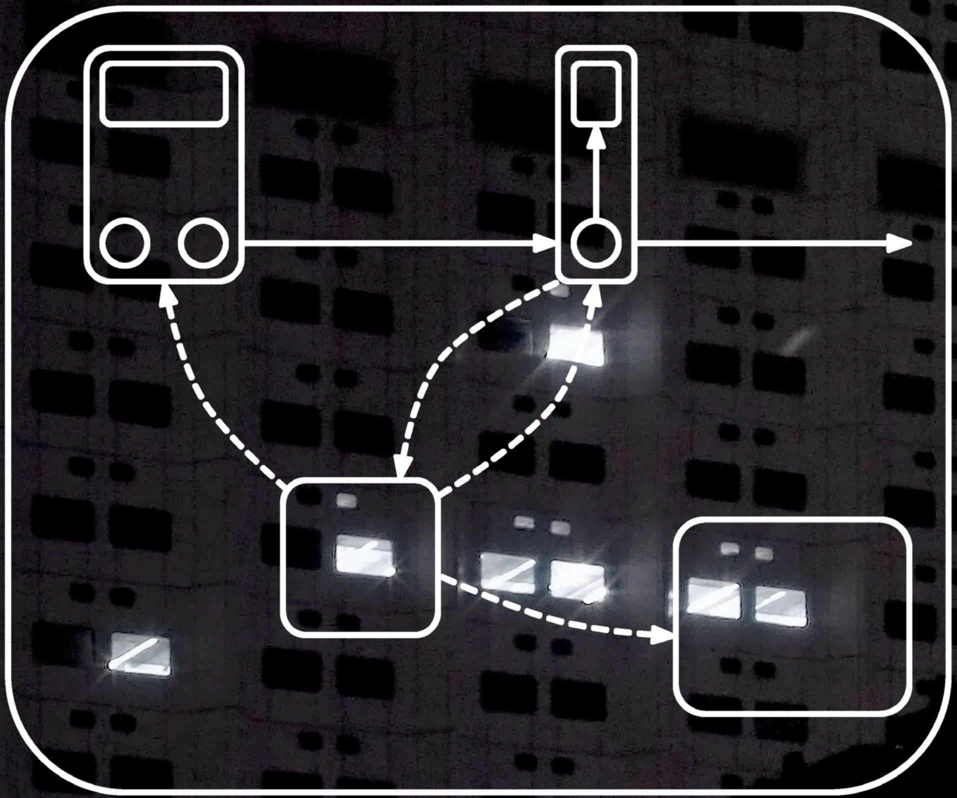
Flexible Access Control for Dynamic Collaborative Environments



Marnix Dekker

Flexible Access Control for Dynamic Collaborative Environments

M.A.C. Dekker

Access control is used in computer systems to control access to confidential data. In this thesis we develop flexible access control models for users in dynamic collaborative environments, such as hospitals, or consultancy firms.

We propose Audit-based Compliance Control (AC^2). In AC^2 user actions are not checked immediately (a-priori), like in conventional access control, but users must account for their actions at a later time (a-posteriori), by providing machine-checkable justification proofs to one or more auditors. This allows users to exchange and access confidential data, like health records in a hospital, in an ad hoc manner. A similar design choice was made recently for the Dutch electronic health record infrastructure (AORTA).

We also take a more conventional approach by proposing two extensions to Role-based Access Control (RBAC). These extensions give users more ways of authorizing and deploying RBAC policy changes, thus favoring dynamic collaboration between users.

Marnix Dekker graduated in Theoretical Physics from the University of Utrecht, and worked subsequently as a software developer in Pisa. He did his PhD research at TNO Information and Communication Technology in Delft, under supervision of Prof.dr. Sandro Etalle from the Distributed and Embedded Security research group of the University of Twente.

Marnix currently works for KPMG CT Information Technology specializing in computer security advisory, and in design and audit of critical computer systems.

An online version is available from
http://dx.doi.org/10.3990/1.9789036529501

Flexible Access Control for Dynamic Collaborative Environments

by M.A.C. Dekker

Graduation committee

Credits

The work in this thesis has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA) research school, within the context of the Centre for Telematics and Information Technology (CTIT). The author was funded by SenterNovem and TNO, through the IOP Gencom project Privacy in an Ambient World (PAW).

ISBN: 978-90-365-2950-1

CTIT PhD.-thesis series ISSN 1381-3617, Number 09-159.

IPA Dissertation series, Number 2009-26

DOI: http://dx.doi.org/10.3990/1.9789036529501

Typeset by LaTeX, edited with TeXShop.

Printed by Wöhrmann Print Service, Zutphen, The Netherlands.

Cover layout and photography by Denis Guzzo (http://denis.guzzo.name).

The cover shows the façade of one of the buildings of Erasmus MC, the university hospital of the city of Rotterdam. The overlay is a figure from this thesis (see page 15).

FLEXIBLE ACCESS CONTROL FOR DYNAMIC COLLABORATIVE ENVIRONMENTS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday, December 2nd, 2009 at 16:45

by

Mari Antonius Cornelis Dekker
born on the 1st of July 1976
in Rotterdam, The Netherlands

This dissertation is approved by:

Prof.dr. S. Etalle (promoter)
Prof.dr. P. H. Hartel (promoter)

Ter herinnering aan mijn opa Antonius Johannes Michael Kreischer.

Summary

Access control is used in computer systems to control access to confidential data. In this thesis we focus on access control for dynamic collaborative environments where multiple users and systems access and exchange data in an ad hoc manner. In such environments it is difficult to protect confidential data using conventional access control systems, because users act in unpredictable ways.

In this thesis we propose a new access control framework, called Audit-based Compliance Control (AC^2). In AC^2 user actions are not checked immediately (a-priori), like in conventional access control, but users must account for their actions at a later time (a-posteriori), by providing machine-checkable justification proofs to auditors. The logical proofs are based on policies received from other users, and other logged actions. AC^2 has a rich policy language based on first-order logics, and it features an automated audit procedure. AC^2 allows users to exchange and access confidential data in an ad hoc manner, and thus collaborate more easily. Applied in a medical setting, for example, doctors would be able to continue their work, regardless of authorization issues such as missing patient consent, and missing or outdated policies. Doctors can deal with these issues at a later time. Although this unconventional approach may seem, at first sight, inappropriate for practical applications, recently a similar design choice has been made for the Dutch national infrastructure for the exchange of electronic health records (AORTA).

At the same time we are aware of the fact that it is a big step for organizations to change from a conventional access control mechanism (a-priori) to a new mechanism. In this thesis we also take a more conventional approach by proposing two extensions to Role-based Access Control (RBAC) - an existing and widely used access control model. These extensions give users more ways of authorizing and deploying RBAC policy changes, thus favoring dynamic collaboration between users.

Samenvatting

Toegangscontrole wordt gebruikt in computersystemen om de toegang tot vertrouwelijke gegevens te bewaken. In dit proefschrift richten we ons op toegangscontrole in dynamische samenwerkomgevingen, waar meerdere gebruikers en systemen gegevens uitwisselen en inzien op een ad-hoc manier. In dergelijke omgevingen is het moeilijk om vertrouwelijk gegevens te beschermen met traditionele toegangscontrole systemen, omdat gebruikers op onvoorspelbare wijze handelen.

In dit proefschrift stellen wij een nieuw toegangscontrole raamwerk voor, genaamd Audit-based Compliance Control (AC^2). In AC^2 worden handelingen van gebruikers niet direct (a-priori) gecontroleerd, zoals in traditionele toegangscontrole, maar moeten gebruikers verantwoording afleggen voor hun handelingen op een later moment (a-posteriori), door het verstrekken van mechanisch-te-controleren bewijzen ter verklaring. Deze logische bewijzen zijn gebaseerd op beleid ontvangen van andere gebruikers, en andere gelogde handelingen. AC^2 heeft een rijke taal voor het uitdrukken van beleiden, die is gebaseerd op eerste-orde logica, en het heeft een geautomatiseerde audit procedure. AC^2 stelt gebruikers in staat om vertrouwelijke gegevens op een ad-hoc manier uit te wisselen en in te zien, en zo om makkelijker samen te werken. Toegepast in een medische omgeving, bijvoorbeeld, zou het zo mogelijk zijn voor doktoren om door te gaan met hun werk, ongeacht autorisatieproblemen, zoals het ontbreken van toestemming van de pati ent, en ontbrekend of oud beleid. Doktoren kunnen dergelijke problemen later oplossen. Hoewel deze benadering op het eerste gezicht niet geschikt lijkt voor praktische toepassingen, onlangs is een vergelijkbare ontwerpkeuze gemaakt voor de Nederlandse nationale infrastructuur voor het uitwisselen van digitale medische dossiers (AORTA).

We beseffen tegelijkertijd dat het een grote stap is voor organisaties om te veranderen van een traditioneel toegangscontrolemechaniek (a-priori) naar een nieuwe (a-posteriori). In dit proefschrift nemen we ook een meer traditionele benadering door twee uitbreidingen van rolgebaseerde toegangscontrole (RBAC) - een bestaand en wijd gebruikt toegangscontrole model - voor te stellen. Deze uitbreidingen geven gebruikers meer mogelijkheden om beleidswijzigingen te autoriseren en uit te rollen, en daarmee vergemakkelingen ze dynamische samenwerking tussen gebruikers.

Acknowledgements

I am grateful to my supervisor Sandro Etalle, professor at the Distributed and Embedded Security (DIES) research group of the University of Twente, and professor at the Security of Embedded Systems (SEC) group of the Technical University of Eindhoven. Sandro always pulled me through the paces of doing research and publishing about it, switching swiftly from one fluent language to the other (Dutch for bad news, English for good news, Italian for politics). This thesis would not have been possible without his time and effort.

I thank my promoter Pieter Hartel, for making the DIES research group a fruitful place full of talented researchers. His critical view and his warm words (by Skype and email) in the last year were essential.

I thank Jason Crampton, Bart Jacobs, Wim Jonker, Fabio Massacci, Thijs Veugen, and Roel Wieringa for assessing my thesis and providing valuable feedback as members of my graduation committee. It is an honor to have them in my graduation committee.

I was lucky to work alongside some great researchers and friends: Ricardo Corin for being such an enviable researcher, Gabriele Lenzini for his visions into the future, Jerry den Hartog for showing me the formal aspects of computer science, Jan Cederquist for enlightening me on the different types of logical implementations, Jason Crampton for his experience in the field of RBAC, and Thijs Veugen for showing me how to write a technical paper in Dutch.

I would like to give credits to the partners of the Privacy in an Ambient World (PAW) project and in particular to TNO, and Jan Huizenga (my former manager at TNO ICT) for giving me a PhD researcher position. I thank Senternovem for funding and steering the project, as well as the members of the board of supervision who have provided valuable feedback during the PAW progress meetings.

I would like to thank Muhammed Dashti, Cas Cremers, Hugo Jonker, Ayse Morali, and Anka Zych and the other members of the Security PhD Association Netherlands (SPAN), for bringing much needed fun and feedback at conferences and meetings.

I thank my current employers Peter van Doorn, and Joost Koedijk (partner and associate partner at KPMG CT Information technology) for giving me a mighty interesting job two years ago, and the time to finish this thesis. I also thank Wim Kegel (architect at Logius) for giving me the chance to design parts of DigiD and allowing me to put my research in a different perspective.

Going back almost 8 years, I would like to thank Emanuele Pardini (owner of Genesy Srl, Pisa) for hiring me as a programmer and giving me the time to make the first steps in modeling and programming. Not to mention Stefania Simonini for putting in a good word for me, and Ermanno Maci for supervising me the first year.

I thank my family and friends for all the help they have given me: My dear friend George Papaefthymiou for all the advise, and long nights of fun. My mother-in-law Enza, for entertaining me daily with Italian discussions and food. My mother and father for everything, from helping me to find this job posting, get a house in Delft, for babysitting when I was in Enschede, and much much more. My sister for telling me so often how proud she is of me. My grandmother for her repeated encouragements to graduate. Bedankt oma, voor al je vragen, en aansporing! Every few months she refreshed me with her sincere curiosity about my research and that has reminded me that privacy of personal data is a concern of people of all ages.

Finally my wife Michela Pelusio for advising me to get a research job in Holland, for having patience with me and the thesis, and for making my life a sweet and exciting adventure.

Delft, November 2009.

Contents

Chapter 1

Introduction

The digital world is changing rapidly. Distributed systems with multiple systems and users communicating through a network, have become pervasive. For example, citizens use computer systems for social life, professional work, political activities, and transactions with the government. In consultancy firms, and research institutes, email and online services are used for better collaboration and research. In hospitals, electronic health record systems are used to collect and access quickly the needed health information about its patients. In many different settings, the computer systems are exchanging *confidential* data. Think of credit card numbers, mail addresses, social security numbers, or health records. When designing and implementing such computer systems, access control plays an important role: Confidential data must be protected from unwanted access, and at the same time access by the right users and systems is vital. In this thesis we focus on designing flexible access control for the protection of confidential data in dynamic collaborative environments.

1.1 Conventional Access Control

Generally speaking, the goal of an access control system is to prevent people, or computers, from performing unwanted actions [56]. Access control systems can be standalone computers, such as network firewall hardware guarding access to the network, or be part of a larger system, for example inside a database system guarding access to the tables. Let us briefly describe how an access control system works in general, before discussing the different types of access control systems.

At step 1 (see figure 1) the user makes a request, for example, a remote procedure call, or an operating system call. The request is received (or intercepted) by the access control system's *reference monitor*, which decides whether the request should be granted or not. At step 2, the reference

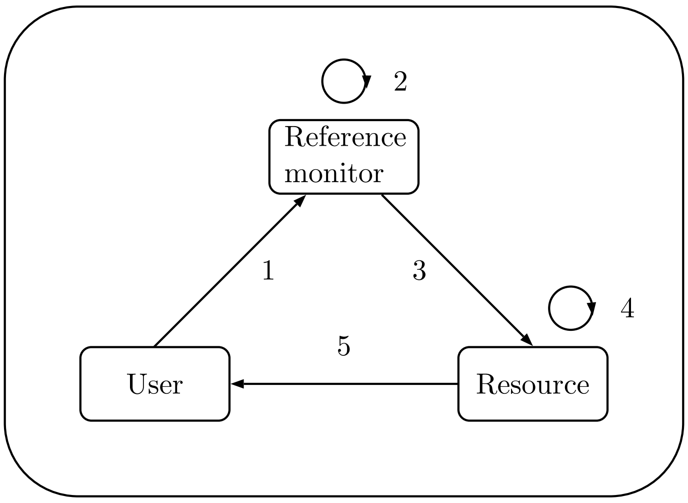


Figure 1: A sketch of a user interacting with an access control system

monitor consults an *access control policy*, to make the decision. This decision is called an *access control decision.* At step 3, if the decision is positive, the reference monitor communicates its decision to the resource, and in step 4 the reference monitor forwards the request to the resource. Finally, the request is processed by the resource, and depending on the setting, data or a confirmation are sent back to the user.

An access control policy can be based on a single configuration file, such as a firewall rule list, or on data scattered across systems, such as the file permissions in a Unix filesystem. It can contain *permissions*, such as 'Bob is allowed to read document X', *prohibitions* such as 'Alice is not allowed to modify document X', or more general statements, such as 'All users with clearance B, can read documents of classifications B and C'. Access control policies may concern properties of users, such as their name, or clearance, properties of objects, such as the status of documents, environmental conditions such as time, past user actions such as payments, or future actions such as fair usage. For example, a digital license to play a copyrighted piece of music three times, is a type of access control policy.

