the consistency of the component policies, or by defining different kinds of conflict resolution strategies.

Another alternative for composing access control policies is implemented by the Polymer system [Bauer et al., 2005]. It supports the reuse of policies which are first-class objects in the language, by enforcing a conjunction of policies, then more restrictions are enforced over a given program; or also by imposing some evaluation order of the sub-policies with precedence operators; or yet by modifying policies with new actions to be executed when some security event is triggered, in a sort of higher-order composition. The formal semantics of Polymer is based on a variant of the lambda calculus as explained in [Bauer et al., 2002].

A different approach to composition is taken in [Lee et al., 2006]. Based on the non-monotonic properties of defeasible logic, the authors propose an automated approach for composing policy specifications for web-service security which relies on a single operator. It takes into account a precedence relation among the policies, therefore, the decision of the policy with the highest priority is taken, which is suitable in the case of the simple policies supported nowadays by web services. This is of course not the case for rich access control, as considered in this chapter.

In [Bruns et al., 2007] policies are defined as four-valued predicates, with the following possible values: $permit, deny, not\ applicable$ and $conflict$, this last value is assigned when a conflict is detected among the policies being combined. These values are organized in a bi-lattice such that conflict resolution is separated from policy composition. The approach proposes a set of high-level composition operators coherent with a four-valued logic for policies.

Semantics	Rewrite Rules
Base rules	$R = \left\{ \begin{array}{l} nil +\!\!+ L \rightarrow L \\ (x : L) +\!\!+ L2 \rightarrow x : (L +\!\!+ L2) \\ x : (x : L) \rightarrow x : L \\ \\ x \in nil \rightarrow false \\ x \in (x : L) \rightarrow true \\ x \in (y : L) \rightarrow x \in L \text{ if } x \neq y \rightarrow true \end{array} \right.$
Union of two policies	$R_+ = \left\{ \begin{array}{l} nil + L \rightarrow L \\ (x : L_1) + (y : L_2) \rightarrow x : (y : (L_1 + L_2)) \end{array} \right.$
Intersection of two policies	$R_{\&} = \left\{ \begin{array}{lll} nil \,\&\, L & \rightarrow nil & \\ x : nil \,\&\, y : nil & \rightarrow nil & \text{if } x \neq y \rightarrow true \\ (x : L_1) \,\&\, (x : L_2) & \rightarrow x : (L_1 + L_2) & \\ (x : L_1) \,\&\, L_2 & \rightarrow L_1 \,\&\, L_2 & \text{if } x \in L_2 \rightarrow false \\ (x : L_1) \,\&\, L_2 & \rightarrow x : (L_1 \,\&\, L_2) & \text{if } x \in L_2 \rightarrow true \end{array} \right.$
Difference of two policies	$R_- = \left\{ \begin{array}{lll} nil - L & \rightarrow nil & \\ (x : nil) - (y : nil) & \rightarrow (x : nil) & \text{if } (x \neq y) \rightarrow true \\ (x : L_1) - L_2 & \rightarrow x : (L_1 - L_2) & \text{if } (x \in L_2) \rightarrow false \\ (x : L_1) - L_2 & \rightarrow L_1 - L_2 & \text{if } (x \in L_2) \rightarrow true \end{array} \right.$
Overrides part of a policy with another one	$R_o = \left\{ \; o(L_1, L_2, L_3) \rightarrow (L_1 - L_2) \,\&\, (L_2 + L_3) \right.$
Closure under a strategy ζ	$R_* = \left\{ \begin{array}{lll} nil & * \zeta & \rightarrow nil \\ (x : L) * \zeta & & \rightarrow [\zeta](x) : (L * \zeta) \end{array} \right.$
Scope under constraints	$R_{\hat{}} = \left\{ \begin{array}{ll} nil \,\hat{}\, \{c_1, \ldots, c_n\} & \rightarrow nil \\ (x : L) \,\hat{}\, \{c_1, \ldots, c_n\} & \rightarrow x : (L \,\hat{}\, \{c_1, \ldots, c_n\}) \\ \text{if } (c_1(x) \wedge \ldots \wedge c_n(x)) \rightarrow true & \end{array} \right.$

Table 6.1: Rules for the composition algebra of [Bonatti et al., 2002]

6.4 Conclusions and Future Works

This chapter has shown that strategic rewriting provides a formal background as well as the necessary expressivity for a policy composition framework. The operational semantics of the strategy language allows us to combine sets of decisions, to give priorities to rules (and to sets of rules), and also to interfere on the flow of information between policies, through a sequencing of requests among policies. We have encoded several existing approaches using rewrite rules thanks to the expressivity provided by our strategy language.

We have shown how to deliver fine-grained control over the rule interaction of sub-policies. We have also shown how to encode conflict resolution combiners, like those in XACML, which assume that policies can have a conflicting nature, an assumption which has been broadly adopted in several recent works in policy composition [Moses, 2005, Bruns et al., 2007, Tschantz and Krishnamurthi, 2006]. Moreover, our definition for policies is very general, and we may capture some policies for which it would be very hard to find a correspondence in their abstract model, since in our framework we can state permissions over any set of concepts, other than subjects, objects and actions. Another advantage of our framework is that we can perform formal reasoning for policy composition in the presence of a multiplicity of access decisions, including when policies assign explicitly an "undefined" decision for some request, which cannot be captured in several of the previous frameworks. We have also shown the interest of strategic rewriting in the preservation of some policy properties like consistency, termination, and on conservative policy extensions.

Future Perspectives

There are several research directions concerning the topics presented in this chapter of the thesis which need to be further explored. Part of these research topics have to do with the rewrite theory under strategies, and further issues are related to the properties of policy combinations. We summarize them in the following:

- We have seen in this chapter that there are numerous works on the modularity of the termination and confluence properties of rewrite systems, but little is know with respect to the modularity of the sufficient completeness of rewrite systems being merged (either for disjoint systems or not). Therefore, it is important to develop results about the sufficient completeness property for the union of rewrite systems.
- A second important research question that emerges from this work, with respect to the rewrite theory, is that we have several modularity results on the union of rewrite systems with no specific strategy, or under some classic term traversal strategy, such as innermost or outermost. But we do not dispose of results for general rewrite strategy expressions, built from terms using a strategy language such as the one we presented in Chapter 3.
- A further research question is which are the strategy operators are able to provide some interesting composition properties such as policy containment described in [Dougherty et al., 2006]? A policy $\wp_1$ is contained by a policy $\wp_2$, written $\wp_1 \preceq \wp_2$ if all decisions produced by $\wp_1$ for its set of request terms are the same as the set of decisions returned by $\wp_2$ over the same inputs. Uniform containment means that if there exist a third policy, $\wp$ and a policy combiner $\oplus$ such that if $\wp_1 \preceq \wp_2$, then $\wp_1 \oplus \wp \preceq \wp_2 \oplus \wp$.

Chapter 7

Policy Enforcement

In this chapter, we address the problem of enforcing expressive access control policies defined in our rewrite-based framework of Chapter 4. We propose a method to enforce these policies over existing programs, by combining the Tom system [Balland et al., 2007], in which our our policies can be appropriately written, and aspect oriented programming [Kiczales et al., 1997], which provides the necessary facilities to perform the program transformations from untrusted code into policy adherent programs. The work described here appeared in [de Oliveira et al., 2007].

Policies can unambiguously discern which authorized operations principals can perform on the resources of a system fairly well when the security conditions do not change over time, for example, in models such as the access control matrix and RBAC. But it becomes much more difficult when policies depend on their running environment for providing an access control decision. Some models for access control put forward the need for flexible access control policies [Kalam et al., 2003, Jajodia et al., 2001, di Vimercati et al., 2005].

Since access control is one of the pillars of secure applications, an important problem that needs to be addressed in such context is the enforcement of flexible rule-based policies. Currently, the most widespread form of access control mechanism is the reference monitor, that watches the execution of a target program, and interferes with it when it is about to violate the security policy. Such monitors need to be easily analyzed and tested, and they should not be bypassed.

We design a generic scheme to *weave* rewrite-based security policies into target programs. The method constructs a reference monitor for a given policy by *inlining* the policy rules inside the program. In addition, we design a methodology to relate the policy description to the program code, which makes explicit the correspondence between the policy environment and the data manipulated by a target program. At the same time, this allows for the separation between policies and programs, which provides reusable policy modules.

Our solution is implemented through the connection of two specific tools, Tom (presetend in Section 3.4) and AspectJ [Kiczales et al., 2001]. The latter tool is a current implementation of Aspect-Oriented Programming (AOP) [Kiczales et al., 1997] for Java. Their combination allows us to inline access control rules as a program monitor, and thus to provide an efficient implementation of a given policy.

The structure of the chapter is as follows. In Section 7.1, we present the running example of this chapter. Section 7.3 shows how aspects can be used to enforce rewrite-based policies on existing code, and discusses their impact on the security of the generated program. Section 7.4 surveys related works, before the conclusion in Section 7.5.

7.1 The Running Example

Flexible policies consider the values of attributes of principals and resources, in addition to other conditions, such as time, location, etc, as part of the policy environment. The access control decisions generated by a policy may change according to the current state of a given application.

To illustrate our approach for dynamic policy enforcement, we take as running example the policy for a conference management system described in [Dougherty et al., 2006]. The rules we give are as follows:

1. During the submission phase, an author may submit a paper;
2. During the review phase, a reviewer r may submit a review for paper p if r is assigned to review p;
3. During the meeting phase, a reviewer r can read the scores for paper p if r has submitted a review for p;
4. A reviewer cannot submit reviews or read scores for a paper he/she is conflicted with;
5. Authors may never read scores.

Example 28. *The Conference System access control policy is defined by:*

- *The policy signature $\mathcal{F}$ is given below:*

$$
\begin{array}{lll}
paper & : Id \times Title & \rightarrow Object \\
author, reviewer & : Id & \rightarrow Subject \\
submitPaper, readScores & : & \rightarrow Action \\
submission, review, meeting & : & \rightarrow Phase \\
conflict & : Subject \times Object & \rightarrow Condition \\
assign & : Subject \times Object & \rightarrow Condition \\
q & : Subject \times Action \times Object & \rightarrow Request \\
aut & : Request \times Phase \times Condition & \rightarrow Decision \\
permit, deny, na & : & \rightarrow Decision
\end{array}
$$

- *The requests are any terms whose symbol in the top position is aut;*
- *The set of decisions contains the constants $permit$, $deny$, and na.*
- *The rewrite rules are given in Figure 7.1, where the variables appearing are of the following sorts: $x, y : Subject$, $z : Title$, $phase : Phase$, $cnd : Condition$, $action : Action$, $p : Object$, and $req : Request$;*
- *The strategy applies the rules r_1 to r_8 from Figure 7.1 in an innermost strategy, and in the case they fail, it applies rule r_9.*

$$
\begin{array}{lcl}
r_1 : aut(q(author(x), submitPaper, paper(x,z)), submission, cnd) & \rightarrow & permit \\
r_2 : aut(q(author(x), submitPaper, paper(x,z)), phase, cnd) & \rightarrow & deny \\
r_3 : aut(q(author(x), readScores, paper(x,y)), phase, cnd) & \rightarrow & deny \\
r_4 : aut(q(reviewer(x), action, p), phase, conflict(x,p)) & \rightarrow & deny \\
r_5 : aut(q(reviewer(x), submitReview, paper(y,z)), review, assign(x, paper(y,z))) & \rightarrow & permit \\
r_6 : aut(q(reviewer(x), submitReview, paper(y,z)), phase, assign(x, paper(y,z))) & \rightarrow & deny \\
r_7 : aut(q(reviewer(x), readScores, paper(y,z)), meeting, assign(x, paper(y,z))) & \rightarrow & permit \\
r_8 : aut(q(reviewer(x), action, paper(x,z)), phase, cnd) & \rightarrow & \\
\quad aut(q(reviewer(x), action, paper(x,z)), phase, conflict(x, paper(x,z))) & & \\
r_9 : aut(req, phase, cnd) & \rightarrow & na
\end{array}
$$

Figure 7.1: Access rules for the conference system.