An access control model is an abstract description of how an access control system works in practice. Prominent models are:

- *Mandatory access control*: in mandatory access control users with a low clearance can not read documents with high classification, and users with a high clearance can not write documents with a low classification. Mandatory access control is applied for instance in military information systems.

- *Role-based access control*: in role-based access control only users in a certain role can perform certain actions. Role-based access control is applied for instance in database systems such as Oracle.

- *Discretionary access control*: in discretionary access control the user who has created the data has all the permissions about it, and the permission to delegate his permissions to others (for example ownership, or read access). Discretionary access control is applied for instance in multi-user systems, such as *Linux*.
- *Digital rights management*: in digital rights management users do not create any data, and user permissions may depend on factors like past payments, the type of device being used, or the country the user is in. Digital rights management is used in DVD players.
- *Attribute-based access control*: in attribute-based access control, only users with certain attributes can perform certain actions. Attribute-based access control is used in the XACML standard.

In the different access control models mentioned here, different types of policies are used to make access control decisions. In mandatory access control, role-based access control, digital rights management and attribute-based access control, the system administrator determines upfront who can access which data. These policies are usually referred to as mandatory policies. In discretionary access control, on the other hand, the user can change the policies about data she created, or data she has received ownership of.

1.2 Dynamic Collaborative Environments

Dynamic collaborative environments can be found in hospitals, consultancy firms, and research institutes. A dynamic collaborative environment (DCE) is typically composed of different computers, often in different locations, and used by a group of peer users who exchange data in an ad-hoc, and sometimes unpredictable, way. DCEs are becoming more common, as more social, civil and professional activities take place using computers and networks. Protecting confidential data is difficult in DCEs because 1) It is not possible to appoint a central authority which deals with data protection, 2) it is impossible to foresee how users will collaborate, and 3) there is no time to go through lengthy procedures to deal with data protection. The left side of figure 2 shows how data is typically exchanged in an e-commerce setting, where one computer interacts with multiple customers, in a centralized and predictable way. The right side of figure 2 shows a dynamic collaborative environment in a research institute, in which data is exchanged between peers in an ad-hoc and decentralized way.

In a DCE it is important to strike a balance between protecting private data from unwanted access (confidentiality), and guaranteeing timely access

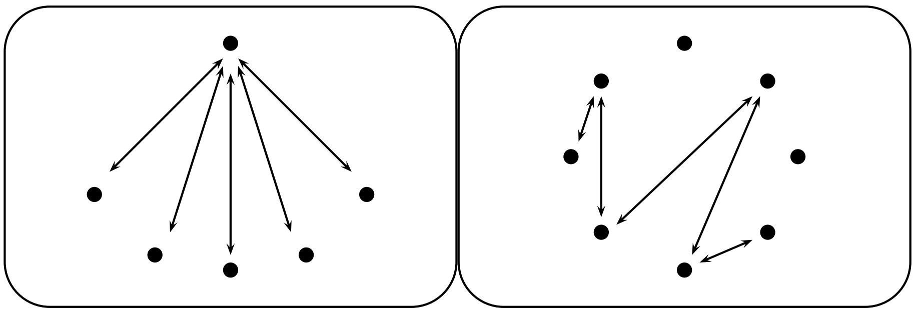


Figure 2: Data exchange in an e-commerce system (left side) and in a dynamic collaborative environment in a research institute (right side)

to it when needed (availability). Let us take an electronic health record system for example:

- Unwanted access to health records may cause lifelong discrimination of patients by employers, and insurance companies.
- If, on the other hand, hospital staff do *not* have access to the necessary health records when needed they may make wrong diagnoses, or they may have to perform extra medical exams. Costs of health care would rise and the quality of the health care would drop. A report from 2003 showed that an estimate 170.000 patients were affected by incomplete or inaccurate medical information, costing around 1.4 billion [35], every year.

It is difficult to find the right balance between confidentiality and availability in DCEs, because the exchange pattern is complex and unpredictable. Consider for example again an electronic health record system:

- Health records are created, exchanged, accessed and modified by medical personnel across hospital shifts, across different hospitals, sometimes even across borders.
- Health records can be used for a variety of purposes, such as for medical operations, for billing patients or their health insurances, for hospital audits, or even research purposes.
- Medical data, whether it is output by an MRI scanner, or read from a national health record database, is subject to regulations and restrictions on how and by whom it can be accessed and processed. Different restrictions apply for different patients, different doctors, depending on the details of the medical setting.

In health care, access and usage policies are particularly complex as they result from a combination of different requirements: Suppose doctor Alice asks nurse Bob to read a certain health record (for example, to monitor the temperature of the patient) and to add certain data. The decision on whether or not Bob may access it could depend on hospital regulations, or on patient consent, but also on whether or not Alice is a certified physician, whether she and Bob are officially treating the patient in question, or whether it is an emergency or not. Taking all the relevant aspects into account in the right way is difficult.

Let us give another example of a dynamic collaborative environment, this time in the profit-sector: a consultancy firm. Hundreds of consultants who work in different departments of the firm on projects for clients. In consultancy firms most documents are confidential (but not top-secret): For example, they may contain confidential sales figures from clients, or confidential research results from the firm itself, that may only be shown to employees. Again different documents are subject to different rules. For example, certain documents may only be shown to employees of a certain department of the firm, other documents may be shown to all the employees, and even to certain clients. Protecting confidential documents from unwanted access is important, but, on the other hand, if consultants do not get information they need in time, then the quality of the work drops, and the costs for the clients rise. In many firms consultants collaborate also across departments in a dynamic way. Suppose Alice, who works in the financial department is working on a report for a client, needs help from Bob on some text about technology. The decision on whether or not Bob, who works in the technology department, can access Alice's report depends on the content of the report, the firm's and financial department's policy, on prior agreements with the client, prior or current projects that Bob works on for other clients, Alice's role in the project, et cetera.

1.3 Research question

We have given an overview of conventional access control, and the characteristics of dynamic collaborative environments. We argue that dynamic collaborative environments have characteristics that complicate the deployment of a conventional access control system. Let us illustrate this by taking a simple workflow in a dynamic collaborative environment.

Alice, Bob and Charlie are peers, with different expertise, working in different departments. They collaborate when a project requires all their expertise. Alice creates a new document, and she sends a message to Bob asking him for his help. Bob reads the document and adds some extra information to it. Bob gives the document to Charlie who stores it for reviewing it later on. The document could be a health record, and Alice, Bob, and Charlie could be medical staff. Or Alice, Bob and Charlie could be employees of a consultancy firm, and the document a summary of sales figures of a client.

Conventional access control is not well suited for the workflow just described: Mandatory or role-based access control only allows Alice, Bob and Charlie to exchange data along a pre-defined role or clearance hierarchy. But in this case, on the other hand, Alice, Bob and Charlie want to exchange data across the organization's hierarchy. Digital rights management and attribute-based access control do not depend on a pre-defined hierarchy, but they do not allow Alice to change the policy and disclose the document to Bob. Only system administrators have the privilege to change policies. Discretionary access control allows users to change policies. If Alice owns the document then she can give Bob write access to the document. But in discretionary access control Alice cannot give the right to Bob to give read access to Charlie (at least, not without giving full ownership to Bob). Moreover it is well known that discretionary access control is not well-suited for settings in enterprises, because in enterprises users rarely 'own' the documents they work on [33].

This leads to the main research question of this thesis:

How can we design a flexible access control system that is suitable for dynamic collaborative environments?

There are two possible approaches to this: One could take an existing access control model, and extend it with the needed features, or one could design an entirely new access control model. We will do both in this thesis, and compare the results in the final chapter.

1.4 Contributions

In this thesis we try to give answers to the research question in two ways: 1) We propose a new access control model and tools for implementing it specifically tailored to dynamic collaborative environments. 2) We extend RBAC to make it more suitable for dynamic collaborative environments.

- In Chapter 2 we introduce a new framework for controlling compliance to discretionary access control policies: AC^2. The AC^2 framework uses a simple policy language (based on first-order logic), that models ownership of data, permissions, obligations, and also (nested) delegation (by the *maySay* predicate). Users can create documents, and authorize others to process the documents. AC^2 uses a formal audit procedure, to control compliance to the policies. Users may be audited and asked to demonstrate that an action was in compliance with a policy. Justification proofs are implemented by a formal proof system (a sequent calculus). We illustrate how the AC^2 framework can be used in a consultancy firm where a group of consultants produce and process confidential documents in a decentralized way. This framework was published in the International Journal of Information Security (IJIS) [2], as joint work with J. G. Cederquist, R. Corin, S. Etalle, J. I. den Hartog and G. Lenzini, and based on early versions of the AC^2 framework published in conference proceedings [1, 25]. See also the acknowledgements of Chapter 2 in Section 2.8.

- We have developed an automated proof checker for the AC^2 proof system. Proof checking is a central part of the AC^2 audit procedure. We give a description of the proof checker in Chapter 3. In the same chapter we also derive an important logical result (a cut-elimination theorem) about the AC^2 proof system, which shows that the logic is well-behaved (consistent), and that there exists a semi-decidable proof finding procedure. We show, as a proof of concept, an automated (justification) proof finder by using Prolog. Parts of Chapter 3 (the details of the cut-elimination proof and a brief description of the proof checker and the proof finder) were published in the International Journal of Information Security (IJIS) [2]. An early version of the proof checker was presented in the proceedings of the 2005 IEEE POLICY workshop [1].

- In Chapter 4 we show how AC^2 can be used in a Electronic Health Record (EHR) system in a hospital. We show that AC^2 fulfills the requirements of legislation on health care, while at the same time providing easy access to health records. Chapter 4 was published in the 2006 VODCA workshop proceedings [7], as joint work with S. Etalle. A short version in Dutch, which is joint work with P. J. M. Veugen, appeared in 2007 in a Dutch magazine for Infomation Security professionals [8].

- In Chapter 5 we show how AC^2 can be used in an Enterprise privacy system. Enterprise privacy systems are used to enforce privacy policies of customers across an enterprise. We compare AC^2 with two (well-known) privacy systems (E-P3P, and P3P) used in this setting, and we argue that AC^2 provides better privacy guarantees. This chapter was published as a bookchapter in Security Privacy and Trust in Modern Data Management, and is joint work with S. Etalle, and J. I. den Hartog.

In the second part of this thesis we take a less revolutionary (and more evolutionary) approach to answering our research question. We extend ANSI RBAC, a widely used standard for role-based access control.

- In Chapter 6 we propose a new administrative model for RBAC, which is at least as safe, *and* more flexible than existing models. We also show that our model can be implemented in an RBAC reference monitor. A short version of our work was presented in the proceedings of the 2007 ACM ASIACCS symposium [3], as joint work with J. Cederquist, J. Crampton and S. Etalle, while an extended version with full proofs and examples was published in the proceedings of the 2007 Secure Data Management workshop [5].

- In Chapter 7 we extend RBAC with a model and a basic procedure for administration in distributed systems. Despite distributed systems becoming more and more common, there is hardly any literature on this aspect of implementing RBAC. This model, which is joint work with J. Crampton and S. Etalle, was published in the proceedings of the 2008 ACM SACMAT symposium [4].

1.5 Conclusions

In this thesis we address the research question of Section 1.3, by proposing a new access control model (AC^2) and by extending an existing on (RBAC).

AC^2 starts from the basic assumption that users *can* behave badly. AC^2 features machine-readable a-posteriori justification proofs through which users can be held accountable for their behavior, whether appropriate or not. We show the flexibility of our access control model, and how it can be implemented in practice. By proposing extensions to RBAC we take a more conventional approach. We propose a general class of administrative policies, and efficient administrative procedures for distributed systems. We show the flexibility of our model, and how it can be implemented in practice.

Although AC^2 may represent a big change from conventional access control, there are several settings where our ideas may be applied in the near future. The design of the future Dutch health record infrastructure (AORTA) is based on the idea that all doctors may access health records, but that they must be able to account for their use of medical data [46]. In AORTA fine-grained a-priori access control is *replaced* by *a-posteriori* auditing of logs of access to health records. In a different setting, Koot has argued that AC^2 has advantage over RBAC in the Service Oriented Architecture of a Dutch insurance company [55]. We believe that there are many more settings where our ideas may be put into practice in the near future.

Part I

Audit-based Compliance Control Framework

Chapter 2

Audit-based Compliance Control

2.1 Introduction

The problem of enforcing data protection policies, i.e. guaranteeing that data is used according to predefined policies and rules, is present in all situations where IT systems are used to process confidential data. While this is a universal problem, in different settings this influences the architecture of an IT system differently. In general, the higher the degree of assurance required, the more inflexible is the system enforcing it. For instance, in military settings, where secrecy needs to be guaranteed at all costs, users are willing to use a rigid access control system to enforce (mandatory) data protection policies. In health care settings [85] more flexible systems are needed which guarantee privacy of patients without interfering too much with the availability of data, by allowing users to override mandatory policy [64, 74]. At the other end of the scale one finds *dynamic collaborative environments* where even more flexibility is demanded, and, as a consequence, discretionary access control systems are prevalently deployed. Consider the following example set in a dynamic collaborative environment:

Example 1 *Alice creates a document, containing some public market analysis. She sends Bob the document and the policy: This may be seen and modified only by employees. Bob, subsequently, adds extra information to the document, making it more confidential, and sends it to Alice and Charlie with the (more restrictive) policy: This may be seen and modified only by seniors. Now Charlie, a senior, needs someone to fix typos quickly and the only one around is not a senior: It's Dave a junior. He wants to send the document to Dave, and allow Dave to get the work done and he is sure Bob would agree, given the urgency, but Bob is not in the office to authorize Dave. Charlie would like to change the policy himself (and authorize Dave),* *while taking the responsibility for the policy change.*

This example, though simple, highlights the essential features of dynamic collaborative environments. First, there is no central authority that issues and enforces policies. Second, it is difficult to determine *which* is the policy that applies to a given document: when Alice creates d_1 and gives Bob the policy ϕ, say to read it, Bob has no way of checking that ϕ is the 'right' policy for d_1. For instance, Alice could have sent a confidential document for which she could not authorize Bob. Bob can only trust Alice's word on it. Third, in a dynamic collaborative environment, users are administrators themselves, and it becomes important to be able to express *administrative* policies, stating i.e., who may authorize other users. Fourth, dynamic collaborative environments often present rapid changes. There is not always time to align all applicable policies first. Infringement by users should be possible in some way, to avoid blocking the work of the users. Going back to our example, Alice, Bob and Charlie would otherwise bypass the access control system (for example by exchanging passwords, or by exchanging documents outside of the access control system).

Standard techniques for protecting documents include *Access Control* [44] and *Digital Rights Management* [89]. In access control and digital rights management systems documents are stored or processed in some controlled environment (e.g., a database or a special device). A general problem of mandatory access control and that of DRM is that only a few central users can issue policies, and that users do not own documents they create, if they can create documents at all. A more flexible approach is discretionary access control, where users *can* create documents and subsequently issue policies about these documents, and authorize other users. Discretionary access control (e.g. present in *Windows* and *Unix* filesystems) is used pervasively in dynamic collaborative environments. However, there is a well-known problem with discretionary access control: a user can always create a document and copy a confidential document into it, and claim it as his. To address this problem, Trust Management (TM) systems have been developed [21], where it is the user who is supposed to infer whether the issuer of the authorization can be trusted. For example by inferring the reputation of a user, or the credentials. Checking whether a license is issued by the right authority is often feasible in DRM. Everybody knows licenses for *Purple Rain* are issued by *Sony*, everybody knows *Windows XP* licenses are issued by *Microsoft*). However, in a dynamic collaborative environment judging the genuineness of an authorization for a document is harder because of the variety of possible sources and the complexity of the environment. All the users create, send, modify documents, and ask others to review or change. At the same time, legislation increasingly demands compliance to policies, and accountability with regard to the disclosures of confidential documents [85, 84, 86].