It is worth noticing that the rule r_9 in Figure 7.1 assures that every request not matched by the left hand side of the previous rules will be associated to the symbol na (which stands for "not applicable"). Additionally, the termination of this rewrite system is not trivial. It requires an advanced technique taking the rule ordering into account, in particular, this example was used to illustrate the approach introduced in [Gnaedig, 2008].

We argue that refinements of the security policy have to precede any deployment. Users must verify that a given policy satisfies the suitable properties (completeness, consistency and termination) before proceeding to its enforcement in a system. In the next section, we present how rewrite-based policies are described in Tom.

7.2 Policies in Tom

The program excerpt in Figure 7.2 illustrates some of the Tom constructs on a simple example. We encode part of the conference system policy for illustration purposes (the actual full version of the policy is found in Appendix A.4). The example is divided in two parts. The first part determines the signature for the terms. It is described in the code enclosed between the keywords `%gom` and curly braces (from lines 2 to 14). Since variables are not declared in Tom, we use empty parentheses after constant symbols to distinguish them from variables. The reader may see the symbols of a given sort, for example, the constant `review()` is of sort `Phase`. The function `aut` returns a value of sort `Decision`. In order to build a request, one can use the following exression q_1 = `` `q(author(1),submitPaper(),paper(1,"title")) ``, which contains only ground terms, and corresponds to the demand of an author whose id is `1`, to perform the action `submitPaper()`.

Rewrite rules are defined within the `%strategy` block. For the policy in Figure 7.2, we gave the rule set a name, `Rules`, which are applied, according to in line 24, under an innermost strategy: subterms in more internal positions are reduced first. A call for rule application is illustrated in line 26, over the input term `q1`, whose result is of type `Decision`.

Figure 7.2 illustrates how Tom programs are embedded in Java. Following the paradigm of *formal islands* [Balland et al., 2006b, Balland et al., 2006c], such programs consist of a list of Tom constructs interleaved with Java code. During the compilation process, all Tom constructs

```
public class Policy {
%gom{
    ...
    Request = q(s:Subject, a:Action, o:Obj)
    Phase = submission()
          | meeting()
          | review()
    Action = submitPaper()| readScores()...
    Decision = permit()
      | deny()
      | aut(r: Request, p:Phase)
    ...
}
 ...
%strategy Rules() {
 ...
 aut(q(author(x),submitPaper(),paper(x,y)),
  submission(),cnd) -> {return `permit();}
 ...
 }
 ...
 public static boolean apply(...){
    ...
    Strategy policy =`Innermost(Rules());
    ...
    Decision d = (Decision)policy.visit(q1);
    ...
 }
 ...
}
```

Figure 7.2: Structure of a policy in Tom

are translated into Java instructions, in a process that preserves the semantics of the Tom code, as shown in [Kirchner et al., 2005].

7.3 Enforcing Rewrite-Based Access Control

We are now ready to address the core problems of rewrite-based policy enforcement through construction of reference monitors.

A reference monitor is a mechanism for enforcing a security policy. It watches over the execution of an untrusted program and is able to interrupt its execution in the case of a policy violation. Examples of program monitors include firewalls, operating system kernels, and other components that intercept calls made by target applications. These monitors can be seen as an independent unit in the system architecture, but very often they are *inlined* in the application's code, at compile or load time. In this case, the untrusted code is transformed in such a way that the program's actions are forced to go through the monitor, which decides whether to abort its execution or not, thus avoiding the system from entering an insecure state.

The languages adopted in previous works to describe the policy enforced by a given reference monitor have a rather low abstraction level, which is comparable to the language used to describe the target program itself [Bauer et al., 2005, Erlingsson and Schneider, 2000]. For example, it is possible to express policies that disallow system calls for opening files in the host computer, or to block access to network services after reading data, but it is hard to encode the dynamic behavior of a high-level policy, which takes an authorization decision based on the current conditions in its running environment.

Thus, the model of the policy environment becomes a crucial element for enforcing dynamic policies. In this work, it consists of a collection of values of interest to the policy (determined by the policy signature). Therefore the application's current state is structured as a term, which is modified along its execution. Whenever the application executes an action concerned by the policy, this term is evaluated in order to compute an access decision by applying the rewrite rules that define the policy.

The policy itself does not describe how the system state evolves, but whether the state transitions are allowed or not. This approach is called *execution monitoring*, and is illustrated in Figure 7.3, which is divided in two parts: the first presents the behavior of a target program alone, the second shows how the transitions are intercepted by a reference monitor enforcing a given policy, which can prevent a transition by aborting the execution of the target.

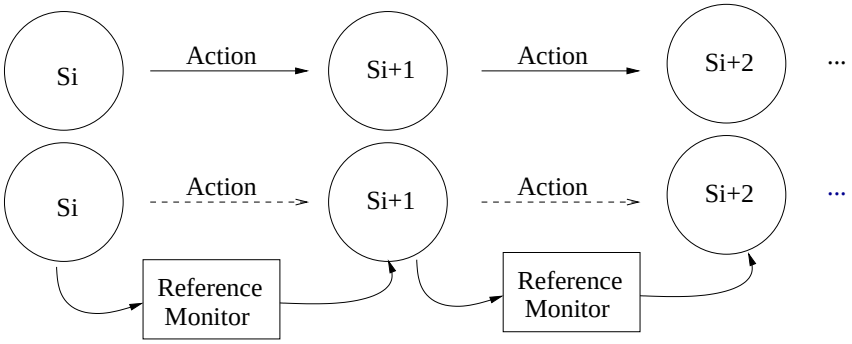


Figure 7.3: Reference monitoring: transitions correspond to actions

The approach taken in some previous works (for instance [Erlingsson and Schneider, 2000]) supposes that target programs are potentially malicious, because they are provided by untrusted third parties. Another assumption is that the reference monitor cannot have access to the source code for that application, but only to executable binaries. In this work, we provide a methodology for the development of the policy and of the target program, giving the choice of building suitable mechanisms to enforce the policy. Thus, our main goal is to help developers to avoid bugs related to access control in their applications.

In the rest of this section, we present in detail our approach for enforcing dynamic policies based on term rewriting, through the use of aspect-oriented programming.

7.3.1 Aspect-Oriented Programming

The concept of Aspect Oriented Programming (AOP) was introduced in [Kiczales et al., 1997], as an approach to separate crosscutting concerns, or capabilities which are orthogonal to the system's main functionalities, in single units, called *aspects*. Aspects encapsulate behaviors that affect multiple classes (or units, modules, etc.) in reusable code fragments. Since aspects centralize the code for crosscutting issues, which would be rather spread along the code, in the hierarchical structure of the program, AOP drastically improves the potential reuse of these program artifacts, and decreases the maintenance costs.

In order to give a (classical) example of crosscutting concern, let us consider again a conference management system. In addition to its basic functionalities, such as submission of papers, assignment of papers to program committee members, and submission of reviews, there are other requirements such as logging, to register in a file all past activities performed by users in the system: the code for implementing the system log is scattered in several parts of the system.

The basic building blocks of an aspect-oriented programs are *pointcuts* and *code advices*. Pointcuts define where a given crosscutting concern should be called. Pointcuts may be associated to a set of function names, for example, and can be expressed through patterns. We call *join points* the locations in the application code that match a given pointcut. Code advice is the actual code executed in join points. In a conference system, for example, a pointcut is any call to the method for submitting papers. The advice code must print to the system log the identity of the current user and some information about the submitted paper, with date and time.

The last essential step in AOP is to *weave* aspects into the main application code. Aspect compilation is a program transformation process that matches the pointcuts defined inside the aspects and inserts the corresponding code advice before, after or around a joinpoint. The expressivity of an aspect-oriented language relies on the pointcut specification, for example, on the kinds of patterns allowed by the aspect language, or on the way code advice can be weaved. To experiment with these concepts, in this chapter, we have adopted AspectJ [Kiczales et al., 2001] which is becoming a very popular AOP implementation for Java.

7.3.2 System Architecture

The general view of our approach is given in Figure 7.4. It highlights the correspondence between the formal description of the policy, through its signature, and the program code. The policy declares a certain number of actions (constants of sort $Action$ in the policy signature) which are related to method calls in the source program. These actions determine sensible methods where requests for access must be made to a reference monitor. The reference monitor is the inlined representation of the policy rules, contained in a piece of code advice that is weaved into the target application code, generating a program that respects the policy.

In addition to requests, it is necessary to map the other constructors declared in the policy signature. This mapping tells how to interpret the current state of an application as a term. We use constructs provided by the aspect-oriented language to identify where the program objects of interest to the policy are created and/or modified (in AspectJ, these are pointcut designators of kind *field set*) in order to capture their values and build their equivalent term representations.

Therefore, the reliability of the policy enforcement depends on the ability of the policy devel-

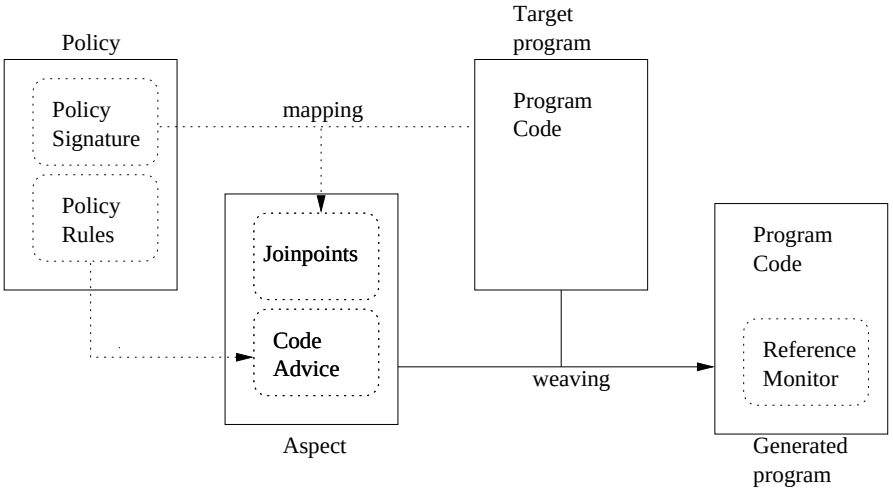


Figure 7.4: System architecture

oper to build a mapping that gives a correct representation from the program values to the policy terms representing the state of the program.

7.3.3 Aspects for Access Control

Capturing the policy environment Guided by the construction of the mapping between the policy signature and the objects manipulated by the target application, the application developer must define a set of pointcut designators over these objects to capture changes in their values. The most practical way of controlling this is to declare some auxiliary variables within the aspect to keep the latest values of such objects. For example, let us consider the current user logged in the system. In our running example, we declared a global variable that holds this information from the moment the user is authenticated via his id and password. This is illustrated in the code fragment listed below, which is a piece of the access control aspect for the conference system.

```
aspect PolicyAspect {
  ...
  private int phase;
  ...
  after(int cp): set(int Conference.currentPhase) && args(cp){
    phase=cp;
  }
  ...
}
```

This piece of code advice is executed each time there is an assignment to the public variable `Conference.currentPhase`. The new value is then stored in the aspect variable `phase`, which will later be used to actually build the term representing the policy environment. This process has to be repeated for every piece of data that is part of the policy environment.