In an attempt to solve this problem, we take a different approach, which we call *audit-based compliance control.* The most eye-catching element of our framework is the fact that policies are not enforced *a-priori*, but checked *a-posteriori.* We will show that this gives users more flexibility, and that it in certain settings it can be used to control compliance of users to policies.

We should stress here that our framework can not replace all a-priori access control systems in an organization, rather it is a way of controlling compliance of users in a closed setting, such as a hospital or a consultancy company. It must be feasible to hold users accountable, before they leave the system. Ordinary *a-priori* access control is still needed to prevent outsiders from entering the closed setting.

Basically, we assume the presence of an *auditing authority* with the task and the ability to observe the critical actions of the users. This requires that users are somehow operating within a well-defined environment. Assuming the presence of such an environment is not unreasonable:

- Employees in companies are often operating from especially prepared computers, where logging systems are present, and they often access central systems such as databases that log transactions as well.

- Logs are often kept already not only for detecting flaws, but also to comply with legislation on accountability and auditability [85, 84].

- Discretionary access control, which is widely used and deployed, also assumes that user actions are audited [76].

We assume also that that the user can keep a secure log of certain actions and or certain circumstances, to prove the necessary facts to the auditors. This a reasonable assumption as well. Depending on the setting, they could be for example cryptographically signed return receipts, that a certain payment was made, or request, or response messages from webservices in a service oriented architecture.

While the fact that compliance checking is done *a-posteriori* is superficially the most striking element of our framework, there are two other ingredients which we would like to mention here.

1. We propose a simple policy language, based on first-order predicate logic. Its operational semantics is defined by a formal proof system, which is an extension of the first-order logic proof system and specifically tailored for discretionary access control policies. First-order logic is more expressive than for example Datalog, which has been used in numerous existing access control frameworks (see the section on Related work). Our proof system allows users to express and refine *delegation* of rights, and to refer to conditions and (pre- or post-) obligations in policies they give to other users. This is not possible in a number

of legacy access control systems, such as RBAC, or XACML. Importantly, despite the expressiveness of the language we demonstrate that the proof system is semi-decidable by proving a cut-elimination theorem. Consistency of the proof system also follows from cut-elimination. We are the first to use a cut-elimination theorem for an access control logic.

2. Another important feature of our system is that users, instead of having to check whether a received policy is the *right* policy for a given piece of data, they simply assume the policy to hold. This is different from what is usually done in, Trust Management [21] or other distributed access control frameworks [9], where the receiver of a policy must make some kind of trust calculation. Referring to Example 1: In AC^2, if Alice is not an authority on the document nor the real creator of the document, then the auditor will not blame Bob, but instead put the blame on Alice.

In this chapter we give a brief overview of our system (Section 2.2). We describe the overall framework in Section 2.3, introducing the policy language syntax, the logging mechanism and the audit procedure. In Section 2.4 we define the semantics of our language by defining a formal proof system, while in Section 2.5, we show, by giving an example, how the framework can be used in a common dynamic collaborative environment: A consultancy firm where protection of confidential documents is needed. Chapter 3, contains the technical details about the proof system. There we show that the cut-elimination theorem holds, which is an important technical result that implies consistency and semi-decidability, and we show prototypes of both the proof finder and the proof checker.

2.2 Overview

In our framework compliance of users to policies is checked a-posteriori. This approach yields a more flexible system for the users, but requires that users take responsibility for their actions. The two main assumptions for this approach are the following.

1. Auditors can observe critical actions. Hence there must be a sufficiently comprehensive *audit trail*, which can not be forged or bypassed, containing the relevant details about the actions and the identity of the users executing them.

2. All the users of the system can be held accountable for their actions. Hence it is required that users only vanish after having accounted for past actions.

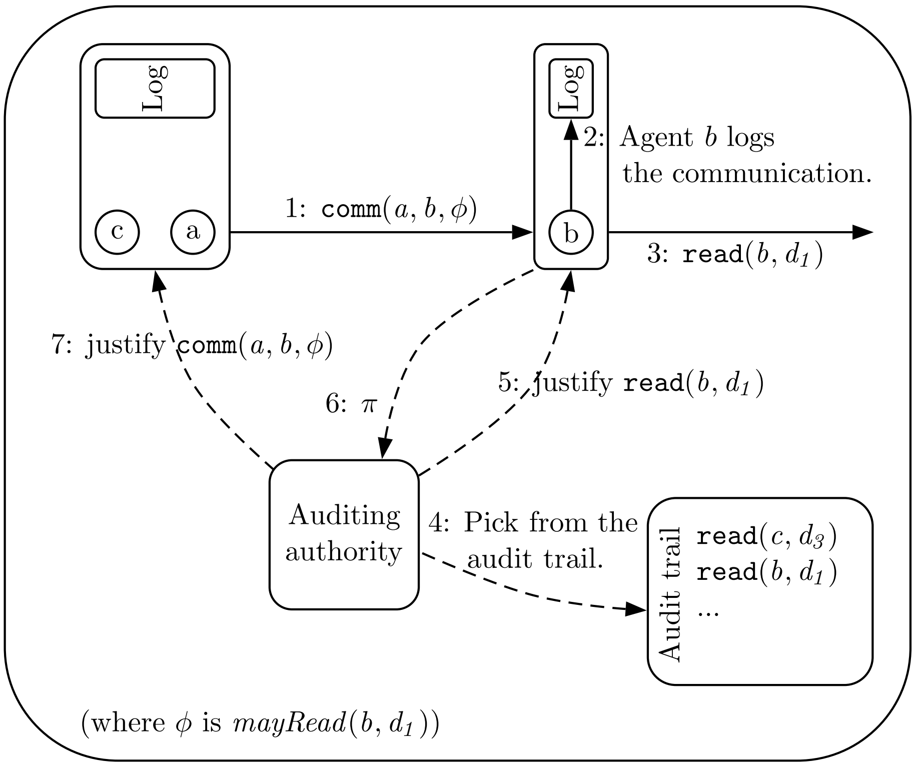


Figure 3: Sample deployment depicting actions, the logging and interaction with an auditing authority.

Although we agree that in some settings these assumptions are not realistic (for example in the setting of an online video store with thousands of customers across different continents), this *does* apply to organizations such as companies, or hospitals. This will be discussed further in section Section 2.5 and in the conclusions of this chapter.

Intuitively, the framework works as in the following example: Bob receives from Alice the authorization ϕ to read a document d_1. Bob reads the document d_1. As mentioned in the introduction, Bob does not check whether or not Alice is one of the authorities that can issue policies about d_1, or just entitled to say ϕ. Bob simply proceeds to read the document d_1, and relies on the auditor to check the actions of Alice.

Figure 3 shows a sample run in the framework: In the first step (1), user a provides a policy ϕ to user b which b records in its log (2). Next (3) user b reads document d_1. We don't make assumptions on how 'reading' is implemented (e.g. whether document d_1 is stored centrally, or sent across by email), neither about how the logs of Alice and Bob are implemented (for example on a shared server, or at separate workstations). In fact in the figure we have depicted another agent c (Charlie) that shares a log with

Alice on a multi-user system.

At a later point the *auditing authority*, who guards access to sensitive files, finds the access of b (4) and requests b to justify this access (5). User b responds to the audit, and replies with a justification proof π, which shows that the access was allowed according to the policy ϕ, communicated by a. The auditor, though initially unaware of a's involvement, can now (7) audit a for having communicated the policy ϕ to b.

In the figure both users a and b are asked to provide a justification proof, but we have not made assumptions about when they generated the justification proofs. In some scenario's users may decide to go ahead and wait with finding the proof, for example because the right authorizations still need to be issued (for example emergency treatment with only informal patient consent). In other scenario's users may want to check beforehand whether a justification proof exists (see for example the section 2.3.5 Honest strategy), and generate the proof immediately. A user-friendly solution would be to supply users with a kind of reference monitor that checks if a justification proof can be found quickly, then allows the user to continue without a justification proof, or cancel.

For reasons of privacy, it is left to the individual users to access their logs and use the right parts to justify their actions. The auditor only checks the justification proof, and the parts of the log that are needed to support the proof, while the parts of the log of the users not needed in the proofs can remain confidential. In settings where the auditor is trusted, proofs may even be generated by the auditor, possibly by using facts about the users, or general policy.

2.3 Framework

In this section the basic definitions of the AC^2 framework are introduced. The section is organized as follows: We discuss the policy language used in the audit framework and we describe the logging mechanism, which is used by the agents to provide evidence for the justification proofs. In the end we give the formal definition of auditing and accountability.

2.3.1 Policy language

In our framework we use a simple policy language, which is in some respects similar to the languages used in Binder [30] and PCA [12]. We will return to the main differences in the related work section of this chapter (Section 2.6).

Basic permissions for actions are expressed using *atomic predicates.* The objects of these predicates are agents and data. Agents are users, or programs or devices operating on behalf of users. We have a set $\mathbf{AG} = \{a, b, c, \ldots\}$ of *agents* and a set $\mathbf{DA} = \{d_1, d_2, d_3 \ldots\}$ of *data.* For example the predicate $\mathit{mayRead}(a, d_1)$ expresses that agent a has permission to read data d_1. Additionally, atomic predicates are used to express basic conditions or facts, e.g. $\mathit{isEmployee}(a)$ expresses the fact that agent a is an employee.

Actions are represented by a set $\mathbf{AC}$, containing

- $\mathtt{create}(a, d_1)$, expressing a has created data d_1,
- $\mathtt{comm}(a, b, \phi)$, expressing a communication of a policy ϕ from agent a to b,
- scenario-specific actions like $\mathtt{read}(a, d_1)$, $\mathtt{write}(a, d_1)$, etc.

In our model we make a distinction between actions and *instances* of actions. Different instances of an action are distinguished using a unique identifier $i \in \mathbb{N}$, as in $\mathtt{create_i}(a, d_1)$. Formally this gives a set $\mathbf{AC}^* \subset \mathbb{N} \rightarrow \mathbf{AC}$ of action instances.

The grammar for the policy language is based on the grammar for first-order predicate logic. It has been shown that first order predicate logic is sufficiently expressive to model a wide range of access control policies [40].

Definition 1 (Policy grammar) *Let s_i be agents or data and act an action, the set* $\mathbf{PO}$ *of* policies*, ranged over by ϕ is defined by the following grammar:*

$$\begin{aligned}\phi ::= &\ p(s_1, ..., s_n) \mid \top \mid \mathit{maySay}(a, b, \phi) \mid \mathit{owns}(a, d_1)\\ &\mid \phi \wedge \phi \mid \forall x.\phi \mid \phi \rightarrow \phi \mid \xi \rightarrow \phi \mid \mathtt{act} \xrightarrow{!} \phi \mid \mathtt{act} \xrightarrow{?} \phi\end{aligned}$$

where ξ are called obligations, and $\mathtt{act} \in \mathbf{AC}$ *are actions.*

Atomic predicates in the grammar are either $\top$, which is the trivial policy (true) that can always be derived, or scenario-specific predicates, denoted by $p(s_1, ..., s_n)$, where $s_1, ..., s_n$ are agent or data variables. For example, depending on the scenario, $\mathit{mayRead}(a, d)$, and $\mathit{mayWrite}(a, d)$.

Please note that, unlike in ordinary logics, we do not include an atomic predicate for falsity $\bot$, and hence negation $\neg$ can not be expressed. Falsity would be a policy that allows a user, who can derive it, to do anything, and since we do not see a practical use for such a policy we omit it here.

The $\mathit{maySay}()$ construct is used to express the right to delegate rights, and the policy $\mathit{maySay}(a, b, \phi)$ means that a is authorized to say ϕ to b. This type of policy is also known as administrative policy (about ϕ). We are not aware of existing access control logics that use this type of construct. This is due to the fact that in existing proposals, instead of modeling who may say a statement, the receiver of a statement must decide whether or not to trust it (see Section 2.6 for more details).

Central in our framework is the notion of *refinement* of administrative policies: Refinement is defined as follows. If an agent is authorized to say a certain policy, then it is also (implicitly) authorized to say a weaker (refined) policy. This allows for a flexible delegation of policies, allowing a user to say a more restricted policy, for instance by adding more conditions.

The predicate *owns*() has the usual meaning, stemming from discretionary access control models [44]: If an agent is the owner of a piece of data then it can derive policy formulae *about* that piece of data, and communicate any policy about the data to other agents. Owners can make other users owner too.

The conjunction $\wedge$ and the universal quantification $\forall$ have their usual meaning. The operators for disjunction and existential quantification are not included in the grammar. This is done for the sake of simplicity. Implication $\rightarrow$ has the usual meaning, and $\phi \rightarrow \psi$, states that a proof of ϕ is needed to obtain the permission ψ. The connectives $\overset{!}{\hookrightarrow}$ and $\overset{?}{\hookrightarrow}$ are used to express use-once and use-many obligations in policies. When a user fulfills a use-many obligation `act` of a policy `act`$\overset{?}{\hookrightarrow} \phi$, then the policy applies to any number of actions allowed by the policy ϕ. Fulfilling a use-once obligation `act` of a policy `act`$\overset{?}{\hookrightarrow} \phi$, however, can only be used once for a single action. The logging mechanism, reported below, and a type of linear logic, to be defined in Section 2.4, are used to implement the *use-once* obligations. We give a brief example: Suppose a user a receives a policy $\mathit{pay}(a, \mathit{pound}) \overset{!}{\hookrightarrow}$ $\mathit{mayViewVideo}(a, d_1)$ then this means that a is allowed to view the video once, for each time he logs a payment of a pound.

Remark 1 (Logics in access control) *Most access control systems can be modeled using logics [9]. From that point of view authorizations are (security) predicates, and the access control decision that grants access corresponds to proving that the predicate, that allows the access, holds. Even though the formal semantics of such logics is not always straightforward [9] - the same can be said for intuitionistic predicate logic - logical derivations are well understood, and logics are useful to analyze properties such as decidability and consistency.*

Remark 2 (Decidability of the language) *The decidability of policy languages is an important issue for the practicality of an access control system [9, 61, 40]. Most systems use decidable logics [61, 40, 14, 30, 59, 39, 18, 21]. For a decidable logic there are procedures that decide whether a statement is true or false. For a semi-decidable logic there are only procedures that decide for the true ones, while they may remain undecided about the false ones.*

Expressive policy languages are often semi-decidable or undecidable [12, 18]. Our framework uses an extension of first-order predicate logic, which is (only) semi-decidable. This type of undecidability is not a problem in our setting, because the agents and not the auditing authority are expected to find proofs.

Let us illustrate this difference by a brief example. Suppose R is an access control reference monitor, that uses an semi-decidable policy language. In this case, each time users request access that is not allowed, there is the risk that R cannot decide. Some mechanism would be required that makes R give up searching, and continue with the requests of the other users. On the other hand, suppose instead A is an auditor who requires users to find a justification proof themselves. In this case, if one user tries to find a proof for access that is not allowed, then only this user looses time, while A can continue to audit other users.