This simple example raises some important subtleties. The first is the difficulty in automating the task of capturing the values of variables in dynamic contexts. The task is facilitated by the capabilities provided by AspectJ, but they are not sufficient. For example, it would not be possible to access the value of `currentPhase` from the conference class if it was declared as private. A second difficulty is polymorphism. Sensible calls may present several signatures in Java. Therefore any program transformation approach would need very powerful static analysis to be able to detect automatically all the forms these sensible calls assume. Other security holes are associated to reflective capabilities recently introduced in Java. Provided that the code can transform itself at run-time, the enforcement process has to be able to identify whether it is harmful, according to the policy.

Identifying access requests In classical access control models, the notion of operation, or action carried out by the active entities of the system has always been placed as the origin of access requests. In practice, every function call (or method call in the object-oriented paradigm) would imply an access request. The role of the policy is to indicate which of these actions are relevant for access control. In our definition, an access request takes the form `q(s:Subject,a:Action,o:Obj)`. Therefore, for each constant symbol of sort $Action$ in the policy signature, we must provide a set of pointcuts which associate the method calls to access requests that concern the policy. For instance, let us consider the program method that allows submitting a review for a paper in the program code. Its interface is shown in the code fragment below:

```
public void submitReview( int paperId, int reviewerId, int score)
```

It can be captured by the following pointcut designator in AspectJ :

```
pointcut submitReviewCut(int objId, int revId):
call(void Conference.submitReview(int,int,int)) && args(objId,revId,*);
```

where we have declared some arguments to keep the data from the paper and reviewer identifiers, information which is used by the code advice. The full code for the access control aspect for the conference system is available in Appendix A.5.

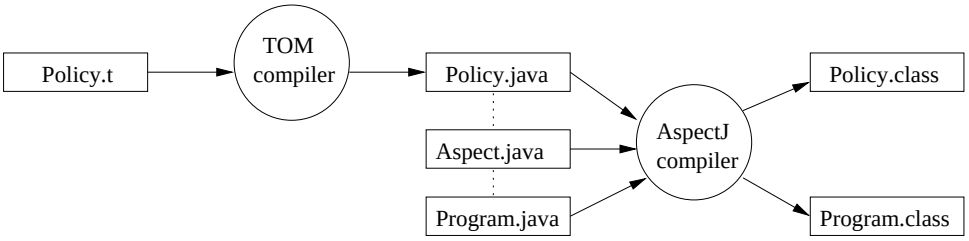


Figure 7.5: Weaving of access control rules using AspectJ and Tom

Weaving policy rules In Figure 7.5 we illustrate the interactions of the components and tools used in our weaving process. Since the Tom compiler generates Java code, there is no restriction to do the program transformations necessary for access control enforcement through the

facilities available in AspectJ. The conversion from Java objects into Tom terms is performed inside the policy class, implementing the complete mapping between the two representations described in Section 7.3.2.

In the code advice, the enforcement mechanism must apply the policy rules over the current request term. In order to evaluate the policy environment using a given rule set, we make a call to specific methods in the Java class generated by the Tom compilation process. This design decision avoids the application developers to be aware of the formal representation of the access control policy, which can be taken care of by the security engineer. Finally, the code advice for the `submitReview` action is presented below:

```
before(Paper p):submitPaperCut(p){
 paperId=p.getId();
 if(!Policy.apply(usr.getId(),paperId,usr.getRole(),phase,"submitPaper"))
 {
   System.err.println("Access␣Denied.");
   throw new AccessControlException();
 }
}
```

This advice code states that before any joinpoint determined by the submit paper pointcut designator, a call to the policy is executed whose arguments are the variables declared in the aspect, containing information from the current application context. When this call returns `false`, the reference monitor aborts the execution of the target application. The following code fragment shows how the values collected in the aspect are translated in the policy into their term representation (notice the use of Tom's backquote construct).

```
public static boolean apply(int sId,
  int oId, String role, int phase,
  boolean assign,  String action)
  throws Exception{
  ...
  switch(phase){
   case 0: {ph = `submission();break;}
   case 1: {ph = `review();break;}
   case 2: {ph = `meeting();break;}
  }
  ...
  Decision d = (Decision)
   policy.visit(`aut(q(s,a,o),ph,asg));
  if(d == `permit())
    return true;
  else
    return false;
    ...
}
```

Performance and Optimization

The main origins of the overhead caused by our enforcement scheme for access control policies are the code generated by Tom for the policy itself, and the overhead introduced by the weaving process made by the AspectJ compiler. Since our rewrite-based security policy language (*ie* full term rewriting) is Turing complete, the policy designer can "easily" add an arbitrary complexity to the existing code. However, in practice, we expect policies to be quite simple rewrite programs with a low computational complexity. Furthermore, the code generated by Tom is optimized [Balland and Moreau, 2006]. AspectJ also disposes of some optimization tools [Avgustinov et al., 2005]. We consider that the practical overhead is small, and that it is worth paying the price for better security mechanisms.

7.4 Related Works

Three categories of approaches to enforce security policies are classically identified [Hamlen, 2006]:

1. *Static analyzers* predict a program's potential harms prior to its execution, discarding those who have unacceptable behavior. An example, is the static type-checking performed by the Java virtual machine that rejects ill-formed bytecode.

2. *Execution monitors* (EM) continuously monitor a program's behavior as it runs, intervening when a coming policy violation is detected.

3. *Program-rewriters* transform untrusted code into self-monitoring code that performs security checks as it executes, thus the target programs are transformed into policy-satisfying ones.

The separations among these three classes are not sharp. Our approach best fits the last category, since we use the AOP technology and Tom as program and policy transformation tools. Actually, program-rewriters are able to combine the power of static analyzes and EM, by accepting untrusted code and well-formed policies as input, and automatically transforming them into policy-adherent and self-monitor code.

Using AOP for policy enforcement is not an original idea in itself. In [Song et al., 2005] the authors use a design by contract approach, to separate access control from the other application features at the design level. They explain how to build models based on AOP to design mechanisms to enforce RBAC policies. Another application of AOP is enforcement of availability properties as aspects [Cuppens et al., 2006]. Availability requirements are specified in a formal model, combining deontic and temporal logics, and then transformed into aspects. In both approaches, policies are written inside the aspect component, thus, in the case policies change, the aspects have to be changed accordingly. In [Dantas, 2007] extends functional programming languages with aspect-oriented features, primarily to explore aspect-oriented enforcement of security policies. It is shown that security advice can be implemented as harmless advice, which is code that cannot modify the behavior of the target application. This is achieved via a type and effect system related to information-flow type systems.

In the second category, we may cite [Schneider, 2000], which introduces a formal definition based on finite-state automata to express safety properties (nothing bad ever happens), proved enforceable by execution monitoring. This work was later extended [Ligatti et al., 2005] to consider other kinds of security properties.

In the class of program rewriters, we can mention some important works where security code is inserted into the executable target [Erlingsson and Schneider, 2000, Evans and Twyman, 1999, Hamlen, 2006, Bauer et al., 2005]. They can assure different levels of trust on the transformation they perform, or on the policy specification.

The main difference between all these works and ours is the power of policies description and representation. In previous works, they are limited to an "event language" that the program monitor can intercept between state transitions, it is powerful but rather low level. In our work, we are able to describe more intelligible policies and relate them to the program code, although this process needs human intervention.

7.5 Conclusions

The need for flexible policies is predominant in today's information systems. The understanding on how to manage complex constraints on the policy environment has augmented significantly with the introduction of several rule-based formalisms for access control [Dougherty et al., 2006, Antoniou et al., 2007, Jajodia et al., 2001, di Vimercati et al., 2005]. This is due to the fact that rule-based languages can express such kind of policies reasonably well, and they furnish good theoretical background to perform static analysis of possible policy vulnerabilities. In our model [de Oliveira, 2007, Dougherty et al., 2007b], the correspondences between the properties of the rewrite system and the policy it implements are straightforward, thus we are able to directly apply a rich corpus of existing proof techniques and tools.

On the other hand, there are several problems linked to the enforcement of dynamic policies. The classical approaches of building access control mechanisms by inlining reference monitors into the target application's code does not give much insight on how to capture the policy environment, for instance.

The main contribution of this chapter was to introduce a methodology for constructing inlined reference monitors for dynamic access control policies. We have shown that aspect-oriented programming is a suitable tool to capture the policy environment and to transform the untrusted code such that it satisfies a given policy. The method used to inline a monitor enforcing the policy ensures that it cannot be bypassed, relying on the correctness of the aspect compiler. We have illustrated the usefulness of the approach on a realistic example where policy and program are separated. One of the main advantages of the approach we have given is that updates to the policy, for instance adding or removing a rule, do not imply in changes to the code in the aspect nor in the target application. It is necessary to simply recompile the source code with Tom and AspectJ.

Our methodology is general enough to be applied to other pairs of policy specification and programming languages. We have presented rewrite-based policies weaved in Java programs, by using Tom and AspectJ.

Possible extensions of this approach include more fine-tuned analysis for policy enforcement

such as testing the coverage of a policy with respect to the target program, in other words, whether there is some important functionality being executed in a given program, which is not specified in the policy. Further work remains to perform a systematic analysis of the target application, in order to mechanize the construction of the pointcuts and of the code advice to capture the policy environment.

Chapter 8

Conclusions and Perspectives

The clear specification of security policies is of fundamental importance to the overall security goals for a given system. As the security requirements for current information systems involve more and more constraints for defining secure application states, it is necessary that policy formalisms provide both expressivity, and tools for formal policy verification.

In this thesis we have presented a formal model of policies whose core semantics relies in strategic term rewriting. We have shown that this paradigm provides a powerful rule-based language for expressing, and reasoning about policy properties. Using our framework, we are able to define a very large class of policies as we have seen in the numerous examples presented along this manuscript. Its main contributions are summarized in the next section.

8.1 Contributions

We summarize the main contributions we have achieved during the development of this thesis.

Policy Specification The main advantage of our framework for policy specification is to provide expressivity and agility in policy specification thanks to the capabilities of rewrite strategies. Because strategies allow us to have a fine control on rule application, we can easily model real world policies, with priorities among rules, default rules, etc. The originality of the approach also relies on the freedom to specify the core policy elements — decisions, requests, access rules, etc — and on the ability to perform formal reasoning about access control. Based on the properties of the underlying rewrite system, we are able to address consistency check, decision completeness, and termination.

Although checking these properties for most of the examples we presented in the thesis did not require very advanced techniques, they are undecidable in general. That is why we have identified some necessary conditions where these important properties can be checked automatically. This is crucial for improving the confidence in a formal policy model, but syntactical limitations are imposed.

Policy Verification We have given rewrite-based procedures that can be applied to the verification of information leakages in in mandatory access control policies, and for the safety problem in access control models. We have shown in Section 5.1.3 that different implementations of a multilevel security policy can permit prohibited flows to happen. In Section 5.2.1, we have given an approach to check safety in the classic HRU model that

allows one to verify safety at run-time. Moreover, the techniques presented in this thesis can be adapted to check other access control models.

Policy Composition We have shown that strategic rewriting provides a formal background, as well as the necessary expressivity for a policy composition framework. The operational semantics of the strategy language allows us to combine sets of decisions, to give priorities to rules (and to sets of rules), and also to interfere on the flow of information between policies, through a sequencing of requests among policies. We have encoded several existing approaches using rewrite rules thanks to the expressivity provided by our strategy language. For instance, we have shown how to encode conflict resolution operators, and we have translated the full set of XACML combiners in our framework.

Another advantage of our framework is that we can perform formal reasoning for policy composition in the presence of a multiplicity of access decisions, including when policies assign explicitly an "undefined" decision for some request, which cannot be easily captured in previous works. We have also shown the interest of strategic rewriting in the preservation of some policy properties like consistency, termination, and on conservative policy extensions.

Policy Enforcement We have introduced a methodology for constructing inlined reference monitors for rule-based access control policies. We have shown that aspect-oriented programming is a suitable tool to capture the policy environment, and to transform the untrusted code such that it satisfies a given policy. Our methodology is general enough to be applied on various kinds of applications. We have shown the usability of the approach on the enforcement of rewrite-based policies into Java programs, by connecting the Tom system and the aspect-oriented language AspectJ.

8.2 Future Perspectives

Since our approach is recent there are a number of concepts associated to policies and access control for which we do not offer explicit support. For instance, obligations [Dougherty et al., 2007a], which represent the compromise of a certain agent in the future to fulfill some constraint, or to perform a given action. We also do not deal explicitly with administrative issues in policies [Sandhu et al., 1999, Barker and Fernández, 2006, Cuppens and Miège, 2004]. This involves the notion of an entity empowered with privilege to modify or to query the policy itself. We do not tackle trust negotiation either. In trust negotiations [Bonatti and Olmedilla, 2005, Yu et al., 2001] autonomous entities must agree, using some protocol, in a policy about how much information they must disclose mutually in order to achieve some common goal. These topics among others are interesting goals for further research.

Concerning policy specification, we believe that a promising research direction is the impact that *anti-patterns* [Kirchner et al., 2007] can bring to policy specifications. An anti-pattern consist on what one does not want to match. Considering that rules may contain anti-patterns would result in more concise specifications.

With respect to policy analysis, it seems to be interesting to change the perspective from rewriting to narrowing. Such technique can provide powerful static analysis for rule-based poli-

cies. For instance, one would ask what are the possible instantiations of the variables in a given query that can lead to a permission?

An important research direction related to policy enforcement is to consider an automatic approach for enforcing flexible rewrite-based policies. For that, it is necessary to build sophisticated type-checking and type inference algorithms. The reason is that in languages with polymorphic type systems with genericity such as Java, sensible actions can have several forms, making it hard for a reference monitor to intercept them. The authors of rule-based policies should not be obliged to specify all these forms, but give more abstract policy rules, and rely on the underlying enforcement mechanisms.

Several questions remain open with respect to properties for rewrite systems under strategies. For instance, we are not aware of results on sufficient completeness of a rewrite system under an user-defined strategy. We are not aware of the necessary conditions for the modularity of the sufficient completeness property. We do not dispose of algorithms to check confluence under an strategy either. Such developments would be not only interesting for policy specification and composition, but also for other domains that make intensive use of strategic rewriting, such as program transformation.

Appendix A

Policy Implementations with Tom

A.1 Vefication of Information Flow in MAC models

The code below implements the algorithm described in Section 5.1.3.

A.1.1 The McLean's policy implementation

The listing below is a Gom module implementing the signature and some predicates for this policy.

```
module McLean

  imports
    int

  abstract syntax

    Level = top()
          | incomp()
          | bottom()

     Bool = True()
          | False()
          | compare(l1:Level, l2: Level)
          | comparestrict(l1:Level, l2: Level)

    Subject = s(Id:int, l:Level)

    Obj = o(Id:int, l: Level)

    Action = read()
           | write()

    Request = q(p:Privilege)

    Privilege = m(s:Subject, o:Obj, a:Action)

    State = M(Privilege*)

    Decision = auth( q:Request, matrix:State)
             | permit()
             | deny()


    /* Idenpotency */
    M:make_insert(e,L){
      %match(Privilege e, State L){
```

```
        a , M(x*, a, y*) -> {return `M(x*, a, y*); }
      }

    }

    /* Means l1 <= l2  */
    compare:make(l1,l2){
      %match(Level l1, Level l2){
          x , x             -> {return `True();}
          bottom(), top()   -> {return `True();}
          top(), bottom()   -> {return `False();}
          incomp(),_        -> {return `False();}
          _, incomp()       -> {return `False();}
      }
    }


    /* Means l1 < l2  */
    comparestrict:make(l1,l2){
      %match(Level l1, Level l2){
          x, x              -> {return `False();}
          top(), bottom()   -> {return `False();}
          bottom(), top()   -> {return `True();}
          incomp(),_        -> {return `False();}
          _, incomp()       -> {return `False();}
      }
     }
```

A.1.2 The implementation of the verification algorithm

The Tom program below implements both the policy and the transition rules for the policy. The strategy `Flow()` detects implicit flows. The function `checkAccess` verifies if the implicit flows are allowed by the policy.

```
import mclean.types.*;
import java.util.*;
import tom.library.sl.*;

public class Checker{

    %include { sl.tom }
    %include{mclean/McLean.tom}
     public static State derived;


    %strategy Flow() extends `Fail() {
         visit State {
            M(X*,m(s1,o1,read()),Y*) -> {
                 State S = `M(X*,Y*);
                 Subject s = `s1;
                 Obj origin =`o1;
                 %match(State S,  Subject s){
                     M(X2*,m(s,o2,write()),Y2*),s -> {
                          State S1 = `M(X2*,Y2*);
                          Object dest = `o2;
                          State tmp=null;
                          while(tmp!=derived){
                              tmp=derived;
                              %match(State S1, Obj origin, Obj dest) {
                                  M(X3*,m(s2,o4,read()),Y3*,
```

```
                                !m(s2,o3,read()), Z3*), o3, o4 ->{
                                 if (derived != null)
                                     derived = `M(m(s2,origin,read()),derived*);
                                 else
                                     derived=`M(m(s2,origin,read()));
                                 `M(X*,m(s,origin,read()),m(s2,origin,read()),Y*);
                            }
                        }
                    }
                }
            }
        }
    }
}


/*Implements BLP according to McLean without removing rights*/
%strategy McLeanDecisionPolicy() extends `Fail(){
    visit Decision{
        /*Rules 1 to 3 are for reading */
        auth( q(m(s(i1,l1), o(i2,l2), read())),
              M(x*, m(s(i1,l1),o(i3,l3), write()),y*))
            -> {
            /* If the l3 is strictly less than l2, then deny */
            if (`comparestrict(l3,l2)==`True())
                return `deny();
        }

        auth( q(m(s(i1,l1), o(i2,l2), read())), M(x*)) -> {
            if (`compare(l2,l1)==`False())
                return `deny();
        }

        auth( q(m(s(i1,l1), o(i2,l2), read())), M(x*)) -> {
            return `permit();
        }

        /*Rules for the write() action */

        auth( q(m(s(i1,l1), o(i2,l2), write())),
              M(x*, m(s(i1,l1),o(i3,l3), read()),y*))
            -> {
            if (`comparestrict(l2,l3)==`True())
                return `deny();
        }

       auth( q(m(s(i1,l1), o(i2,l2), write())),
              M(x*))
            -> {
            if (`comparestrict(l1,l2)==`False())
                return `deny();
        }

        auth( q(m(s(i1,l1), o(i2,l2), write())), M(x*)) -> {
            return `permit();
        }

    }
}
```

```
public static boolean checkAccess(State matrix){
    Strategy decpol= `McLeanDecisionPolicy();
    Strategy findFlow= `Innermost(Flow());
    try {
        State temp=  (State) findFlow.visit(matrix);
        System.out.println("Derived Access: "+derived);

        while( derived != `M()){
            Privilege req = null;
            Decision res = null;
            %match(State derived) {
                M(x,Y*) -> {
                    req = `x;
                    derived = `M(Y*);
                }
            }
            System.out.println("Derived Request: "+ req);
            res = (Decision) decpol.visit(`auth(q(req),matrix));
            System.out.println("Result "      + res);
            if (res == `permit())
                return true;
            else
                if (res == `deny())
                    return false;
        }
    }catch(Exception e){
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
    return true;

}

public static void main(String args[]){

    int numSub = 1;
    int numObj = 1;

    State matrix = `M();

    try {
        numSub = Integer.parseInt(args[0]);
        numObj = Integer.parseInt(args[1]);
    } catch (Exception e) {
        System.out.println("Usage: java Checker" +
                           " <Number of subjects> <NumberofObjects>");
        return;
    }



    Level [] slevels = {`top(), `incomp(),`bottom()};
    Level [] olevels = {`top(), `incomp(),`bottom() };

    long stopChrono = 0;
    long startChrono = System.currentTimeMillis();
    long step =0;
    for(int i=0; i<numSub; i++){
        Subject s = `s(i, slevels[i%3]);
        for(int j=0; j<numObj;  j++){
            Obj o = `o(j,olevels[j%3]);
            derived=  `M();
```

```
                step++;
                Privilege mr = `m(s,o,read());
                System.out.println("Read request: "+ mr);
                try{
                    matrix = `ConsM(mr,matrix);
                    System.out.println("Matrix: "+ matrix);

                    if(!checkAccess(matrix)){

                        stopChrono = System.currentTimeMillis();
                        System.out.println("An information  flow was found. Total time:"
                                           + (stopChrono-startChrono) + " ms\n");
                        System.out.println("Matrix: "+ matrix);
                        System.out.println("Steps: " +step);
                        System.exit(0);

                    }

                    Privilege mw = `m(s,o,write());
                    System.out.println("Write request: "+ mw);
                    matrix = `ConsM(mw,matrix);
                    System.out.println("Matrix: "+ matrix);

                    if(!checkAccess(matrix)){

                        stopChrono = System.currentTimeMillis();
                        System.out.println("An information  flow was found. Total time:"
                                           + (stopChrono-startChrono) + " ms\n");
                        System.out.println("Matrix: "+ matrix);
                        System.out.println("Steps: " +step);
                        System.exit(0);
                    }

                }catch(Exception e){
                    System.err.println(e.getMessage());
                    e.printStackTrace();
                }
            }
        }

        stopChrono = System.currentTimeMillis();
        System.out.println("No information flow was found. Total time:" +
                           (stopChrono-startChrono) + " ms");
        System.out.println("Matrix: "+ matrix);
        System.out.println("Steps: " +step);
    }
}
```