Remark 3 (Concerning obligations) *Obligations have been used in other access control systems with a different meaning [66, 53]. In these proposals obligations are call-back functions that have to be executed by the* access control mechanism*, before access can be granted. In our approach obligations are actions that have to be performed by the* user*. Our approach is similar to the approach taken in the UCON framework [68]. Post-obligations, obligations to be fulfilled later on, are hard to implement when using a-priori access control, because a separate audit mechanism would be needed to check if promises have expired or if they were fulfilled. In our framework, because an audit mechanism is already used, post-obligations are straightforward to implement.*

2.3.2 Proof obligation and conclusion

In our framework, the *proof obligation* function and the *conclusion derivation* functions link policies and actions. They are *public* functions which are known to all users. This ensures that all the users are aware of the meaning of the basic permissions. A straightforward way to implement this would be to use a central trusted authority that provides them to all users.

- The *proof obligation* function describes which policy an agent needs to satisfy in order to justify the execution of an action.

$$\mathbf{pro} : (\mathbf{AC} \times \mathbf{AG}) \rightarrow \mathbf{PO}$$

- The *conclusion derivation* function, describes what policy an agent can conclude from the evidence of an action that occurred.

$$\mathbf{concl} : (\mathbf{AC} \times \mathbf{AG}) \rightarrow \ \mathbf{PO}$$

For the default actions, $\texttt{create}(a, d_1)$ and $\texttt{comm}(a, b, \phi)$, we have:

$$\mathbf{pro}(\texttt{create}(a, d_1), b) = \top \tag{2.1}$$
$$\mathbf{pro}(\texttt{comm}(a, b, \phi), a) = \mathit{maySay}(a, b, \phi) \tag{2.2}$$
$$\mathbf{pro}(\texttt{comm}(a, b, \phi), c) = \top \quad (a \neq c) \tag{2.3}$$

$$\mathbf{concl}(\texttt{create}(a, d_1), a) = \mathit{owns}(a, d_1) \tag{2.4}$$
$$\mathbf{concl}(\texttt{create}(a, d_1), b) = \top \quad (b \neq a) \tag{2.5}$$
$$\mathbf{concl}(\texttt{comm}(a, b, \phi), b) = \phi \tag{2.6}$$
$$\mathbf{concl}(\texttt{comm}(a, b, \phi), c) = \top \quad (c \neq b) \tag{2.7}$$

This can be explained intuitively as follows: (2.1) agents do not need permissions for creating data. (2.2) in a communication, the source agent needs an authorization to say a policy. (2.3) other agents do not. (2.4) an agent who creates data can conclude that it is the owner of the data. (2.5) other agents cannot conclude anything from a creation action. (2.6) the target agent in a communication can conclude the corresponding policy. (2.7) other agents cannot conclude anything from a communication.

2.3.3 Logging actions

In our framework agents execute actions, and may need to justify them later on. We assume that agents have a basic log at their disposal to store securely facts, for example about the circumstances under which they perform actions, and to store evidence of actions they or other agents have performed. Note that we do not make any assumptions on whether agents share a logging device (for example on a central server), or if they each have separate devices. We model the log of an agent by the following basic definition:

Definition 2 (Logged action) *A* logged action *is a triple* $\texttt{lac} = \langle \texttt{act}_{id}, \Gamma, \Delta \rangle$ *consisting of an action instance* $\texttt{act}_{id} \in \mathbf{AC}^*$*, a set of facts* $\Gamma \subseteq \mathbf{PO}$ *(the conditions), and a set of action instances* $\Delta \subset \mathbf{AC}^*$ *(the use-once obligations). The* log of an agent *a is a list of logged actions.*

It is the choice of the agent whether or not to log an action. It is only important that individual log entries can not be forged, and cannot be modified later on. For example, it can be favorable to log the conditions under which an action was performed, or to log a communication of a policy from another agent to demonstrate that a subsequent action was allowed.

Additionally, an agent can log actions it performs by itself, including related conditions, i.e. facts about the current situation that the logging devices certifies to be valid, the time, the location, or the type of computer the agent uses to execute the action. We do not model this explicitly, but we assume that the agent obtains a secure *package* of facts from its logging device, represented by Γ. As an aside note that, to deal more efficiently with facts that remain true all the time, one could also have a set of global facts which then do not have to be included in each logged action.

The list Δ indicates the use-once obligations the agent consumes. The list Δ refers to instances of actions the agent did or promises to do, related to the action. We abstract away from the details of expressing promises, and instead assume we have a way to check if promises have *expired*. For example, if a policy states that the agent may modify a document provided it notifies someone within a day, then the agent must create a future reference to a notification action and fulfill this obligation within a day.

To prevent that logged actions are forged, the logging device must be somehow tamper-resistant. The logging device should protect some basic consistency properties of its log:

- An agent can log the same action at most once, i.e. there cannot be two different logged actions $\langle \mathtt{act}_{id}, \Gamma, \Delta \rangle$ and $\langle \mathtt{act}_{id}, \Gamma', \Delta' \rangle$ in the log for the same action $\mathtt{act}_{id}$.
- An action can only be used one time as a use-once obligation, i.e. an action act_{id} may not occur in the obligations Δ of two different logged actions in the log.

Now, we want to introduce the concept of system. To this end, we need the following definition:

Definition 3 (System state) *A system state is a collection s of logs of the different agents, i.e. a mapping from agents to lists of logged actions $s : \mathbf{AG} \to \mathbf{AC}^*$. We denote by S the collection of all states.*

The system model is defined as a labeled transition system:

Definition 4 (Transitions) *A system is a tuple: $\langle S, \mathcal{L}, \to \rangle$, where S is the powerset of S, introduced in Definition 3, $\mathcal{L} = \mathbf{AC}^* \times \mathcal{P}(\mathbf{AG})$ is the set of transition labels consisting of an action and a set of agents that log that action, and*

$$\to \subseteq S \times \mathcal{L} \times S$$

is the transition relation. We use the notation $s \xrightarrow{\mathtt{act},L} s'$ for $(s, (\mathtt{act}, L), s') \in \to$.

A transition *models an action happening in the system and being logged by some agents observing the action. Thus we have*

$$s \xrightarrow{\mathtt{act},L} s'$$

when $L \subseteq \mathbf{AG}$ and $\mathtt{act} \in \mathbf{AC}^*$. The full state s can be decomposed in substates for individual agents. The state of agent a is denoted $s(a)$.

Given the above transition between s and s', $s'(a) = s(a)$ if $a \notin L$ and $s'(a) = s(a).\overline{\texttt{act}}$ if $a \in L$ where $\overline{\texttt{act}}$ is a log of action `act` *by agent a. In other words, s' is the same as s except that* `act` *has been logged by the agents in L. $s_0 \in S$ is the initial state in which all logs are empty.*

An *execution* of the system consists of a sequence of transitions

$$s_0 \xrightarrow{\texttt{act}_1, L_1} \ldots \xrightarrow{\texttt{act}_n, L_n} s_n,$$

starting with the (empty) initial state s_0. The *execution trace* (tr) for this execution is $\texttt{act}_1 \ldots, \texttt{act}_n$. In a state s the log $s(a)$ of an agent a can also be seen as a trace of actions (by ignoring the conditions and obligations logged with the actions). As a's log is initially empty and a can only log actions that actually occur, a's log is a sub-trace of the execution trace, i.e. we have $s_n(a) \preceq tr$, where $\preceq$ denotes the sub-trace relation ($tr_1 \preceq tr_2$ iff tr_1 can be obtained from tr_2 by leaving out actions but maintaining the order of the remaining actions).

Example 2 (Execution trace) *For example the execution trace for the actions of Figure 3 is as follows:*

$$\texttt{create}(a, d_1), \texttt{comm}(a, b, \mathit{mayRead}(b, d_1)), \texttt{read}(b, d_1).$$

The log of agent b is only a subtrace:

$$\texttt{comm}(a, b, \mathit{mayRead}(b, d_1)), \texttt{read}(b, d_1)$$

2.3.4 Audits

Agents may be audited by some auditing *authority*, at some point in the execution of the system. This authority will audit the agent to find out whether the agent is able to account for the actions it initiated.

Before going into the details of how this can be implemented, we fix some notations: The knowledge of the auditing authority is represented by an *evidence* trace $\mathcal{E}$ which is a sub-trace of the execution of the system (up till now). For example the evidence trace could be the transaction log of some central database, or a log of some fileserver. Which actions are in $\mathcal{E}$ depends on the power (and possibly the interests) of the authority; a more powerful authority will in general be able to collect a larger evidence trace. When an auditor audits agents, using an evidence trace, agents are asked to account for the actions they performed in the evidence trace by providing valid proofs for them. If an action was logged by the agent, then the agent can also use the conditions or fulfilled obligations, logged with the action, in the proof. If the agent did not log the action it will have to provide a proof which does not depend on conditions or fulfilled obligations. This shows why it can be advantageous for agents to log actions.

Definition 5 (Action accountability) *We say that an agent a* correctly accounts *for an action* `act` *if it provides a valid proof of* $\Gamma_1, \Gamma_2, \Delta \vdash_a$ $\mathbf{pro}(\texttt{act}, a)$ *where* Γ_1, Γ_2 *and* Δ *must be empty if the agent a did not log the action at all, while otherwise the list* Γ_2 *may contain logged actions from the log of agent a where* Γ_1, Δ *are the conditions and obligations logged with the action* `act`*.*

Such a valid proof is called a justification proof. We will go into details about the definition of $\vdash$ (read *entails*) in the next section. The justification proof reveals new actions in Γ_2 and Δ. Accountability with respect to an evidence trace $\mathcal{E}$ is defined by taking into account also those new actions.

Definition 6 (Accountability) *We say an agent a* passes the audit (or accountability test) *$\mathcal{E}$, written $ACC(a, \mathcal{E})$, if it correctly accounts for all actions in $\mathcal{E}$ and for all actions revealed by proofs it provides.*

In providing a proof of accountability for an action, the agent may reveal actions that were not yet known to the auditing authority. These actions may be added to the actions to be audited i.e. the evidence trace. Clearly, it is also possible to have an authority which iteratively audits all agents involved in actions in the evidence trace. In this case newly revealed actions may require the authority to revisit agents or add new agents to its list. Since, the number of actions to be audited is always limited by the number of actions executed in the system we know the process will still terminate.

2.3.5 Honest strategy

A straightforward strategy for an honest agent a to be able to pass any audit is to derive the proof obligation $\mathbf{pro}(\texttt{act}, a)$, before executing an action `act`. If the proof needs conditions or obligations, then the action `act` itself must be logged.

Theorem 1 (Accountability of honest agents) *If agent a follows the* honest strategy*, then for any system execution and any auditing authority with evidence trace $\mathcal{E}$, we have that $ACC(a, \mathcal{E})$ holds.*

Proof 1 *The proof is straightforward. If the evidence trace $\mathcal{E}$ contains an action* `act` *for which* $\mathbf{pro}(\texttt{act}, a)$ *is not trivial, then it can provide a justification proof for it. The justification proof may refer to conditions, obligations or evidence of prior actions, which have been logged by the agent. The additional actions, thus revealed by the agent, can be justified by the agent in the same way.*

For the sake of simplicity we have assumed that the agents must produce all the justification proofs, when auditors ask for them. Nevertheless, variations are possible: for instance, in a different form of our system, the burden of producing the proofs may be left to the auditors. In another variation, the user may be required to *log* the proof (when possible, together with the action). Finally, when the auditor is trusted by the agents, they can submit (part of) their logs to the auditors. In this case, the auditor can single out the actions that can not be justified, and ask only for those actions for a justification by the agent. In any case, *finding* a proof may be expensive and difficult. Tools that automate the process of finding proofs, and replying to audits automatically are important here. We present tools for our framework in Chapter 3.

The way the auditor collects an evidence trace, and how bad actions can be observed by the auditor, has been left unspecified. For the first part, collecting an evidence trace, the trivial solution is to collect evidence of all the actions, and audit all of them. One could also use anomaly detection and audit the 'usual' actions less frequently. The second part, observing bad actions, poses a challenge as well. In our framework, the proof obligation function for creating any document yields the trivial policy. The problem of auditing which kind of data is introduced into the system is still needed however. It should be prevented for example that a user who owns a document, writes some secret data d_1 into it, in order to bypass security policy for d_1. This is a general problem of discretionary access control systems [44]. To model this step our framework should be extended with a second review of the evidence, after the justification proof has been validated (for example a human review). The details of this part of the auditing is beyond the scope of our framework.

2.4 Proof System

We now introduce a proof system, underlying the accountability relation (the $\vdash$ symbol). The proof system allows agents to derive, possibly referring to evidence in their log, certain policy formulae. The proof system (to be introduced below) is an extension of the sequent calculus for intuitionistic first-order logic, tailored to the justification proofs needed in the AC^2 framework.

2.4.1 Sequent notation

Throughout this section we use sequent notation for proof rules, a notation which is explicit about the assumptions used in proofs. To familiarize the reader with the sequent notation we report the standard proof rules for $\rightarrow$, $\wedge$ and $\forall$ in Figure 4. In the sequent notation Γ represents a set of assumptions, and Γ, ϕ denotes a set of assumptions that contains ϕ. Γ is usually referred to as the (logical) *context*. In the last line, as usual, in $\forall I$ y must be 'free' (i.e. not occurring in formulas in Γ), and in $\forall E$ z is an arbitrary value.

$$\frac{}{\Gamma, \phi \vdash \phi}\ I \qquad \frac{}{\Gamma \vdash \top}\ \top I$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi} \rightarrow I \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \phi \rightarrow \psi}{\Gamma \vdash \psi} \rightarrow E$$

$$\frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2} \wedge I \qquad \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_1} \wedge E_1 \qquad \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_2} \wedge E_2$$

$$\frac{\Gamma \vdash \phi(y)}{\Gamma \vdash \forall x.\phi(x)} \forall I \qquad \frac{\Gamma \vdash \forall x.\phi(x)}{\Gamma \vdash \phi(z)} \forall E$$

Figure 4: Natural deduction calculus for first-order predicate logic, in sequent notation.

The AC^2 proof system extends the $\rightarrow$, $\wedge$, $\forall$ fragment of (intuitionistic) first-order logic in the following way:

- In AC^2 different agents can derive different proofs. For this reason we annotate the entailment relation $\vdash$ with the name of the agent $\vdash_a$. For the example, the policy *owns*(a, d_1) allows agent a to derive any policy that only affects d_1. But this is not the case for agent b.

  The conclusion derivation function **concl**() links policies and actions, allowing agents to derive policies from actions. This is expressed by the following rule.

  $$\frac{\alpha \in \Gamma_2 \qquad \mathbf{concl}(\alpha, a)}{\Gamma_2 \vdash_a \phi},$$

  where Γ_2 contains actions that are in the log of agent a, executed either by him, or by others.