A.2 A Rewrite-Based Implementation of the HRU Model

The Tom program in this section brings the specification for the HRU model with the effects of commnads over a given configuration. The function `isSafe` checks safety between two different states of the system.

```
import newhru.matrix.types.*;
import newhru.matrix.*;
import tom.library.sl.*;

public class NewHRU{
```

```
%gom{  module Matrix
         imports int boolean String

         abstract syntax

           Right = read()
                 | write()
                 | own()
                 | execute()

           RightList = nilr()
                     | consr(r:Right, rl: RightList)
                     | concr(rl:RightList, rl2:RightList)
                     | getRights(i:int, j: int, ol:ResourceList)

           Subject = s(i:int, rl:RightList)

           SubjectList = nils()
                   | conss(s:Subject, sl: SubjectList)
                   | concs(sl:SubjectList, sl2:SubjectList)
                   | delAllRights(sl:SubjectList)
                   | insRight2(r:Right, i:int, sl:SubjectList)
                   | delSub2(i: int, sl: SubjectList)


           Resource = o(i:int, sl:SubjectList)

           ResourceList = nilo()
                   | conso(o:Resource, ol: ResourceList)
                   | conco(ol:ResourceList, sl2:ResourceList)
                   | insSub(i:int, ol:ResourceList)
                   | insSub2(i:int, ol:ResourceList)
                   | delSub(i:int, ol:ResourceList)
                   | insObj(i:int, ol:ResourceList)
                   | delObj(i:int, ol:ResourceList)
                   | insRight(r:Right, j:int, l:int, ol:ResourceList)
                   | delRight(r:Right, j:int, l:int, ol:ResourceList)



           Instruction = createSubject(i:int)
                       | createObject (i:int)
                       | destroyObject(o:int)
                       | destroySubject(o:int)
                       | enter(r:Right,s:int,o:int)
                       | delete(r:Right,s:int,o:int)

           InstructionList = nili()
                   | consi(o:Instruction, ol: InstructionList)
                   | conci(ol:InstructionList, sl2:InstructionList)

           Condition = True()
                     | False()
                     | inRight(r:Right, rl:RightList)
                     | inSubject(s:Subject, sl:SubjectList)
                     | inResource(i:int, ol:ResourceList)
                     | in(r:Right, x:int , y:int)
                     | inr(r:Right, x:int , y:int, ol: ResourceList)

           ConditionList = nilc()
```

```
              | OK()
              | NotOK()
              | safe(ol1:ResourceList, ol2: ResourceList)
              | consc(c:Condition, cl: ConditionList)
              | concc(cl:ConditionList, cl2:ConditionList)
              | check(cl:ConditionList, ol:ResourceList)

Command = cmd(cl:ConditionList,il:InstructionList)

Config = m(ol:ResourceList)
       | q(cm: Command, ol: ResourceList)

module Matrix:rules(){

 concr(nilr(),L) -> L
 concr(L, nilr())-> L
 concr(consr(x,L),L2) -> consr(x,concr(L,L2))
 consr(x,consr(x,L)) -> consr(x,L)


 concs(nils(),L) -> L
 concs(L, nils())-> L
 concs(conss(x,L),L2) -> conss(x,concs(L,L2))
 conss(x,conss(x,L)) -> conss(x,L)
 conss(s(x,RL1),conss(s(y,RL2),L)) -> conss(s(y,RL2),conss(s(x,RL1),L))
   if x>y


 conco(nilo(),L) -> L
 conco(L, nilo())-> L
 conco(conso(x,L),L2) -> conso(x,conco(L,L2))
 conso(x,conso(x,L)) -> conso(x,L)
 conso(o(x,SL1),conso(o(y,SL2),L)) -> conso(o(y,SL2),conso(o(x,SL1),L))
   if x>y



 conci(nili(),L) -> L
 conci(L, nili())-> L
 conci(consi(x,L),L2) -> consi(x,conci(L,L2))
 consi(x,consi(x,L)) -> consi(x,L)


 concc(nilc(),L) -> L
 concc(L, nilc())-> L
 concc(consc(x,L),L2) -> consc(x,concc(L,L2))
 consc(x,consc(x,L)) -> consc(x,L)


 inRight(x, nilr()) -> False()
 inRight(x, consr(x,RL)) -> True()
 inRight(x, consr(y,RL)) -> inRight(x,RL) if x !=y

 inSubject(x, nils()) -> False()
 inSubject(s(x,RL), conss(s(x,RL2),SL)) -> True()
 inSubject(s(x,RL), conss(s(y,RL2),SL)) -> inSubject(s(x,RL),SL)
   if x > y

 delAllRights(conss(s(i, RL), SL)) -> conss(s(i, nilr()), delAllRights(SL))
 delAllRights(nils()) -> nils()

 inResource(x, nilo()) -> False()
```

```
inResource(x, conso(o(x,SL1),OL)) -> True()
inResource(x, conso(o(y,SL2),OL)) -> inResource(x,OL) if x != y

inr(r, x, y,  nilo()) -> False()
inr(r, x, y,  conso( o(x, nils()), OL)) ->  False()

inr(r, x, y,  conso( o(x, conss( s(y,RL), SL)), OL)) ->
   inRight(r,RL)
inr(r, x, y,  conso( o(z, SL), OL) ) ->
   inr(r, x, y, OL) if x > z

inr(r, x, y,  conso( o(x, conss( s(z,RL), SL)), OL)) ->
   inr(r, x, y, conso( o(x, SL), OL)) if y > z

nilc() -> OK()
consc(True(), nilc()) -> OK()
consc(True(),CL) -> CL
consc(False(),CL) -> NotOK()

 q(cmd( CL, IL), OL) -> m(OL)
 if check(CL, OL) == NotOK()

q(cmd( CL, nili()), OL) -> m(OL)

q(cmd( CL, consi(createSubject(i),IL)),OL) ->
  q(cmd(OK(), IL), insSub(i,OL))
  if check(CL, OL) == OK()

q(cmd( CL, consi(destroySubject(i),IL)),OL) ->
  q(cmd(OK(), IL), delSub(i,OL))
if check(CL, OL) == OK()

q(cmd( CL, consi(createObject(i),IL)),OL) ->
  q(cmd(CL, IL), insObj(i,OL))
  if check(CL, OL) == OK()

q(cmd( CL, consi(destroyObject(i),IL)),OL) ->
  q(cmd(CL, IL), delObj(i,OL))
 if check(CL, OL) == OK()

q(cmd( CL, consi(enter(r,i,j),IL)),OL) ->
  q(cmd(CL, IL), insRight(r,i,j,OL))
    if check(CL, OL) == OK()

q(cmd( CL, consi(delete(r,i,j),IL)),OL) ->
  q(cmd(CL, IL), delRight(r,i,j,OL))
   if check(CL, OL) == OK()

check(OK(),OL) -> OK()
check(CL, nilo()) -> NotOK()
check(consc(in(r,x,y),CL),OL) -> consc(inr(r,x,y,OL), check(CL,OL))

insRight(r, x, y, conso(o(x, conss( s(y, RL), SL)), OL)) ->
   conso(o(x, conss( s(y, consr(r,RL)), SL)), OL)

insRight(r, x, y, conso(o(z, SL), OL)) ->
  conso(o(z, SL), insRight(r, x, y,  OL))  if x > z

insRight(r, x, y, conso(o(x, conss( s(z, RL), SL)), OL)) ->
  conso(o(x, insRight2(r, y, conss( s(z, RL), SL))), OL)
  if y > z
```

```
insRight2(r,i, nils()) -> nils()
insRight2(r,i,conss(s(i,RL),SL)) -> conss(s(i, consr(r, RL)),SL)
insRight2(r,i,conss(s(j,RL),SL)) -> conss(s(j, RL), insRight2(r, i, SL))
  if i > j

delRight(r, x, y, conso(o(x, conss(s(y, consr(r,RL)), SL)), OL)) ->
   conso(o(x, conss( s(y, RL), SL)), OL)

delRight(r, x, y, conso(o(x, conss(s(y, consr(r1,RL)), SL)),OL)) ->
    delRight(r, x, y, conso(o(x, conss(s(y,
       concr(RL,consr(r1,nilr())))), SL)), OL)) if r1 != r

delRight(r, x, y, conso(o(z, SL), OL)) ->
   delRight(r, x, y, conco(OL, conso(o(z, SL), nilo()))) if x != z

delRight(r, x, y, conso(o(x, conss( s(z, RL), SL)), OL)) ->
   delRight(r, x, y, conso(o(x, concs(SL, conss(s(z,RL),nils()))), OL))
   if y > z

insObj(i,  nilo()) -> conso(o(i, nils()), nilo())

insObj(i,  conso(o(x, nils()), OL)) -> conso(o(i, nils()),
   conso(o(x, nils()), OL)) if inResource(i,OL) != True()

insObj(i, conso(o(x, conss(s(i, RL), SL)),  OL)) ->
  conso( o(i, delAllRights(SL)),conso(o(x, conss(s(i, RL), SL)), OL))
  if inResource(i,OL) != True()


insSub(i, nilo()) -> conso( o(i, conss( s(i,nilr()),nils())), nilo())

insSub(i, conso(o(x, SL) ,  OL)) -> conso( o(x, conss(s(i, nilr()), SL)),
 conso(o(i, conss(s(i, nilr()), SL)) , insSub2(i, OL)))
if inResource(i,OL) != True()

insSub2(i, nilo()) -> nilo()
insSub2(i, conso(o(x, SL) ,  OL)) -> conso( o(x, conss(s(i, nilr()), SL)),
   insSub2(i, OL))

delObj(i, nilo()) -> nilo()
delObj(i, conso( o(i,SL), OL)) -> OL
delObj(i, conso( o(j,SL), OL)) -> conso(o(j, SL),delObj(i, OL)) if i != j


delSub(i, nilo()) ->  nilo()
delSub(i, conso(o(i, SL), OL)) ->  delSub(i, OL)
delSub(i, conso(o(j, SL), OL)) ->  conso(o(j, delSub2(i,SL)),
  delSub(i, OL)) if i != j

delSub2(i, nils()) -> nils()
delSub2(i, conss(s(i, RL), SL))-> SL
delSub2(i, conss(s(j, RL), SL))-> conss(s(j, RL), delSub2(i, SL)) if i> j

getRights(x, y, conso(o(x, conss(s(y, RL), SL)), OL)) -> RL
getRights(x, y, conso(o(z, SL), OL)) -> getRights(x, y, OL) if x > z
getRights(x, y, conso(o(x, conss(s(z, RL), SL)), OL)) ->
   getRights(x, y, conso(o(x, SL),OL))

safe(nilo(), OL) -> OK()
safe(conso(o(x, nils()),OL), OL2) -> OK()
safe(conso(o(x, conss(s(y, nilr()),SL)),OL),OL2) ->
  safe(conso(o(x, SL),OL),OL2)
```

```
                    safe(conso(o(x, conss(s(y, consr(r,RL)),SL)),OL),OL2) ->
                      consc(inr(r,x,y,OL2), safe(conso(o(x, conss(s(y,RL),SL)),OL), OL2))

                  }
              }
 }
```

A.3 A Policy Composition Algebra in Tom

The program below encodes the composition algebra of [Bonatti et al., 2002]. The operator identifiers, and the syntax change with respect to Table 6.1 because Tom does not support AC matching, and no infix operators. Then, + is called `union`, & is called `inter`, − is named `diff`, and *o* corresponds to `over`. The list handling functions : and ++ correspond respectively to `cons` and `conc`.

```
import sets.set.*;
import sets.set.types.*;
import tom.library.sl.*;


public class Sets{
      %include{sl.tom}

      %gom{
            module Set
            imports int boolean
              abstract syntax

              Access = permit(subject:int, object:int, action:int)


              List = cons(a:Access, l:List)
                   | nil()
                   | conc(l:List,l2:List)
                   | union(s1:List, s2:List)
                   | inter(s1:List, s2:List)
                   | diff(s1:List, s2:List)
                   | over(s1:List, s2:List, s3:List)

              Bool = in(i: Access, l:List)
                    | True()
                    | False()

               inter:make(l1,l2){
                  %match(List l1, List l2){
                     L, nil() -> {return `nil();}
                     nil(), L -> {return `nil();}
                     cons(x,L1),cons(x,L2) -> {return
                            `union(cons(x, nil()),inter(L1,L2));
                     }
                     cons(x,L1),L -> {
                         if (`in(x,L)==`False())
                            return `inter(L1,L);
                         else
                            return
                            `union(cons(x, nil()),inter(L1,L));
                     }
                     cons(x,nil()),cons(y,nil()) -> {
```

```
                    if (`x != `y) return `nil(); }
                }
            }

            diff:make(l1,l2) {
              %match(List l1, List l2){
                 nil(),L -> {return `nil();}
                 L, nil()-> {return `L;}
                 cons(x,L),cons(x,L1)-> {return `diff(L,L1);}
                 cons(x,nil()),cons(y,nil()) -> {
                      if (`x!=`y) return `cons(x, nil());}
                 cons(x,L1), L2 -> {
                    if (`in(x,L2)==`False())
                      return `cons(x, diff(L1,L2));
                    else
                      return `diff(L1,L2);
                    }
              }
            }
            module Set:rules() {
                 in(x, nil()) -> False()
                 in(x, cons(x,L)) -> True()
                 in(x, cons(y,L)) -> in(x,L) if x !=y

                 conc(nil(),L) -> L
                 conc(L, nil())-> L
                 conc(cons(x,L),L2) -> cons(x,conc(L,L2))
                 cons(x,cons(x,L)) -> cons(x,L)

                 union(L, nil()) -> L
                 union(nil(), L) -> L
                 union(cons(x,L1),cons(y,L2)) -> cons(x,cons(y,union(L1,L2)))

                 over(L1, L2, L3) -> inter(diff(L1,L2), union(L2,L3))

            }
        }

    public static List closure(List aut, Strategy strat){
       try{
          return (List) strat.visitLight(aut);
        }catch(Exception e){
           System.err.println("Exception in strategy application: "+ e.getMessage());
        }
        return aut;
    }

}
```

A.4 The Conference Policy in Tom

We list here the conference policy as programmed in Tom. The first part of the code, contained in the Gom definition, describes the signature of the rewrite system. The reader may identify some additional sorts w.r.t Definition 42, which are used to describe the policy environment. The second part of the code contains the rewrite rules defining the aut function, which relies on the pattern matching provided by Tom.

```
package conference;
```

```
import conference.policy.conference.types.*;
import tom.library.sl.*;
import jjtraveler.reflective.VisitableVisitor;
import jjtraveler.Visitable;
import jjtraveler.VisitFailure;

public class Policy{

 %include{sl.tom}

 %gom{
   module Conference

    imports String int

    abstract syntax
      Decision = permit()
      | deny()
      | notApplicable()
      | aut(r: Request, p:Phase, cnd:Condition)

      Request = q(s:Subject, a:Action, o: Obj)

      Phase = submission() | meeting() | review()

      Action = submitPaper()
      | readScores()
      | submitReview()
      | assignPaper()
      | addReviewer()

      Subject = author(id:int)
      | reviewer(id:int)
      | chair(id:int)

      Obj = paper(id:int, title:String)

      Condition = assigned(id:int, o:Obj)
      | conflict(id:int, o:Obj)
    }

    %strategy Rules() extends Fail(){
      visit Decision{

       aut( q(author(x), submitPaper(), paper(x, z)),
            submission(), cnd) -> {return `permit();}

       aut( q(author(x), submitPaper(), paper(x, z)),
            phase, cnd) -> {return `deny();}

       aut(q(author(x), readScores(), paper(x,z)),
           phase, cnd)  -> {return `deny();}

       aut(q(reviewer(x),action, p), phase,
           conflict(x,p)) -> { return `deny(); }

       aut(q(reviewer(x), submitReview(), paper(y,z)),
           review(), assigned(x, paper(y,z)))
       -> {return `permit();}

       aut(q(reviewer(x), submitReview(), paper(y,z)),
           phase, assigned(x, paper(y,z)))
```

```
    -> {return `deny();}

    aut(q(reviewer(x), readScores(), paper(y,z)),
        meeting(), assigned(x, paper(y,z)))
    -> {return `permit();}

    aut(q(reviewer(x), action, paper(x,z)),
    phase , cnd ) -> { return
      `aut(q(reviewer(x), action, paper(x,z)),
           phase, conflict(x, paper(x,z))); }

    aut(_,_,_) -> {return `notApplicable();}
  }
}

public static boolean apply(int sId, int oId,
    String role, int phase, boolean assign,
    String action){

    Subject s;
    Action a = `readScores();
    Obj o = `paper(oId,"");
    Phase p = `submission();
    Condition cnd = `conflict(sId,o);

    if (role.equals("Author")){
        s = `author(sId);
    }else{
        if (role.equals("Reviewer")){
            s=`reviewer(sId);
        }
        else{
            s=`chair(sId);
        }
    }

    if (assign) cnd = `assigned(sId, o);

    switch(phase){
    case 0: {p = `submission();break; }
    case 1: {p = `review();break;}
    case 2: {p = `meeting();break;}
    }

    if (action.equals("submitPaper")){
        a = `submitPaper();
    }
    else{
        if(action.equals("readScores")){
            a = `readScores();
        }else{
            if (action.equals("submitReview")){
                a=`submitReview();
            }
        }
    }

    Strategy policy = `Innermost(Rules());
    try{
       Decision d = (Decision)
        policy.visit(`aut(q(s, a, o), p, cnd));
        if(d == `permit())
```

```
            return true;
        else
            return false;
      }catch(Exception e){
          System.err.println(e.getMessage());
      }
      return false;
    }
  }
```

A.5 AspectJ Code for the Conference System

The following code brings the AspectJ implementation for weaving the conference policy of Example 28 into the conference management system.

```
package conference;
import java.*;
import org.aspectj.lang.*;

aspect PolicyAspect {

  private int phase;
  private User  usr;
  private int  paperId;

  pointcut readScoresCut():
    call(int Paper.readScores(int));

  pointcut submitPaperCut(Paper pa):
    call(void Conference.submitPaper(Paper))
   && args(pa);

  pointcut submitReviewCut(int objId,int revId):
    call(void Conference.submitReview(int,int,int))
   && args(objId,revId,*);

  pointcut loginCut():
    call(void Conference.login(String,int,String));

  pointcut readScoreCut(): call(int Paper.readScore(int));

  after(User user): set(
    User Conference.currentUser)  && args(user){
    usr=user;
  }

  after(int cp): set(int Conference.currentPhase)
   && args(cp){
    phase=cp;
  }

  before(Paper p): submitPaperCut(p){
    paperId=p.getId();
    if(!Policy.apply(usr.getId(),paperId,
       usr.getRole(),phase, true,"submitPaper")){
       System.out.println("Access␣Denied.");
       System.exit(1);
    }
  }
```

```
  before(int oi,int ri):submitReviewCut(oi,ri){
    paperId=oi;
    if(!Policy.apply(usr.getId(),paperId,
        usr.getRole(),phase, true,"submitReview")){
        System.out.println("Access␣Denied.");
        System.exit(1);
      }
    }

   before():readScoresCut(){
     if(!Policy.apply(usr.getId(),paperId,
          usr.getRole(),phase, true,"readScores")){
         System.out.println("Access␣Denied.");
         System.exit(1);
     }
   }
}
```

Bibliography

[Abadi et al., 1993] Abadi, M., Burrows, M., Lampson, B. W., and Plotkin, G. D. (1993). A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734.

[Antoniou et al., 2007] Antoniou, G., Baldoni, M., Bonatti, P. A., Nejdl, W., and Olmedilla, D. (2007). Rule-based policy specification. In Yu, T. and Jajodia, S., editors, *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*. Springer.

[Antoy et al., 2000] Antoy, S., Echahed, R., and Hanus, M. (2000). A needed narrowing strategy. *J. ACM*, 47(4):776–822.

[Arts and Giesl, 2000] Arts, T. and Giesl, J. (2000). Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178.

[Avgustinov et al., 2005] Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Optimising aspectj. In [Sarkar and Hall, 2005], pages 117–128.

[Baader, 2007] Baader, F., editor (2007). *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 25-29, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*. Springer.

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and all That*. Cambridge University Press.

[Baader and Snyder, 2001] Baader, F. and Snyder, W. (2001). Unification theory. In Robinson, J. A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press.

[Balland et al., 2006a] Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. (2006a). Tom Manual. LORIA, Nancy (France), version 2.4 edition.

[Balland et al., 2007] Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. (2007). Tom: Piggybacking rewriting on java. In [Baader, 2007], pages 36–47.

[Balland et al., 2006b] Balland, E., Kirchner, C., and Moreau, P.-E. (2006b). Formal islands. In Johnson, M. and Vene, V., editors, *AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 51–65. Springer.

[Balland et al., 2006c] Balland, E., Kirchner, C., Moreau, P.-E., and de Oliveira, A. S. (2006c). Modular formal islands: Embed theory in your practice. In *Proceedings of the Third Taiwanese-French Conference on Information Technology*. LORIA, INRIA.

[Balland and Moreau, 2006] Balland, E. and Moreau, P.-E. (2006). Optimizing pattern matching compilation by program transformation. In Favre, J.-M., Heckel, R., and Mens, T., editors, *3rd Workshop on Software Evolution through Transformations (SeTra'06)*. Electronic Communications of EASST. To appear.

[Bancilhon et al., 1986] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15. ACM.

[Barahona and Felty, 2005] Barahona, P. and Felty, A. P., editors (2005). *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*. ACM.

[Barker and Fernández, 2006] Barker, S. and Fernández, M. (2006). Term rewriting for access control. In Damiani, E. and Liu, P., editors, *DBSec*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer.

[Barker and Stuckey, 2003] Barker, S. and Stuckey, P. J. (2003). Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546.

[Bauer et al., 2007] Bauer, L., Garriss, S., and Reiter, M. K. (2007). Efficient proving for practical distributed access-control systems. In [Biskup and Lopez, 2007], pages 19–37.

[Bauer et al., 2002] Bauer, L., Ligatti, J., and Walker, D. (2002). A calculus for composing security policies. Technical Report TR-655-02, Princeton University.

[Bauer et al., 2005] Bauer, L., Ligatti, J., and Walker, D. (2005). Composing security policies with polymer. In [Sarkar and Hall, 2005], pages 305–314.

[Becker and Nanz, 2007] Becker, M. Y. and Nanz, S. (2007). A logic for state-modifying authorization policies. In [Biskup and Lopez, 2007], pages 203–218.

[Becker and Sewell, 2004a] Becker, M. Y. and Sewell, P. (2004a). Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY*, pages 159–168. IEEE Computer Society.

[Becker and Sewell, 2004b] Becker, M. Y. and Sewell, P. (2004b). Cassandra: Flexible trust management, applied to electronic health records. In *CSFW*, pages 139–154. IEEE Computer Society.

[Bell and LaPadula, 1996] Bell, D. E. and LaPadula, L. J. (1996). Secure computer systems: A mathematical model, volume ii. *Journal of Computer Security*, 4(2/3):229–263.

[Bell and LaPadula, 1974] Bell, E. D. and LaPadula, L. J. (1974). Secure computer systems: Mathematical foundations. Technical Report Mitre Report ESD-TR-73-278 (Vol. I-III), Mitre Corporation.

[Benanav et al., 1987] Benanav, D., Kapur, D., and Narendran, P. (1987). Complexity of matching problems. *J. Symb. Comput.*, 3(1/2):203–216.

[Bertino et al., 1998] Bertino, E., Bettini, C., Ferrari, E., and Samarati, P. (1998). An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3):231–285.

[Bertolissi et al., 2007] Bertolissi, C., Fernández, M., and Barker, S. (2007). Dynamic event-based access control as term rewriting. In Barker, S. and Ahn, G.-J., editors, *DBSec*, volume 4602 of *Lecture Notes in Computer Science*, pages 195–210. Springer.

[Biba, 1977] Biba, K. (1977). Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA.

[Birkhoff, 1935] Birkhoff, G. (1935). On the structure of abstract algebras. *Proc. Camb. Philos. Soc*, 31:433–454.

[Bishop, 2004] Bishop, M. (2004). *Introduction to Computer Security*. Addison-Wesley Professional.

[Biskup and Lopez, 2007] Biskup, J. and Lopez, J., editors (2007). *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*. Springer.