- For the semantics of *owns* (see the description of the grammar in Section 2.3.1) we must define which policies *affect* which data. We define the function **data_aff** : **PO** $\rightarrow \mathcal{P}(\mathbf{DA})$ for policies, such that if $\mathbf{data\_aff}(\phi(d_1)) = \{d_1\}$ then the policy ϕ only affects the data d_1. The semantics of the *owns* predicate is formalized as follows.

  $$\frac{\Gamma \vdash_a \mathit{owns}(a, d_1) \wedge ... \wedge \mathit{owns}(a, d_n) \quad \mathbf{data\_aff}(\phi) \subseteq \{d_1, ..., d_n\}}{\Gamma \vdash_a \phi}$$

  Basically, if a policy ϕ only affects the data $\{d_1, ..., d_n\}$, then if the agent a owns all the data $\{d_1, ..., d_n\}$, then ϕ can be derived by agent a.

- The *maySay*(a, b, ϕ) construct expresses the right to delegate a policy (see the description of the grammar in Section 2.3.1). This means that

$maySay(a, b, \phi)$ implies $maySay(a, b, \psi)$ if ϕ implies ψ, denoted $\phi \rightarrow \psi$. This is expressed as follows:

$$\frac{\vdash (\phi_1 \rightarrow ... \rightarrow (\phi_n \rightarrow \psi)) \quad \Gamma \vdash_a maySay(b, c, \phi_1) \wedge ... \wedge maySay(b, c, \phi_n)}{\Gamma \vdash_a maySay(b, c, \psi)}$$

The refine-rule allows to derive for example $maySay(b, c, \phi \wedge \psi)$ from the separate policies $maySay(b, c, \phi)$ and $maySay(b, c, \psi)$, and $maySay(b, c, \phi \rightarrow \psi)$ from $maySay(b, c, \psi)$). In other words agents who can say a certain policy ϕ can always say a more restrictive policies (with more conditions, or fewer privileges) to other agents. The first premise has an empty sequent to avoid that assumptions that hold only for agent a (and can not be said to b) are used to derive policies for b.

- In ordinary natural deduction there is only one type of assumption. Therefore a single context (Γ) is normally used. In the AC^2 framework the policies are derived from conditions and actions so we use three separate contexts to distinguish the three different types of assumptions.

Remark 4 (Proof by contradiction) *Note that we not included the* excluded middle *$(\neg\phi \vee \phi)$, or double-double negation $(\neg(\neg\phi) \rightarrow \phi)$ to allow for proofs by contradiction. Our logic is* constructive. *We believe that in our framework where the auditing authority may inquire several agents, the use of constructive proofs makes it easier for the authority to keep track of the chains of responsibilities. A proof by contradiction of the policy* there exists an agent who told me that I am allowed to ... *would not tell, for instance, the authority which authorization is being used.*

2.4.2 Sequent calculus

We now convert the proof rules (in natural deduction style) to a *sequent calculus*. Sequent calculi are due to Gentzen, and they are more suitable for analysis and automated proof search than natural deduction style proof systems. The full sequent calculus of AC^2 is shown in Figure 5: We use ϕ and ψ to denote policies, while α denotes an action. Sequents have the form $\Gamma_1; \Gamma_2; \Delta \vdash_a \phi$, where a is the agent doing the reasoning, and Γ_1, Γ_2 and Δ are three different contexts. The sequent Γ_1 is a list of policies. The sequent Γ_2 is a list of actions from the agent's log, which are used to derive conclusions using the conclusion derivation function **concl**, or as use-once obligations. The sequent Δ is a linear context, which contains a list of actions except that there is no way to reuse an action twice (see below). The linear context is used for use-once obligations. The empty context is denoted ν.

$$\frac{}{\Gamma_1, \phi; \Gamma_2; \Delta \vdash_a \phi}\ \mathrm{I} \qquad \frac{\Gamma_1; \Gamma_2; \Delta \vdash_a \phi \quad \Gamma_1, \phi; \Gamma_2; \Delta' \vdash_a \psi}{\Gamma_1; \Gamma_2; \Delta, \Delta' \vdash_a \psi}\ \mathrm{cut}$$

$$\frac{}{\Gamma_1; \Gamma_2; \Delta \vdash_a \top}\ \top\mathrm{R} \qquad \frac{\Gamma_1; \Gamma_2; \Delta \vdash_a \phi \quad \Gamma_1; \Gamma_2; \Delta' \vdash_a \psi}{\Gamma_1; \Gamma_2; \Delta, \Delta' \vdash_a (\phi \wedge \psi)}\ \wedge\mathrm{R}$$

$$\frac{\Gamma_1, \phi_1; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, (\phi_1 \wedge \phi_2); \Gamma_2; \Delta \vdash_a \psi}\ \wedge\mathrm{L}_1 \qquad \frac{\Gamma_1, \phi_2; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, (\phi_1 \wedge \phi_2); \Gamma_2; \Delta \vdash_a \psi}\ \wedge\mathrm{L}_2$$

$$\frac{\Gamma_1; \Gamma_2; \Delta \vdash_a \phi_1 \quad \Gamma_1, \phi_2; \Gamma_2; \Delta' \vdash_a \psi}{\Gamma_1, (\phi_1 \rightarrow \phi_2); \Gamma_2; \Delta, \Delta' \vdash_a \psi}\ {\rightarrow}\,\mathrm{L} \qquad \frac{\Gamma_1, \phi; \Gamma_2; \vdash_a \psi}{\Gamma_1; \Gamma_2; \vdash_a (\phi \rightarrow \psi)}\ {\rightarrow}\,\mathrm{R}$$

$$\frac{\Gamma_1, \phi(x); \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, \forall y.\ \phi(y); \Gamma_2; \Delta \vdash_a \psi}\ \forall\mathrm{L} \qquad \frac{\Gamma_1; \Gamma_2; \Delta \vdash_a \phi(x)}{\Gamma_1; \Gamma_2; \Delta \vdash_a \forall y.\ \phi(y)}\ \forall\mathrm{R}$$

$$\frac{\Gamma_1, \phi, \phi; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, \phi; \Gamma_2; \Delta \vdash_a \psi}\ \text{C-L}_1 \qquad \frac{\Gamma_1; \Gamma_2, \alpha, \alpha; \Delta \vdash_a \psi}{\Gamma_1; \Gamma_2, \alpha; \Delta \vdash_a \psi}\ \text{C-L}_2$$

$$\frac{\Gamma_1, \phi_1, \phi_2, \Gamma_1'; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, \phi_2, \phi_1, \Gamma_1'; \Gamma_2; \Delta \vdash_a \psi}\ \text{P-L}_1 \qquad \frac{\Gamma_1; \Gamma_2, \alpha_1, \alpha_2, \Gamma_2'; \Delta \vdash_a \psi}{\Gamma_1; \Gamma_2, \alpha_2, \alpha_1, \Gamma_2'; \Delta \vdash_a \psi}\ \text{P-L}_2 \qquad \frac{\Gamma_1; \Gamma_2; \Delta, \alpha_1, \alpha_2, \Delta' \vdash_a \psi}{\Gamma_1; \Gamma_2; \Delta, \alpha_2, \alpha_1, \Delta' \vdash_a \psi}\ \text{P-L}_3$$

$$\frac{\Gamma_1, \phi; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, (\alpha \overset{!}{\hookrightarrow} \phi); \Gamma_2; \Delta, \alpha \vdash_a \psi}\ !\!\rightarrow \text{L} \qquad \frac{\Gamma_1; \Gamma_2; \Delta, \alpha \vdash_a \phi}{\Gamma_1; \Gamma_2; \Delta \vdash_a (\alpha \overset{!}{\hookrightarrow} \phi)}\ !\!\rightarrow \text{R}$$

$$\frac{\Gamma_1, \phi; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, (\alpha \overset{?}{\hookrightarrow} \phi); \Gamma_2, \alpha; \Delta \vdash_a \psi}\ ?\!\rightarrow \text{L} \qquad \frac{\Gamma_1; \Gamma_2, \alpha; \Delta \vdash_a \phi}{\Gamma_1; \Gamma_2; \Delta \vdash_a (\alpha \overset{?}{\hookrightarrow} \phi)}\ ?\!\rightarrow \text{R}$$

$$\frac{\Gamma_1, \mathbf{concl}(\alpha, a); \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1; \Gamma_2, \alpha; \Delta \vdash_a \psi}\ \text{obs-act}$$

$$\frac{\Gamma_1; \nu; \nu \vdash_a \psi}{\Gamma_1', \mathit{maySay}(b, c, \Gamma_1); \Gamma_2; \Delta \vdash_a \mathit{maySay}(b, c, \psi)}\ \text{refine}$$

$$\frac{\mathbf{data\_aff}(\phi) \subseteq \{d_1, \ldots, d_n\}}{\Gamma_1, \mathit{owns}(a, d_1), ..., \mathit{owns}(a, d_n); \Gamma_2; \Delta \vdash_a \phi}\ \text{owns-L}$$

$$\frac{\Gamma_1, \mathit{maySay}(b, c, (\mathit{owns}(a, d))); \Gamma_2; \Delta \vdash_a \mathit{maySay}(b, c, \psi)}{\Gamma_1, \mathit{owns}(a, d); \Gamma_2; \Delta \vdash_a \mathit{maySay}(b, c, \psi)}\ \text{owns-maysay}$$

Figure 5: The proof system of AC^2.

The first ten rules in the proof system are standard rules for $\top$, initialization, cut and, left and right rules for conjunction, implication and universal quantification. This is basically the sequent calculus formalization of the $\wedge, \rightarrow, \forall$ fragment.

The next five rules are called *structural rules.* Structural rules are used in sequent calculi to model the behavior of the hypotheses explicitly. There are usually three types of structural rules: *contraction*, *permutation*, and *weakening.* Our proof system includes the contraction rules (C-L$_1$ and C-L$_2$) for the two non-linear contexts, and the permutation rules (P-L$_1$, P-L$_2$, and P-L$_3$) for the three contexts. There is no contraction rule for the context Δ to prevent agents to use a use-once obligation, twice in the same proof. Weakening is derivable and is not present as a separated rule. In this chapter, to keep the proofs simple and readable, we hide all the occurrences of permutation rules when presenting proofs. In our implementation (see Chapter 3) on the other hand the permutation rules are implemented.

Remark 5 (Weakening) *Weakening says that, if a certain proposition ϕ can be derived from the assumptions Γ, then ϕ can also be derived from Γ, ψ (for any ψ). Weakening is a derivable rule in our proofsystem. The linear context Δ allows weakening as well. The reason is that in the AC^2 framework, while it must be prevented that agents use use-once obligations twice, there is no motivation for preventing agents from consuming use-once obligations, unnecessarily.*

The next four rules are the implication left and right rules for the use-one and use-many obligations. They add and remove actions from the contexts Γ_2 and Δ. The final four rules, refine, owns-L and obs-act, do not occur in the usual logical systems.

The obs-act rule links actions and policies. It removes the conclusion of an action α, $\mathbf{concl}(a, \alpha)$, from Γ_1, and adds α to the non-linear context for actions Γ_2.

In the conclusion of the refine rule, the formula $maySay(b, c, \Gamma_1)$ is used as an abbreviation for list of policies $maySay(b, c, \phi)$ where ϕ in Γ_1. The action contexts, in the premise are empty, because we don't allow local facts to appear in policies for other agents unless they can be said to other agents. In our logic, as mentioned in Section 2.3, if an agent can derive a certain policy, it does not necessarily mean that it can communicate that policy to other agent. The presence of the free contexts Γ', Γ_2 and Δ is just to allow for *weakening*, which would not be a derivable rule otherwise.

The owns-L rule has the same structure as *falsity*-left in a standard sequent calculus. Although we do not include a separate symbol for *falsity* in our grammar, the formula $owns(a, d)$ or $\forall a, d.owns(a, d)$ behaves much like falsity, as it allows to derive all policies. Garg and Pfenning do not include falsity in their authorization logic, arguing that it is unnecessary and that it would only yield misleading policies [38]. In our setting on the other hand, a type of falsity (ownership of data) *is* present - $owns(a, d)$ allows to derive all policies about d - which we use to model discretionary policies - Falsity does not create any fundamental problems in our setting.

For the owns-L rule we would need to define $\mathbf{data\_aff}(\phi)$ for each policy ϕ. Set-theory would be required to define **data_aff** for compound policies, which is cumbersome (For instance $\mathbf{data\_aff}(\phi \wedge \psi) \subseteq \big(\mathbf{data\_aff}(\phi) \cap \mathbf{data\_aff}(\psi)\big)$ would have to hold.) Fortunately, the owns-L rule can be restricted to atomic policies, provided we add the rule owns-maysay. It can be shown that the proof system obtained in this way is sound and complete with respect to the one without owns-maysay: Soundness follows since, owns-maysay is provable using cut, the general form of owns-L and weakening. Completeness can be shown by proving (the general) owns-L, by case-analysis over ϕ. If ϕ is atomic, then the restricted form of owns-L is applicable. If ϕ is of the form $maySay(b, c, \psi)$, then ϕ can be stripped using owns-maysay and refine. If ϕ is of another compound form, then ϕ can be stripped using other rules. In owns-maysay, the formula on the right side of the entailment relation $\vdash_a$ is restricted to formulas of the form $maySay(b, c, .)$ to satisfy the sub-formula property, which is important for proof finding.

2.4.3 Properties and implementation

We refer the reader to Chapter 3 for technical details about the proof system. In Chapter 3 we show the implementation of a proof checker, using the type-checker Twelf (in Section 3.2), we show a cut-elimination theorem (in Section 3.3.1), which implies that our logic is consistent, and semi-decidable, and we show the implementation of a proof finder, using the theorem prover Prolog (in Section 3.3.2).

2.5 Scenario

In this section, to illustrate our approach, we give the details of a particular scenario: Employees of a consultancy firm exchange documents and policies from customers.

2.5.1 General setting

At R&D, a research and consultancy firm, employees work regularly with confidential data from customers. The firm and the employees of the firm are trusted to treat the data with care, and to protect data from illegitimate access.

The firm stores and processes a large amount of files produced or acquired during projects for customers. Customers specify how their data can be used:

They specify usage policies about how related documents can be used, and who should have access. Typically, customers allow access to their data only on a *need-to-know* basis, and they require data to be accessed in some secure way. In addition to these usage policies, the firm may specify additional policies, for example to avoid conflict of interests.

R&D has offices located at various sites, and each site has a storage for files. For example, at one site, with 300 employees, the storage contains 1.5 million files, in about 120.000 folders. More than half of the employees have *administrative* rights, i.e. they manage who may access files or folders. For example, a project folder is maintained and managed by a project manager who decides which employees should be granted access to the project folder. Subfolders of the project folder are used to group data, under a different access policy. For example, in each project folder there is often a folder with evaluations of individual employees, regarding the project. This subfolder is only accessible to management. It is safe to say that the filesystem contains no *top-secret* data, nor *public* data: Top-secret data, like documents from banks, require special care and clearance, and are stored on designated systems. Public data, like finished surveys and reports with public information, are stored in a kind of internal library accessible to everyone in the firm, or even the public.

In the rest of this section we assume that R&D (internally) audits the compliance of employees, by using AC^2, instead of using a more traditional access control mechanism. Employees use simple terminals (computers or laptops) to access the file storage. We assume that all access to the file storage is monitored and that employees can not bypass this monitoring. That auditors check the compliance of employees to the various usage policies by checking who accessed the file storage for which files and when. And we assume that employees use digitally signed emails to communicate policies to each other. It is not necessary for the auditors to know exactly which policies are being emailed by employees, because (as in Figure 3) when a policy is used by an agent, the auditors will find out about it during audits.