[Boichut et al., 2007] Boichut, Y., Genet, T., Jensen, T. P., and Roux, L. L. (2007). Rewriting approximations for fast prototyping of static analyzers. In [Baader, 2007], pages 48–62.

[Bonatti et al., 2002] Bonatti, P. A., di Vimercati, S. D. C., and Samarati, P. (2002). An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35.

[Bonatti and Olmedilla, 2005] Bonatti, P. A. and Olmedilla, D. (2005). Driving and monitoring provisional trust negotiation with metapolicies. In *POLICY*, pages 14–23. IEEE Computer Society.

[Bonatti and Samarati, 2002] Bonatti, P. A. and Samarati, P. (2002). A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 10(3):241–272.

[Borovanský et al., 1998] Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Ringeissen, C. (1998). An overview of elan. *Electr. Notes Theor. Comput. Sci.*, 15.

[Borovansky et al., 2001] Borovansky, P., Kirchner, C., Kirchner, H., and Ringeissen, C. (2001). Rewriting with strategies in elan: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95.

[Bouhoula, 1994] Bouhoula, A. (1994). Spike: a system for sufficient completeness and parameterized inductive proofs. In Bundy, A., editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 836–840. Springer.

[Bouhoula and Jacquemard, 2006] Bouhoula, A. and Jacquemard, F. (2006). Automatic verification of sufficient completeness for conditional constrained term rewriting systems. Technical Report RR-5863, INRIA.

[Brewer and Nash, 1989] Brewer, D. F. C. and Nash, M. J. (1989). The chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214.

[Bruns et al., 2007] Bruns, G., Dantas, D. S., and Huth, M. (2007). A simple and expressive semantic framework for policy composition in access control. In *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 12–21, New York, NY, USA. ACM.

[Ceri et al., 1989] Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166.

[Ceri et al., 1990] Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Databases*. Springer-Verlag New York, Inc. New York, NY, USA.

[Cholvy and Cuppens, 1997] Cholvy, L. and Cuppens, F. (1997). Analyzing consistency of security policies. In *IEEE Symposium on Security and Privacy*, pages 103–112. IEEE Computer Society.

[Cirstea and Kirchner, 2001] Cirstea, H. and Kirchner, C. (2001). The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498.

[Cirstea et al., 2003] Cirstea, H., Kirchner, C., Liquori, L., and Wack, B. (2003). Rewrite strategies in the rewriting calculus. In Gramlich, B. and Lucas, S., editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier.

[Cirstea et al., 2005] Cirstea, H., Moreau, P.-E., and Reilles, A. (2005). Rule-based programming in java for protocol verification. *Electr. Notes Theor. Comput. Sci.*, 117:209–227.

[Clavel et al., 2002] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. (2002). Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243.

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., editors (2007). *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer.

[Comon, 1986] Comon, H. (1986). Sufficient completness, term rewriting systems and "anti-unification". In Siekmann, J. H., editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer.

[Cuppens et al., 2007] Cuppens, F., Cuppens-Boulahia, N., and Ghorbel, M. (2007). High Level Conflict Management Strategies in Advanced Access Control Models. *Electronic Notes in Theoretical Computer Science*, 186:3–26.

[Cuppens et al., 2006] Cuppens, F., Cuppens-Boulahia, N., and Ramard, T. (2006). Availability enforcement by obligations and aspects identification. In *ARES*, pages 229–239. IEEE Computer Society.

[Cuppens et al., 2005] Cuppens, F., Cuppens-Boulahia, N., and Sans, T. (2005). Nomad: A security model with non atomic actions and deadlines. In *CSFW*, pages 186–196. IEEE Computer Society.

[Cuppens and Miège, 2004] Cuppens, F. and Miège, A. (2004). Adorbac: an administration model for or-bac. *Comput. Syst. Sci. Eng.*, 19(3).

[Damianou et al., 2001] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The ponder policy specification language. In Sloman, M., Lobo, J., and Lupu, E., editors, *POLICY*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer.

[Dantas, 2007] Dantas, D. S. (2007). *Analyzing Security Advice in Functional Aspect-oriented Programming Languages*. PhD thesis, Princeton University.

[Dauchet, 1992] Dauchet, M. (1992). Simulation of turing machines by a regular rewrite rule. *Theor. Comput. Sci.*, 103(2):409–420.

[Dauchet et al., 1990] Dauchet, M., Heuillard, T., Lescanne, P., and Tison, S. (1990). Decidability of the confluence of finite ground term rewrite systems and of other related term rewrite systems. *Inf. Comput.*, 88(2):187–201.

[de Oliveira, 2007] de Oliveira, A. S. (2007). Rewriting-based access control policies. *Electr. Notes Theor. Comput. Sci.*, 171(4):59–72.

[de Oliveira et al., 2007] de Oliveira, A. S., Wang, E. K., Kirchner, C., and Kirchner, H. (2007). Weaving rewrite-based access control policies. In *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 71–80, New York, NY, USA. ACM.

[Dershowitz, 1982] Dershowitz, N. (1982). Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301.

[Dershowitz, 1987] Dershowitz, N. (1987). Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116.

[Dershowitz, 1994] Dershowitz, N. (1994). Hierarchical termination. In *Proceedings 4th International Workshop on Conditional Term Rewriting Systems, Jerusalem (Israel)*, volume 968 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag.

[di Vimercati et al., 2005] di Vimercati, S. D. C., Samarati, P., and Jajodia, S. (2005). Policies, models, and languages for access control. In Bhalla, S., editor, *DNIS*, volume 3433 of *Lecture Notes in Computer Science*, pages 225–237. Springer.

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

[Dougherty et al., 2007a] Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2007a). Obligations and their interaction with programs. In [Biskup and Lopez, 2007], pages 375–389.

[Dougherty et al., 2007b] Dougherty, D. J., Kirchner, C., Kirchner, H., and de Oliveira, A. S. (2007b). Modular access control via strategic rewriting. In [Biskup and Lopez, 2007], pages 578–593.

[Echahed and Prost, 2005] Echahed, R. and Prost, F. (2005). Security policy in a declarative style. In [Barahona and Felty, 2005], pages 153–163.

[Erlingsson and Schneider, 2000] Erlingsson, U. and Schneider, F. B. (2000). Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA. ACM Press.

[Evans and Twyman, 1999] Evans, D. and Twyman, A., editors (1999). *Flexible Policy-Directed Code Safety, IEEE Symposium on Security and Privacy, 1999*. IEEE Computer Society.

[Fisler et al., 2005] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. (2005). Verification and change-impact analysis of access-control policies. In Roman, G.-C., Griswold, W. G., and Nuseibeh, B., editors, *ICSE*, pages 196–205. ACM.

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

[Gavrila and Barkley, 1998] Gavrila, S. and Barkley, J. (1998). Formal specification for RBAC user/role and role/role relationship management. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC-98)*, pages 81–90. ACM Press.

[Genet, 1998] Genet, T. (1998). *Contraintes d'ordre et automates d'arbres pour les preuves de terminaison*. PhD thesis, Université Henri Poincaré - Nancy I.

[Giesl et al., 2004] Giesl, J., Thiemann, R., Schneider-Kamp, P., and Falke, S. (2004). Automated termination proofs with aprove. In van Oostrom, V., editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer.

[Gnaedig, 2008] Gnaedig, I. (2008). Termination of priority rewriting. Submitted.

[Gnaedig and Kirchner, 2006] Gnaedig, I. and Kirchner, H. (2006). Computing constructor forms with non terminating rewrite programs. In Bossi, A. and Maher, M. J., editors, *PPDP*, pages 121–132. ACM.

[Godoy and Huntingford, 2007] Godoy, G. and Huntingford, E. (2007). Innermost-reachability and innermost-joinability are decidable for shallow term rewrite systems. In [Baader, 2007], pages 184–199.

[Godoy et al., 2007] Godoy, G., Huntingford, E., and Tiwari, A. (2007). Termination of rewriting with right-flat rules. In [Baader, 2007], pages 200–213.

[Godoy et al., 2004] Godoy, G., Nieuwenhuis, R., and Tiwari, A. (2004). Classes of term rewrite systems with polynomial confluence problems. *ACM Trans. Comput. Log.*, 5(2):321–331.

[Godoy and Tiwari, 2005a] Godoy, G. and Tiwari, A. (2005a). Confluence of shallow right-linear rewrite systems. In Ong, C.-H. L., editor, *CSL*, volume 3634 of *Lecture Notes in Computer Science*, pages 541–556. Springer.

[Godoy and Tiwari, 2005b] Godoy, G. and Tiwari, A. (2005b). Termination of rewrite systems with shallow right-linear, collapsing, and right-ground rules. In Nieuwenhuis, R., editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 164–176. Springer.

[Goguen and Malcolm, 2000] Goguen, J. and Malcolm, G. (2000). *Software Engineering with Obj: Algebraic Specification in Action*. Kluwer Academic Publishers.

[Goguen and Meseguer, 1982] Goguen, J. A. and Meseguer, J. (1982). Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20.

[Gong et al., 2003] Gong, L., Ellison, G., and Dageforde, M. (2003). *Inside Java 2 Platform Security: Architecture, Api Design, and Implementation*. Addison-Wesley Professional.

[Gramlich, 1992] Gramlich, B. (1992). Generalized sufficient conditions for modular termination of rewriting. In Kirchner, H. and Levi, G., editors, *Proceedings of the 3rd Algebraic and Logic Programming Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag.

[Gramlich, 1996] Gramlich, B. (1996). On proving termination by innermost termination. In Ganzinger, H., editor, *RTA*, volume 1103 of *Lecture Notes in Computer Science*, pages 93–107. Springer.

[Guelev et al., 2004] Guelev, D. P., Ryan, M., and Schobbens, P.-Y. (2004). Model-checking access control policies. In Zhang, K. and Zheng, Y., editors, *ISC*, volume 3225 of *Lecture Notes in Computer Science*, pages 219–230. Springer.

[Halpern and Weissman, 2003] Halpern, J. Y. and Weissman, V. (2003). Using first-order logic to reason about policies. In *CSFW*, pages 187–201. IEEE Computer Society.

[Hamlen, 2006] Hamlen, K. (2006). *Security Policy Enforcement By automated Program-Rewriting*. Phd thesis, Cornell University.

[Harrison et al., 1976] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

[Heintze and Riecke, 1998] Heintze, N. and Riecke, J. G. (1998). The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377.

[Hirokawa and Middeldorp, 2007] Hirokawa, N. and Middeldorp, A. (2007). Tyrolean termination tool: Techniques and features. *Inf. Comput.*, 205(4):474–511.

[Huet, 1980] 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.

[Hullot, 1979] Hullot, J.-M. (1979). Associative-commutative pattern matching. In *Proceedings 9th International Joint Conference on Artificial Intelligence*.

[Humenn, 2003] Humenn, P. (2003). The formal semantics of XACML.

[Jacquemard, 2003] Jacquemard, F. (2003). Reachability and confluence are undecidable for flat term rewriting systems. *Inf. Process. Lett.*, 87(5):265–270.

[Jaeger and Tidswell, 2001] Jaeger, T. and Tidswell, J. (2001). Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, 4(2):158–190.

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

[Jaume and Morisset, 2006a] Jaume, M. and Morisset, C. (2006a). A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148.

[Jaume and Morisset, 2006b] Jaume, M. and Morisset, C. (2006b). Towards a formal specification of access control. In Degano, P., Kusters, R., Vigano, L., and Zdancewic, S., editors, *Proceedings of FCS-ARSPA'06*, pages 213–232.

[Jouannaud and Kirchner, 1986] Jouannaud, J.-P. and Kirchner, H. (1986). Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194.