2.5.2 Examples

In this section we outline the features of AC^2 in a few sample runs. We give four different examples of policies and proofs. We highlight the use of administrative policies and the logging device. To represent the users of the system we use the fictional employees Alice (a), Bob (b) and Carol (c). The data that needs protection are a set of documents d_1, d_2, d_3 etc. We use x to denote an agent variable.

The grammar in this scenario contains the scenario-specific predicates $\mathit{mayRead}()$, $\mathit{mayWrite}()$ and $\mathit{isUsingV4}()$, The first two are permissions for agents to read and write data, while the third is a predicate about the kind of terminal an agent is using. If agents are using the right client software, then their logging devices will certify this predicate.

The function **data_aff**() for the permissions is defined by

$$\textbf{data\_aff}(\textit{mayRead}(a, d_1)) = \{d_1\}, \text{ and}$$
$$\textbf{data\_aff}(\textit{mayWrite}(a, d_1)) = \{d_1\}.$$

The proof obligation and the conclusion derivation function are as reported in Section 2.3, and additionally for the actions `read`() and `write`() the proof obligations are *mayRead*() and *mayWrite*(), respectively.

Now, a typical work flow is as follows: A customer gives a consultancy or research assignment to the firm. A contract is signed, containing, among other things, informal usage policies for the data related to the project. The account manager delegates the project to a project manager. The project manager must ensure that the usage policies specified in the contract are not violated.

The size of the file storage, and the number of different usage policies and documents, makes a centralized approach to policy administration cumbersome. Project managers would always be requesting new authorizations for their project members to the central authority. By giving project managers administrative rights over documents and folders, they can work more autonomously (avoiding the administrative bottleneck of a central authority). Let us give an example.

Example 3 (Administrative policies) *Alice is responsible for the authorizations regarding the documents in the folder for project PR. She must authorize other employees, to get the work done, while observing the firm's and the customer's policy. Technically, Alice is the owner of data in PR because she created (introduced) the files onto the filesystem. She has logged the following action:*

$\mathtt{act}_1 : \mathtt{create}(a, d_1).$

Alice can now derive $\textit{owns}(a, d_1)$*, but also any other policy about* d_1*. Evidence of the action* $\mathtt{act}_1$ *gives Alice the authority to authorize other employees for the file* d_1*.*

Bob works on a project PR. Alice sends him the authorization to read d_1*. She can derive:* $\textit{maySay}(a, b, \textit{mayRead}(b, d_1))$*. This policy is an administrative policy for Alice, which justifies the action:*

$\mathtt{act}_2 : \mathtt{comm}(a, b, \textit{mayRead}(b, d_1)).$

For Bob this is a justification for reading the document d_1*, so he logs Alice's communication for later. Now Bob reads the document* d_1*:*

$\mathtt{act}_3 : \mathtt{read}(b, d_1).$

Suppose the firm's auditor has observed Bob's action $\mathtt{act}_3$. The auditor asks Bob to justify it. Bob supplies the following proof:

$$\dfrac{\dfrac{}{[\mathit{mayRead}(b, d_1)]; \nu; \nu \vdash_b \mathit{mayRead}(b, d_1)}\ \text{I}}{\nu; [\mathtt{comm}(a, b, \mathit{mayRead}(b, d_1))]; \nu \vdash_b \mathit{mayRead}(b, d_1)}\ \text{obs-act}$$

Since Bob uses his log of action $\mathtt{act}_2$ the auditor finds evidence of the action, and subsequently asks Alice to justify $\mathtt{act}_2$. Her justification depends on her log of the action $\mathtt{act}_1$. Alice must show that she can derive the $\mathit{maySay}(a, c, \mathit{mayRead}(c, d_1))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{data\_aff}(\mathit{mayRead}(a, d_1)) = \{d_1\}}{[\mathit{owns}(a, d_1)]; \nu; \nu \vdash_a \mathit{mayRead}(b, d_1)}\ \text{owns-L}}{[\mathit{maySay}(a, b, \mathit{owns}(a, d_1))]; \nu; \nu \vdash_a \mathit{maySay}(a, b, \mathit{mayRead}(b, d_1))}\ \text{refine}}{[\mathit{owns}(a, d_1)]; \nu; \nu \vdash_a \mathit{maySay}(a, b, \mathit{mayRead}(b, d_1))}\ \text{owns-maysay}}{\nu; [\mathtt{create}(a, d_1)]; \nu \vdash_a \mathit{maySay}(a, b, \mathit{mayRead}(b, d_1))}\ \text{obs-act}$$

Alice's justification proof for $\mathtt{act}_2$ illustrates a key rule of the AC^2 proof system (*owns-L*). The proof refers to action $\mathtt{act}_1$.

Formally, the auditor could ask Alice to justify $\mathtt{act}_1$, but the justification proof is trivial. (The proof obligation for Alice for the action `create`() is the policy $\top$, and a proof of $\top$ is simply the proof rule $\top$R.) The auditor may want to review the type of document created (and thus owned) by Alice, its contents, and investigate whether copyrights or other agreements are applicable. The details of such a review are out of the scope of this thesis (see Section 2.3.4).

AC^2 allows *refinement* of administrative policies. Basically, this means that if users have the authorization to send policies to other users, then they may also send stricter policies. Let us give an example.

Example 4 (Administrative Refinement) *Suppose Bob is authorized to authorize Carol to read document d_2. Suppose he receives the policy: $\mathit{maySay}(b, c, \mathit{mayRead}(c, d_2))$, from another project manager, for this reason. The policy allows Bob to* say *the policy $\mathit{mayRead}(c, d_2)$ to Carol. But Bob wants to add an additional condition, to ensure that Carol uses the version 4 terminal: $\mathit{isUsingV4}(c)$, for security reasons for example. So Bob sends Carol a refined policy:*

$$\mathtt{act}_4 : \ \mathtt{comm}(b, c, \mathit{isUsingV4}(c) \rightarrow \mathit{mayRead}(c, d_2)).$$

The justification proof for $\mathtt{act}_3$ *relies on the fact that,*

$$\vdash \mathit{mayRead}(c, d_2) \rightarrow (\mathit{isUsingV4}(c) \rightarrow \mathit{mayRead}(c, d_2)),$$

holds, i.e. it is a tautology. In our framework this tautology entails that also

$$\mathit{maySay}(b, c, \mathit{mayRead}(c, d_2)) \rightarrow \mathit{maySay}(b, c, \mathit{isUsingV4}(c) \rightarrow \mathit{mayRead}(c, d_2)),$$

holds (by the refine rule). Bob can derive the authorization to communicate a refined *policy.*

Let us see how Carol can use the policy. Carol is not always using the right version of the client. Carol is a researcher who also uses another type of terminal for research work. Carol, when she accesses d_2*,*

$\mathtt{act}_5 : \ \mathtt{read}(c, d_2),$

she logs her action together with the (favorable) fact that she is using the right version. Her log contains the action of reading d_2 *as follows:*

$$\langle \mathtt{act}_5, \mathit{isUsingV4}(c), \nu \rangle.$$

Later, Carol can use the log entry of $\mathtt{act}_3$*, together with the log entry of* $\mathtt{act}_4$ *to prove (to an auditor) that she was allowed to read* d_2*:*

$$[\mathit{isUsingV4}(c)]; [\mathtt{act}_3]; \nu \vdash_c \mathit{mayRead}(c, d_2).$$

The work at the firm is dynamic. It happens frequently that authorizations, for accessing documents, are outdated. The auditing approach of AC^2 however ensures that, despite wrong or outdated authorizations consultants can continue their work without delay. We give an example.

Example 5 (Availability) *Alice has given Bob the authorization to read documents of the project PR, like in Example 3. On Friday, Bob finishes his work on document* d_1*, which needs to be delivered to the customer by Monday. Unexpectedly, he decides to have the researcher Carol review some charts in the document over the weekend, because Carol is an expert at this. Carol has not been authorized by Alice to read* d_1*. Unfortunately, Alice has already left the office. Bob knows Alice well and is sure she will agree. Bob takes the responsibility of any sanctions, by authorizing Carol himself. Carol subsequently reads the document* d_1*:*

$\mathtt{act}_6 : \ \mathtt{comm}(b, c, \mathit{mayRead}(c, d_1)).$
$\mathtt{act}_7 : \ \mathtt{read}(c, d_1).$

At this point Carol can justify her action, by referring to the authorization sent by Bob. Bob however did not have the authorization to authorize Carol. The action $\mathtt{act}_6$ *is not (yet) in compliance with the firm's policies. He writes Alice an email about this, asking her (a-posteriori) authorization. When Alice comes back to office, she checks her emails and finds Bob's authorization request waiting. Alice can derive, from* $\mathit{owns}(a, d_1)$*, the policy:* $\mathit{maySay}(a, b, \mathit{maySay}(b, c, \mathit{mayRead}(c, d_1)))$*, which is the justification for the following communication:*

$\mathtt{act}_8 : \ \mathtt{comm}(a, b, \mathit{maySay}(b, c, \mathit{mayRead}(c, d_1))).$

The latter example highlights the flexibility of the audit-based approach in combination with administrative policies. With a file system of thousands of files and different authorizations and hundreds of employees and changing projects, it is common for authorizations to be outdated or simply wrong. The AC^2 framework allows consultants to continue their work without delay. Policies can be supplied *on demand*.

Suppose the firm's auditor asks Bob for a justification for $\texttt{act}_6$, Bob can use his log of $[act8]$. Bob's justification proof for $\texttt{act}_6$ is as follows:

$$\dfrac{\dfrac{}{[maySay(b,c,mayRead(c,d_1))];\nu;\nu \vdash_b maySay(b,c,mayRead(c,d_1))}\,\text{I}}{\nu;[\texttt{comm}(a,b,maySay(b,c,mayRead(c,d_1)))];\nu \vdash_b maySay(b,c,mayRead(c,d_1))}\,\text{obs-act}$$

The auditor, when checking Bob's justification proof for $\texttt{act}_6$, finds that the proof relies on Bob's log of Alice's action $\texttt{act}_8$. In turn the auditor might ask Alice to account for action $\texttt{act}_8$. She can use $\texttt{act}_1$ ($\texttt{create}(a, d_1)$) to derive a justification proof. The proof is similar to the justification proof shown in Example 3, but it uses refinement twice. We omit the names of rules for reasons of space. From top down, the steps are owns-L, refine, refine, owns-maysay, owns-maysay, obs-act.:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{data\_aff}(mayRead(c,d_1)) = \{d_1\}}{[owns(a,d_1)];\nu;\nu \vdash_a mayRead(c,d_1)}}{[maySay(b,c,owns(a,d_1))];\nu;\nu \vdash_a maySay(b,c,mayRead(c,d_1))}}{[maySay(a,b,maySay(b,c,owns(a,d_1)))];\nu;\nu \vdash_a maySay(a,b,maySay(b,c,mayRead(c,d_1)))}}{[maySay(b,c,owns(a,d_1))];\nu;\nu \vdash_a maySay(a,b,maySay(b,c,mayRead(c,d_1)))}}{[owns(a,d_1)];\nu;\nu \vdash_a maySay(a,b,maySay(b,c,mayRead(c,d_1)))}}{\nu;[\texttt{create}(a,d_1)];\nu \vdash_a maySay(a,b,maySay(b,c,mayRead(c,d_1)))}$$

Use-once obligations can be used to enforce procedures, demanding that users perform a certain action before or after performing another. We give an example of use-once obligations.

Example 6 (Use-once obligations) *Alice decides to give Bob administrative rights, in case he needs another review at a late hour. The firm's procedures however require that she can provide a list of employees who have had access to the documents. She wants to receive a short email from Bob, explaining the circumstances, for each time Bob gives access to another employee. She issues the following policy:*

$$\phi = \texttt{notify}(a) \xrightarrow{!} \forall x.\ maySay(b, x, mayRead(x, d_1)).$$

Alice's action is:

$\texttt{act}_8 :\ \texttt{comm}(a, b, \phi).$

At a later time during the project, Bob needs Carol's help again. He notifies Alice, and keeps the evidence of this for later by logging the action, and he authorizes Carol:

$\texttt{act}_9 :\ \texttt{notify}(a).$
$\texttt{act}_{10} :\ \texttt{comm}(b, c, \mathit{mayRead}(c, d_1)).$

and he logs his action with a reference to the use-once obligation he is using up. His log contains: $\langle \texttt{act}_9, \nu, \nu\rangle$ *and* $\langle \texttt{act}_{10}, \nu, \texttt{act}_9\rangle$*.*

When the auditor asks Bob for a justification, Bob can prove:

$$\nu; [\texttt{act}_8]; [\texttt{act}_9] \vdash_b \mathit{maySay}(b, c, \mathit{mayRead}(c, d_1)).$$

Bob cannot authorize anyone without notifying Alice because the logging device does not permit him to point twice to the same message. Basically, the separated linear context used in the proof system prevents him from completing a second justification proof. Alice can be sure that she gets an email each time another employee gets access to a project document.

An important assumption in AC^2 is that auditors can observe the critical actions. In the examples of this section we assume that the auditor only audits access the file storages. Suppose for the sake of example that Bob can also attach documents to emails, and send them to people outside the firm. We introduce an action `email`(), with parameters for the sender, receiver, and the document that is attached. This type of action is not audited by the firm's auditor.

Example 7 (Undetected action) *Bob wants to send the document* d_3 *to Carol, who now works for a new employer at a different firm. For example Bob might send an email with the document* d_3 *attached, to Carol, who receives and reads the document* d_3*. Let us go over the actions.*

$\texttt{act}_{11} :\ \texttt{comm}(a, b, \mathit{mayRead}(b, d_3)).$
$\texttt{act}_{12} :\ \texttt{read}(b, d_3).$
$\texttt{act}_{13} :\ \texttt{email}(b, c, d_3).$
External action: $\texttt{read}(c, d_3).$

The auditor only observes Bob's action $\texttt{act}_{12}$*. Bob can justify this action by using evidence of the communication* $\texttt{act}_{11}$*, in case of an audit. Bob's email, and Carol's action outside the firm are not observed by the auditor.*

The firm's auditor audits only the access to the file storage, and does not observe the illegal action $\texttt{act}_{13}$ performed by Bob, nor the external action by Carol. To solve this problem, the firm's security officer should extend the audit trail to include also such emails, or relay on the auditor of Carol's new firm, to detect and report the reading of d_3 by Carol.

2.6 Related Work

Our work is related to a number of different areas in access control research. We discuss the different areas below.

2.6.1 Audit logs

Logging and auditing have always been considered central in security, and in particular central to a successful practical implementation of access control [76]. Jajodia et al. discuss explicit requirements for logging and auditing user actions on a database [47]. Logging and auditing is usually performed, not as a replacement of, but in addition to an (a-priori) access control system. In addition sometimes the access control system itself is audited, for flaws or errors. An audit of the access control mechanism can be sufficient when it is certain that the mechanism can not be turned off between the audits. In our framework we focus on auditing individual actions, and in principle we can assume that the a-priori access control system is turned off completely, providing only basic authentication of users. Audits are sometimes used to observe unauthorized access, or the bypassing of access control mechanisms [77]. Observing such misbehavior is a general problem, which plays a role also in our setting. In our framework it is particularly important that the audit trails can not be tampered with by users, and that it is hard for users to prevent that crucial actions are being registered in the audit trail [77].

2.6.2 Overriding

Rissanen et al. propose a method for overriding in the Privilege Calculus, a type of access control system [74]. They focus on how to find suitable auditors in a hierarchy of auditors, to justify each override. In our framework on the other hand, we focus on the form of the justifications and we do not assume any hierarchy of auditors. Rissanen et al. list a number of reasons to use a more flexible mechanism than the traditional a-priori access control systems. Their main motivation is that emergency situations can not be encoded in policies. We take a different approach by providing a way (through the use of administrative policies) to change the authorization of users to adapt to the new situation. In our framework, for example, a user can be authorized to give authorizations to other users, a-posteriori, for example for actions during an emergency situation. Also in a medical setting, Longstaff et al. [64] give a high-level description (a UML model) of a medical information system with the possibility to override access control decisions. They focus on the conditions under which an override is justified. In a different setting, Shmatikov and Talcott audit users to discover the violation of DRM licenses. They use a reputation system to discourage bad behavior, and encourage good behavior [81]. In our framework we do not make assumptions about specific sanctions imposed by auditors, but in certain settings it may be an interesting future possibility to combine our framework with the reputation system of Shmatikov.

2.6.3 Logics in access control

In our framework we use on purpose a simple policy language based on first-order logic, where first-order quantification allows to express groups of objects and subjects. For the sake of simplicity we did not go into the details of all practically useful constructs, such as constructs regarding time, groups of subjects, or objects. It has been shown, however, that first-order logic supports most access control policies [40]. Unfortunately, first-order quantification is only semidecidable, which means that there is a procedure that finds all the proofs of statements, but this procedure may not terminate when no proof exists. The authors argue that undecidability is not a severe problem in the setting of PCA [12], because agents must find proofs, not the PCA reference monitor. An analysis of the expressivity of first-order logics was presented by Halpern and Weismann who discuss in particular the practical use of certain decidable subsets of first-order logics for access control [40]. A number of access control frameworks are based on Datalog, e.g., Delegation Logic [59], the RT framework [62] and Binder [30]. It has been argued however that Datalog has severe limitations and that a more expressive language should be used instead [61]. Datalog with constraints has been used in the Cassandra system to implement an Electronic Health Record system. Theoretically, the policy language used in the Cassandra system is undecidable [18]. For a more lengthy discussion of the different aspects of logic-based access control systems we refer the reader to a survey by Abadi [9].

Many systems use the *says* construct, which models the communication of a (security) statement between users. In these proposals, the receiver, before concluding that the communicated statement is true, must check some side-condition, such as whether the sender is trusted, or an authority about such a statement. This side-condition is absent in our framework, because in our framework the agent who sends the policy remains responsible for it. If a policy is used to justify an action, auditors may find out about it, and they may ask the original sender of the policy for a justification.

2.6.4 Proof checking and proof systems for access control

The proof system proposed here differs from normal first order logics in the use of the linear context to model use-once obligations, the refinement rule and the rule that allows any policy to follow from the owns predicate. We do not use all the linear operators and logical rules, because many constructions do not yield useful policies. Independently, Garg et al. [37] have presented a similar system recently (here use-once obligations are referred to as consumable credentials). They have claimed to be the first to use linear logic in an access control setting. Differently from us, they use all the operators and logical rules of linear logic, but they conjecture that it is sufficient to use some subset of linear logic (like we do).

The cut-elimination theorem, proven for our logic (see Chapter 3), shows that the logic is well-behaved: it is semi-decidable and consistent. Independently, the same theorem, for a different authorization logic, was used recently by Garg and Pfenning [38]. Their logic is a constructive sequent calculus (like ours) and they prove that the cut-rule is admissible (like we do). Garg and Pfenning refer to this as the non-interference property of the logic. They discuss in detail the precise consequences of cut-elimination with respect to access control policies.

BLF [90] is an implementation of a Proof-Carrying-Code framework that uses both Binder and Twelf. In this framework, developers of a program include a proof that the program is safe, while consumers can check the proof to get confidence about the program. This is based on two ideas: First, that checking the correctness of a proof is relatively easy, compared to finding one. Second, that finding the proof, that a program is safe, is easier for the developer of the program than for arbitrary consumers of the program. Like in our framework, the proofs are written and checked using Twelf. In BLF, the proofs for complex programs can become lengthy. To solve this, an alternative procedure was proposed, using only hints from which the full proof can be derived, instead of giving the full proof. Such a variation could be a possibility also for our framework.

More related to our auditing by means of proofs, Appel and Felten [12] propose the Proof-Carrying Authentication framework (PCA), also implemented in Twelf. Their system is implemented as an access control system for web servers. Differently from our work, PCA's language is based on a higher order logic that allows quantification over predicates. The disadvantage of using higher-order logic is that proof search is in general undecidable and that properties like consistency must be proven separately for individual settings. In our case, semi-decidability and consistency hold for all the different settings.

2.6.5 Access control

A number of access control systems use a policy language based on XML to express access control policies [66, 53, 89]. We do not use an explicit XML syntax here because we are interested in the formal properties of our logic, which are more easily shown when using logical formulas and logics.

XACML [66] is a type of attribute-based access control system. An XACML policy consists of a list of rules. Each rule is a tuple of *action*, *subject*, *object*, *condition* and an *effect*. The latter can be either permit, deny, or not applicable, and behaves like an intermediate decision in the sense that this value may or may not be, for example when overruled by another part of the policy, the final outcome of the decision. In our framework overruling a positive decision is not possible: when a policy allows an action then this is always final. Also, if no policy applies access is always denied. The XACML decision vfalues *negation* and *not applicable* hence coincide in our framework. The first three elements of the XACML tuple are in our logic contained in the action expressions. Conditions are expressed using logical implication $\rightarrow$. The *maySay* predicate can not yet be translated to an XACML policy, but apparently the new version of XACML should allow the expression of administrative policy.

XrML [89], a rights language designed for DRM, is similar to XACML, except that in XrML some form of administrative policy *is* possible; an XrML license may contain a special *flag* allowing the further distribution of the same license. The *maySay*()-construct in AC^2, and the possibility of nesting of the *maySay*()-construct, is basically a refined form of the XrML distribution flag: in AC^2 one can also specify all the sender and receiver pairs, in the *maySay* predicate, along a delegation chain.

Bandman et al. define a type of cascaded administrative policies, using regular expressions to constrain the users that can receive them [16]. The *maySay*() construct allows to express similar policies, although we use first-order predicates instead of regular expressions.

In AC^2 we distinguish between conditions and obligations, that can be logged by agents. This use of obligations was inspired by Sandhu and Park's UCON model [68], in which the decision is modeled as a reference monitor that checks the three components: ACL, Conditions and Obligations. Differently than in the UCON model, we do not assume a security monitor, to check that these conditions are valid, but a logging device to certify (or sign) conditions. UCON's *post- and pre-obligations* are supported in our framework, but *ongoing obligations* would require a special construction. Obligations are used with a different meaning in E-P3P [53] and in XACML [66]. In these frameworks, obligations are call-back functions that are executed by the access control mechanism at the time a request is evaluated. In our framework there is no central security monitor that evaluates access requests, but obligations are actions to be performed by the agents requesting access.

2.7 Conclusions

In this chapter we have described a framework called Audit-based Compliance Control (AC^2). AC^2 is targeted at dynamic collaborative environments, such as consultancy firms, or hospitals, where a small group of users exchange, modify and refine a large number of documents and policies.

In our framework we assume that no security monitor is present to prevent unauthorized actions, but that critical actions are monitored and that users can be asked to justify their actions, a-posteriori. Our framework uses a simple, but expressive, policy language based on first-order predicate logic, and extended with an *owns* predicate and a *maySay* predicate. To reason about policies AC^2 uses a formal proof system which can be implemented straightforwardly (see Chapter 3). To show how our framework can be used in practice we have discussed a common scenario: employees of a consultancy firm, processing various confidential documents. In the following chapters we show how our framework can be used in other types of dynamic collaborative environments. In particular, for the protection of health records in a hospital (Chapter 4, and for the protection of customer data across enterprises (Chapter 5). We implemented our logic by building a proof finder (using Prolog) and a proof checker (using Twelf). We refer to Chapter 3 for a description of both tools.

To the best of our knowledge, the framework proposed here is the first to describe a logic for (administrative) policies combined with a-posteriori compliance checks of performed actions. Checking authorizations of users after the access yields a flexible system, and avoids the usual costs of unavailability due to flawed or outdated policies.

A crucial requirement to deploy our framework successfully is that the actions of the users can be monitored, and that the users performing these actions can be held accountable. This may exclude certain settings, such as the internet, where monitoring user actions is infeasible and holding users accountable is even harder. In other settings these requirements are not unreasonable. Recent laws and legislation demand that enterprises and hospitals account for the disclosure of confidential documents [84, 85]. Tracing which employees have accessed data, and demanding justifications afterwards, is a flexible way of ensuring that procedures are being followed, without affecting the availability of data. High availability discourages bad security practices. Recall the example in the introduction. If the senior Charlie was not allowed to authorize an employee to review the charts, he would have been tempted to bypass the security measures, say by sending the file conspicuously by email, in order to get the work done quickly.

2.8 Acknowledgements

I am grateful to Ricardo Corin, Sandro Etalle, Jerry den Hartog, Gabriele Lenzini, and Ioan Staicu for kickstarting the Privacy in an Ambient World project with an inspiring publication [25] and getting the different research partners of the project involved. It was a pleasure to work with them and to improve and implement their initial model.

I also thank Ricardo Corin and Jerry den Hartog for designing the logging

mechanism, and an improved definition of accountability in AC^2, and I thank Jan Cederquist for helping me with choosing the right formalization (intuitionistic logic in sequent notation) for AC^2, and choosing the right implementation in Twelf.

Chapter 3

Proof Finding and Proof Checking

3.1 Introduction

In the previous chapter we have introduced the AC^2 framework. In this chapter we focus on the technical details of the AC^2 proof system and show how to find and check justification proofs automatically.

In AC^2 agents perform (and log) actions, while auditors may audit the actions of the agents a-posteriori, i.e. after their occurrence. If an auditor asks the agent to justify a particular action, then the agent must present a justification proof. The justification proof consists of two parts: 1) evidence (log-entries, for example of communications), and 2) a logical proof (a derivation of a certain permission, for example for sending a policy). The auditor has to check two things: 1) that the evidence is sound, that is, that the log entries are genuine 2) that the proof is correct. If the proof is correct, and the evidence is sound, then we say that the agent is accountable for that action, i. e. that the agent was allowed to perform the action. Recalling the definition of action accountability (Definition 5): *An agent a accounts for an action* `act` *if it provides a valid proof of* $\Gamma_1, \Gamma_2, \Delta \vdash_a \mathbf{pro}(\texttt{act}, a)$, where $\mathbf{pro}(\texttt{act}, a)$ is the proof obligation for action `act`.

In the AC^2 framework the agents are responsible for keeping evidence of facts and past actions, and for finding justification proofs for their actions. This is very different from conventional access control where agents simply send requests and let the reference monitor decide whether the request is allowed or not. Recall that the AC^2 justifications proofs are proofs of propositions of the form $\Gamma_1; \Gamma_2 \vdash_a \phi$, where $\phi = \mathbf{pro}(\alpha, a)$ the proof obligation for an action α. In this chapter, because finding the justification proofs manually can be laborious, we present a prototype of an automated proof finder, called PPF. The proof finder basically works as follows: Given a set of evidence (a list of logged actions), an agent name a and a predicate ϕ, it tries to find a proof of ϕ, a proof of the form $\Gamma_1; \Gamma_2; \Delta \vdash_a \phi$, where the logical contexts are formed by the evidence supplied by the agent. If it finds a proof it prints the proof to allow the agent to store it, or to send it to the auditor. This is also slightly different from most proof finders (also known as theorem provers), which answer yes or no, without printing a formal proof.

The auditor, on the other hand, has to check the proofs provided by the agents. Manual proof checking is both error-prone and slow. Therefore, the second contribution of this chapter, is a proof checker for the AC^2 justification proofs, called TPC. Basically, given a proof of $\Gamma_1; \Gamma_2; \Delta \vdash_a \phi$, the proof checker checks whether the proof is correct or not.

Let us show outline how the proof checker and the proof finder work together in the AC^2 framework. Suppose that an agent a has performed an

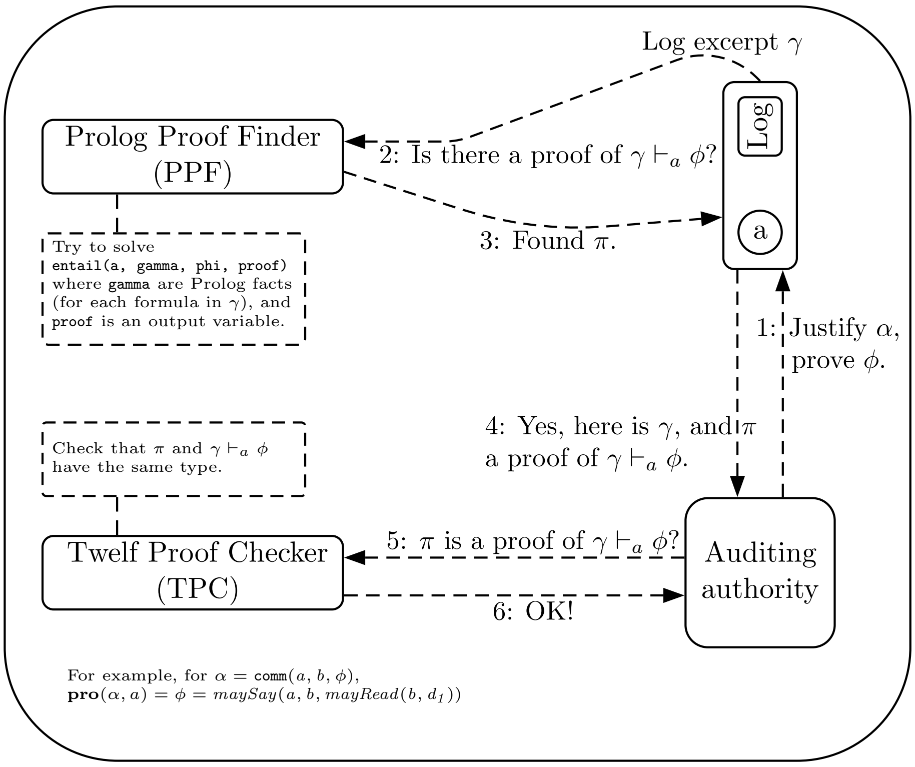

Log excerpt γ
Prolog Proof Finder (PPF)
2: Is there a proof of $\gamma \vdash_a \phi$?
Log
a
3: Found π.
Try to solve entail(a, gamma, phi, proof) where gamma are Prolog facts (for each formula in γ), and proof is an output variable.
1: Justify α, prove ϕ.
4: Yes, here is γ, and π a proof of $\gamma \vdash_a \phi$.
Check that π and $\gamma \vdash_a \phi$ have the same type.
Twelf Proof Checker (TPC)
5: π is a proof of $\gamma \vdash_a \phi$?
Auditing authority
6: OK!
For example, for α = comm(a, b, ϕ), $\mathbf{pro}(\alpha, a) = \phi = \mathit{maySay}(a, b, \mathit{mayRead}(b, d_1))$

Figure 6: Use of the proof checker (TPC) and the proof finder (PPF) in the event of an audit.

action α and that the auditing authority wants a to justify it. The steps of such an audit are represented in Figure 6. First, (1) agent a is audited for action α. There agent a computes $\mathbf{pro}(\alpha, a) = \phi$, and selects an excerpt γ of its log and (2) tries to find a proof of $\gamma \vdash_a \phi$ with the proof finder (PPF). Then (3) the proof π and the excerpt γ are sent to the auditor for checking

(4) and finally, (5) the auditor checks that π is indeed a proof of $\gamma \vdash_a \phi$ by using the proof checker (TPC) (6).