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

[Kapur et al., 1991] Kapur, D., Narendran, P., Rosenkrantz, D. J., and Zhang, H. (1991). Sufficient-completeness, ground-reducibility and their complexity. *Acta Inf.*, 28(4):311–350.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In Knudsen, J. L., editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP*, pages 220–242.

[Kirchner, 2005] Kirchner, C. (2005). Strategic rewriting. *Electr. Notes Theor. Comput. Sci.*, 124(2):3–9.

[Kirchner et al., 2008] Kirchner, C., Kirchner, F., and Kirchner, H. (2008). Strategic computations and deductions. Accepted for publication.

[Kirchner et al., 1995] Kirchner, C., Kirchner, H., and Vittek, M. (1995). Designing CLP using computational systems. In Hentenryck, P. V. and Saraswat, S., editors, *Principles and Practice of Constraint Programming*, chapter 8, pages 133–160. MIT press.

[Kirchner et al., 2007] Kirchner, C., Kopetz, R., and Moreau, P.-E. (2007). Anti-pattern matching. In Nicola, R. D., editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124. Springer.

[Kirchner et al., 2005] Kirchner, C., Moreau, P.-E., and Reilles, A. (2005). Formal validation of pattern matching code. In [Barahona and Felty, 2005], pages 187–197.

[Knuth and Bendix, 1970] Knuth, D. E. and Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J., editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford.

[Koch et al., 2002] Koch, M., Mancini, L. V., and Parisi-Presicce, F. (2002). Decidability of safety in graph-based models for access control. In Gollmann, D., Karjoth, G., and Waidner, M., editors, *ESORICS*, volume 2502 of *Lecture Notes in Computer Science*, pages 229–243. Springer.

[Kurihara and Ohuchi, 1990] Kurihara, M. and Ohuchi, A. (1990). Modularity of simple termination of term rewriting systems. *Journal of IPS Japan*, 31(5):633–642.

[Kurihara and Ohuchi, 1992] Kurihara, M. and Ohuchi, A. (1992). Modularity of simple termination of term rewriting systems with shared constructors. *Theor. Comput. Sci.*, 103(2):273–282.

[Lampson, 1974] Lampson, B. W. (1974). Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24.

[Lee et al., 2006] Lee, A. J., Boyer, J. P., Olson, L., and Gunter, C. A. (2006). Defeasible security policy composition for web services. In Winslett, M., Gordon, A. D., and Sands, D., editors, *FMSE*, pages 45–54. ACM.

[Li and Mitchell, 2003] Li, N. and Mitchell, J. C. (2003). Datalog with constraints: A foundation for trust management languages. In Dahl, V. and Wadler, P., editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer.

[Ligatti et al., 2005] Ligatti, J., Bauer, L., and Walker, D. (2005). Enforcing non-safety security policies with program monitors. In di Vimercati, S. D. C., Syverson, P. F., and Gollmann, D., editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373. Springer.

[Liu, 2007] Liu, A. X. (2007). Change-impact analysis of firewall policies. In [Biskup and Lopez, 2007], pages 155–170.

[Marché and Urbain, 2004] Marché, C. and Urbain, X. (2004). Modular and incremental proofs of ac-termination. *J. Symb. Comput.*, 38(1):873–897.

[Martí-Oliet and Meseguer, 2002] Martí-Oliet, N. and Meseguer, J. (2002). Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154.

[Martí-Oliet et al., 2005] Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2005). Towards a strategy language for maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441.

[McLean, 1988] McLean (1988). The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press.

[McLean, 1985] McLean, J. (1985). A comment on the 'basic security theorem' of Bell and LaPadula. *Information Processing Letters*, 20(2):67–70.

[Middeldorp, 1989] Middeldorp, A. (1989). A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 396–401.

[Middeldorp and Toyama, 1991] Middeldorp, A. and Toyama, Y. (1991). Completeness of combinations of constructor systems. In *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*. also Report CS-R9058, CWI, 1990.

[Miège, 2005] Miège, A. (2005). *Definition of a formal framework for specifying security policies. The Or-BAC model and extensions.* PhD thesis, Ecole Nationale Supérieur des Télécommunications - Telecom Paris.

[Morisset, 2007] Morisset, C. (2007). *Définition d'un cadre sémantique pour la spécification, l'implantation et la comparaison de modèles de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie.

[Morisset and de Oliveira, 2007] Morisset, C. and de Oliveira, A. S. (2007). Automated detection of information leakage in access control. In Monica Nesi, editor, *Second International Workshop on Security and Rewriting Techniques - SecReT 2007*, Paris, France.

[Moses, 2005] Moses, T. (2005). Extensible access control markup language (XACML) version 2.0. Technical report, OASIS.

[Myers, 1999] Myers, A. C. (1999). Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241.

[Naldurg et al., 2006] Naldurg, P., Schwoon, S., Rajamani, S. K., and Lambert, J. (2006). *NETRA*: : seeing through access control. In Winslett, M., Gordon, A. D., and Sands, D., editors, *FMSE*, pages 55–66. ACM.

[Ohlebusch, 2002a] Ohlebusch, E. (2002a). *Advanced Topics in Term Rewriting*. Springer.

[Ohlebusch, 2002b] Ohlebusch, E. (2002b). Hierarchical termination revisited. *Inf. Process. Lett.*, 84(4):207–214.

[Park and Sandhu, 2004] Park, J. and Sandhu, R. (2004). The $UCON_{ABC}$ usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174.

[Paterson and Wegman, 1978] Paterson, M. S. and Wegman, M. N. (1978). Linear unification. *Journal of Computer and System Sciences*, 16:158–167.

[Pavlich-Mariscal et al., 2005] Pavlich-Mariscal, J. A., Michel, L., and Demurjian, S. A. (2005). A formal enforcement framework for role-based access control using aspect-oriented programming. In Briand, L. C. and Williams, C., editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 537–552. Springer.

[Peterson and Stickel, 1981] Peterson, G. E. and Stickel, M. E. (1981). Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264.

[Reilles, 2007] Reilles, A. (2007). Canonical abstract syntax trees. *Electr. Notes Theor. Comput. Sci.*, 176(4):165–179.

[Rusinowitch, 1987] Rusinowitch, M. (1987). On termination of the direct sum of term-rewriting systems. *Inf. Process. Lett.*, 26(2):65–70.

[Ryan, 2000] Ryan, P. (2000). Mathematical Models of Computer Security. *Foundations of Security Analysis and Design*, 2171.

[Sabelfeld, 2003] Sabelfeld, A.; Myers, A. (Jan 2003). Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19.

[Sandhu et al., 1999] Sandhu, R., Bhamidipati, V., and Munawer, Q. (1999). The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135.

[Sandhu et al., 2000] Sandhu, R., Ferraiolo, D., and Kuhn, R. (2000). The NIST model for role-based access control: towards a unified standard. In *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, New York, NY, USA. ACM Press.

[Sandhu, 1993] Sandhu, R. S. (1993). Lattice-based access control models. *IEEE Computer*, 26(11):9–19.

[Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29(2):38–47.

[Sarkar and Hall, 2005] Sarkar, V. and Hall, M. W., editors (2005). *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. ACM.

[Schneider, 2000] Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

[Schneider-Kamp et al., 2007] Schneider-Kamp, P., Giesl, J., Serebrenik, A., and Thiemann, R. (2007). Automated termination analysis for logic programs by term rewriting. In Puebla, G., editor, *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 177–193. Springer.

[Song et al., 2005] Song, E., Reddy, R., France, R. B., Ray, I., Georg, G., and Alexander, R. (2005). Verifiable composition of access control and application features. In Ferrari, E. and Ahn, G.-J., editors, *SACMAT*, pages 120–129. ACM.

[Toyama, 1987a] Toyama, Y. (1987a). Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25(3):141–143.

[Toyama, 1987b] Toyama, Y. (1987b). On the church-rosser property for the direct sum of term rewriting systems. *J. ACM*, 34(1):128–143.

[Tschantz and Krishnamurthi, 2006] Tschantz, M. C. and Krishnamurthi, S. (2006). Towards reasonability properties for access-control policy languages. In Ferraiolo, D. F. and Ray, I., editors, *SACMAT*, pages 160–169. ACM.

[Visser, 2001] Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. In Middeldorp, A., editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer.

[Wang and Sakai, 2006] Wang, Y. and Sakai, M. (2006). Decidability of termination for semi-constructor TRSs, left-linear shallow TRSs and related systems. In Pfenning, F., editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 343–356. Springer.

[Wijesekera and Jajodia, 2003] Wijesekera, D. and Jajodia, S. (2003). A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325.

[Winslett et al., 2005] Winslett, M., Zhang, C. C., and Bonatti, P. A. (2005). PeerAccess: a logic for distributed authorization. In Atluri, V., Meadows, C., and Juels, A., editors, *ACM Conference on Computer and Communications Security*, pages 168–179. ACM.

[Yu et al., 2001] Yu, T., Winslett, M., and Seamons, K. E. (2001). Interoperable strategies in automated trust negotiation. In *ACM Conference on Computer and Communications Security*, pages 146–155.

[Zantema, 1995] Zantema, H. (1995). Termination of term rewriting by semantic labelling. *Fundam. Inform.*, 24(1/2):89–105.

Résumé

Dans cette thèse, nous nous intéressons à la spécification et à l'analyse modulaires de politiques de sécurité flexibles basées sur des règles. Nous introduisons l'utilisation du formalisme de réécriture stratégique dans ce domaine, afin que notre cadre hérite des techniques, des théorèmes, et des outils de la théorie de réécriture. Ceci nous permet d'énoncer facilement et de vérifier des propriétés importantes des politiques de sécurité telles que l'absence des conflits. En outre, nous développons des méthodes basées sur la réécriture de termes pour vérifier des propriétés plus élaborées des politiques. Ces propriétés sont la sûreté dans le contrôle d'accès et la détection des flux d'information dans des politiques obligatoires.

Par ailleurs, nous montrons que les stratégies de réécriture sont importantes pour préserver des propriétés des politiques par composition. Les langages de stratégies disponibles dans des systèmes comme Tom, Stratego, Maude, ASF+SDF et ELAN, nous permettent de définir plusieurs genres de combinateurs de politiques.

Enfin, nous fournissons également une méthodologie pour imposer à des applications existantes, de respecter des politiques basées sur la réécriture via la programmation par aspects. Les politiques sont tissées dans le code existant, ce qui génère des programmes mettant en oeuvre des moniteurs de référence. La réutilisation des politiques et des programmes est améliorée car ils peuvent être maintenus indépendamment.

Mots clefs :
Politiques de Sécurité, Composition, Contrôle d'Accès, Réécriture, Stratégies.

Abstract

In this thesis we address the modular specification and analysis of flexible, rule-based policies. We introduce the use of the strategic rewriting formalism in this domain, such that our framework inherits techniques, theorems, and tools from the rewriting theory. This allows us to easily state and verify important policy properties such as the absence of conflicts, for instance. Moreover, we develop rewrite-based methods to verify elaborate policy properties such as the safety problem in access control and the detection of information flows in mandatory policies.

We show that strategies are important to preserve policy properties under composition. The rich strategy languages available in systems like Tom, Stratego, Maude, ASF+SDF and ELAN allows us to define several kinds of policy combiners.

Finally, in this thesis we provide a systematic methodology to enforce rewrite-based policies on existing applications through aspect-oriented programming. Policies are weaved into the existing code, resulting in programs that implement a reference monitor for the given policy. Reuse is improved since policies and programs can be maintained independently from each other.

Keywords:
Security Policies, Policy Composition, Access Control, Strategic Rewriting, Term Rewriting.