The proof finder and the proof checker use the same syntax for proofs, but they are implemented in completely different ways:

- The proof checker is implemented using Twelf, a type-checker. The implementation of the 20 inference rules in Twelf consists of about 100 lines of code.

- The proof finder, on the other hand, is implemented using Prolog. It prints proofs in Twelf syntax though (every proof is a type declaration). Besides a representation of the inference rules, it contains extra modules for the generation of the proof in a format appropriate for the proof checker. There are about 400 lines of Prolog code.

There are two reasons for this difference: Proof finding is difficult (only semi-decidable) for the AC^2 proof system. This is a direct consequence of the expressiveness of the AC^2 policy language, and is a common problem for (fragments of) first-order predicate logic. We cannot use standard first-order logic theorem provers, because they do not take into account the special connectives *maySay*(), *owns*() and the associated rules. We implement a proof finder using the resolution procedure of Prolog. To allow this to work we first prove a technical result about the AC^2 proof system in Section 3.3.1: a *cut-elimination theorem*. Cut-elimination implies consistency, and semi-decidability, and it allows us to implement the proof finder.

Proof checking on the other hand is easier than proof finding. At the same time, we should stress that the correctness of proof checking is crucial in the AC^2 audit procedure: If the auditor uses a proof checker that is flawed, then agents would be able to perform illegitimate actions, and get away with it. The question arises of how to be sure that the proof checker is correct (a version of the question *Quis custodiat ipsos custodes*). Since it is not possible to try out all the proofs (to show that bad ones are rejected and good once accepted), we can only verify the proof checker manually, by inspecting its code. In the field of software verification, the lines of code that have to be verified by hand (before one can trust it), are referred to as the *trusted code base*.

In this chapter we implement the proof checker by using the logical framework Twelf [73]. Twelf uses the *propositions-as-types* correspondence, also called the Curry-Howard isomorphism. Proof checking in Twelf thus reduces to type-checking, which is a simple procedure (compared to proof finding).

Twelf allows to keep the trusted code base to a minimum, thereby facilitating manual inspection. Indeed, our proof checker, introduced in the next section, consists of less than a hundred type definitions, and it is straightforward to verify (manually) that they are precise translations of the AC^2

grammar and the derivation rules of the AC^2 proof system. The syntax of proofs output by the proof finder, and checked by the proof checker is shown below.

```
thm: (entail a
   nil
    (cons (creates a d1) nil)
     nil
      (maysay a b (mayread b d1)))=
(obs_act
    (owns_maysay (refine
         (owns_left data_mayread (map_cons map_nil)(append_cons append_nil))
         (map_cons map_nil)
         (append_cons append_nil)))
    concl_creates).
```

This sample shows the syntax of a proof by agent a of the administrative policy $maySay(a, b, mayRead(b, d_1))$, using the fact that she created the data. The corresponding proof in sequent calculus notation is given below. In Twelf syntax the proposition (the final term on the bottom of the proof below) is the term before the $=$ sign, and its proof is written after the '=' sign.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{data\_aff}(mayRead(a, d_1)) = \{d_1\}}{[owns(a, d_1)]; \nu; \nu \vdash_a mayRead(b, d_1)}\ \text{owns-L}}{[maySay(a, c, owns(a, d_1))]; \nu; \nu \vdash_a maySay(a, b, mayRead(b, d_1))}\ \text{refine}}{[owns(a, d_1)]; \nu; \nu \vdash_a maySay(a, b, mayRead(b, d_1))}\ \text{owns-maysay}}{\nu; [\mathtt{create}(a, d_1)]; \nu \vdash_a maySay(a, b, mayRead(b, d_1))}\ \text{obs-act}$$

3.2 Proof Checking

Proof checking is implemented using Twelf. Twelf, an implementation of the Edinburgh Logical Framework [70], was designed for experimenting with deductive systems. Twelf has been used to implement a variety of logical systems and programming languages [73]. Twelf provides a convenient syntax for type definitions, it provides a type checker, and it uses a short notation for proofs, which makes it convenient to use in settings where proofs are generated and checked remotely [12, 90, 65], such as in our setting by auditing authorities. In this section we show the code of the proof checker, and the syntax of the justification proofs that can be checked by the proof checker.

3.2.1 Twelf code

Our proof checker consists of Twelf's type-checker and a list of type definitions, called *signature* that represents the grammar and proof rules of AC^2. Let us introduce this signature, before showing how our proof checker can be used in a specific scenario. Our signature is based on the LF encoding for first-order logic [41]. The basic types are shown in Figure 7. The main types are a type `obj`, for data and agents, and a type `pred` for predicates

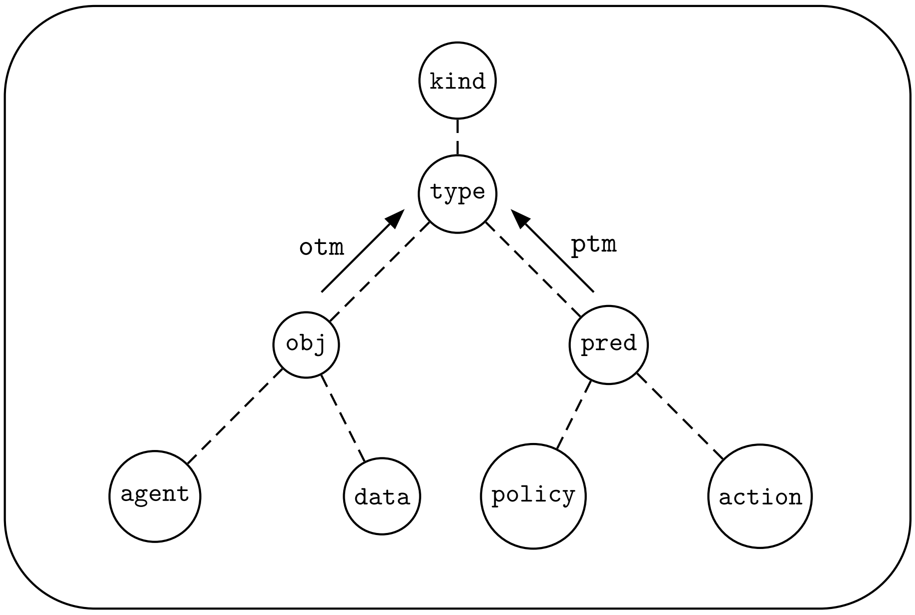


Figure 7: Type diagram for the implementation in Twelf.

and actions. They are instances of Twelf's meta-type `type`. In the literature `pred` and `obj` are sometimes refered to as *individuals* and *propositions* [41]. We define policies and actions as different subtypes of `pred`, and agents and data as different subtypes of `obj`.

```
obj: type.
pred: type.
policy: pred.
action: pred.
agent: obj.
data: obj.
```

To declare the (sub)type of agents, data, or atomic predicates of the policy language *type constructors* are used. The type constructors `otm`, for `obj`, and `ptm`, for `pred`, are declared as follows.

```
otm: obj -> type.
ptm: pred -> type.
```

The meta-logic function `->` used here goes from `type` to `type` (see Figure 7). Let us give one example of the use of the type constructors. To declare that the name *john* is of type *agent*, one declares `john: otm agent`.

The basic atomic predicates in AC^2 are *owns*() and $\top$ (written in the code as `true`). The basic actions are `create`() and `comm`(). The connectives are *maySay*(), $\rightarrow$, $\wedge$, and $\forall$ (for quantification over agents and data variables), and $\overset{!}{\hookrightarrow}$, and $\overset{?}{\hookrightarrow}$ for use-once-obligations and use-many-obligations. Their types are defined below.

```
true: ptm policy.
owns: otm agent -> otm data -> ptm policy.
comm: otm agent -> otm agent -> ptm policy -> ptm action.
creates: otm agent -> otm data -> ptm action.

maysay: otm agent -> otm agent -> ptm policy -> ptm policy.
imp: ptm policy -> ptm policy -> ptm policy.
forall: (otm T -> ptm policy) -> ptm policy.
and: ptm policy -> ptm policy -> ptm policy.
?imp: ptm action -> ptm policy -> ptm policy.
!imp: ptm action -> ptm policy -> ptm policy.
```

For instance, `(owns a d)` is the representation of the policy stating that agent `a` owns the data `d`. The expression `(owns d a)` violates the type definitions (Twelf will throw an error when encountering it). Agents and data are both subtypes of `obj`, so a single quantifier `forall` with a variable type `T` suffices. In Twelf variables are denoted with a capital letter, and their type of variables can usually be omitted as Twelf infers it from how they are used.

Example 8 *Suppose the policy grammar, in some scenario, includes permissions for reading and writing, the policy that the permission to write implies the permission to read is written as follows:*

```
(forall[a] (forall[d] (imp (maywrite a d) (mayread a d) ) ) )
```

The AC^2 proof system is a sequent calculus. Sequents are lists of policies (the first context) or lists of actions (the second and the third context). The following lines declare types for lists, the empty list, and the list constructor. The last two are the two basic axioms about lists.

```
list: pred -> type.
nil: list X.
cons: ptm X -> list X -> list X.
append: list X -> list X -> list X.

append_nil: append nil X X.
append_cons: (append (cons X XS) YS (cons X ZS)) <- (append XS YS ZS).
```

For example, `(cons (owns a d) nil)` is a list containing the policy $owns(a, d)$. Let us prove a simple proposition about a list.

Example 9 (Joining two lists) *The list `(cons p (cons q nil))` is the same as appending the lists `(cons p nil)` and `(cons q nil)`.*

A proof is `(append_cons append_nil)`, for example. We can verify the proof by declaring a simple lemma:

```
p: ptm policy.
q: ptm policy.
```

```
lem1:
(append (cons p nil) (cons q nil) (cons p (cons q nil)))
                                        = append_cons append_nil.
```

If one type-checks this, Twelf accepts this as a lemma, and stores it under the name `lem1`*. In the process the Twelf type-checker checked that the type of the proof term (after the = sign) and the type of the proposition (before the = sign) are the same.*

The propositions we are ultimately interested in are of the form $\Gamma_1; \Gamma_2; \Delta \vdash_a \phi$. Agents must provide auditors with justification proofs, which are proofs of this kind of propositions. The entailment relation $\vdash$ is declared as follows.

```
entail: otm agent -> list policy -> list action -> list action ->
                                              ptm policy -> type.
```

Now we can define the types for the proof rules. We start with the basic first-order logic rules.

```
true_r:   entail A Gamma1 Gamma2 Delta true.
init:     entail A (cons Phi Gamma1) Gammact2 Delta Phi.
cut:     entail A Gamma1 Gammact2 Delta Phi ->
         entail A (cons Phi Gamma1) Gamma2 Delta' Chi ->
         append Delta Delta' Delta'' ->
          entail A Gamma1 Gamma2 Delta'' Chi.
imp_l:   entail A Gamma1 Gamma2 Delta Phi ->
          entail A (cons Chi Gamma1) Gamma2 Delta' Psi ->
          append Delta Delta' Delta'' ->
          entail A (cons (imp Phi Chi) Gamma1) Gamma2 Delta'' Psi.
imp_r:   entail A (cons Phi Gamma1) Gamma2 Delta Chi ->
          entail A Gamma1 Gamma2 Delta (imp Phi Chi).
and_l1:  entail A (cons Phi Gamma1) Gamma2 Delta Chi ->
          entail A (cons (and Phi Psi) Gamma1) Gamma2 Delta Chi.
and_l2:  entail A (cons Psi Gamma1) Gamma2 Delta Chi ->
          entail A (cons (and Phi Psi) Gamma1) Gamma2 Delta Chi.
and_r:   entail A Gamma1 Gamma2 Delta Phi ->
         entail A Gamma1 Gamma2 Delta' Chi ->
         append Delta Delta' Delta'' ->
          entail A Gamma1 Gamma2 Delta'' (and Phi Chi).
forall_l:entail A (cons (Phi X) Gamma1) Gamma2 Delta Chi ->
          entail A (cons (forall Phi) Gamma1) Gamma2 Delta Chi.
forall_r:entail A Gamma1 Gamma2 Delta (Phi X) ->
          entail A Gamma1 Gamma2 Delta (forall Phi).
!imp_l:  entail A (cons Phi Gamma1) Gamma2 Delta Psi ->
          entail A (cons (!imp Act Phi) Gamma1) Gamma2
          (cons Act Delta) Psi.
!imp_r:  entail A Gamma1 Gamma2 (cons Act Delta) Psi ->
          entail A Gamma1 Gamma2 Delta (!imp Act Psi).
?imp_l:  entail A (cons Phi Gamma1) Gamma2 Delta Psi ->
```

```
          entail A (cons (?imp Act Phi) Gamma1)
          (cons Act Gamma2) Delta Psi.
?imp_r:  entail A Gamma1 (cons Act Gamma2) Delta Psi ->
          entail A Gamma1 Gamma2 Delta (!imp Act Psi).
```

The structural rules for contraction and permutation of formulae in the sequents, are defined as follows. We write `perm_l1`, `perm_l2` and `perm_l3` for the three permutation rules, and `contr_l1`, `contr_l2` for the contraction of the two non-linear contexts.

```
perm_l1: entail A Gamma1 Gamma2 Delta Phi ->
   append L1 (cons X1 (cons X2 L2)) Gamma1 ->
   append L1 (cons X2 (cons X1 L2)) Gamma1'->
   entail A Gamma1' Gamma2 Delta Phi.
perm_l2: entail A Gamma1 Gamma2 Delta Phi
   append L1 (cons X1 (cons X2 L2)) Gamma2 ->
   append L1 (cons X2 (cons X1 L2)) Gamma2' ->
   entail A Gamma1 Gamma2' Delta Phi.
perm_l3: entail A Gamma1 Gamma2 Delta Phi
   append L1 (cons X1 (cons X2 L2)) Delta ->
   append L1 (cons X2 (cons X1 L2)) Delta' ->
   entail A Gamma1 Gamma2 Delta' Phi.
contr_l1: entail A (cons Phi (cons Phi Gamma1)) Gamma2 Delta Psi ->
   entail A (cons Phi Gamma1) Gamma2 Delta Psi.
contr_l2: entail A Gamma1 (cons Phi (cons Phi Gamma2)) Delta Psi ->
   entail A Gamma1 (cons Phi Gamma2) Delta Psi.
```

We conclude by formalizing the proof rules obs-act, owns-L, refine and owns-maysay.

```
obs_act: concl A Act Phi ->
         entail A (cons Phi Gamma1) Gamma2 Delta Psi ->
         entail A Gamma1 (cons Act Gamma2) Delta Psi.

refine:  entail A G nil nil Psi ->
         map G ([x] (maysay B C x)) Gs ->
         append Gs W Gamma1 ->
         entail A Gamma1 Gamma2 Delta (maysay B C Psi).

owns_l: (data_aff Phi Data) ->
         map_o Data ([d] (owns A d)) Go ->
         append Go W Gamma1 ->
         entail A Gamma1 Gamma2 Delta Phi.

owns_maysay:
        entail A (cons (maysay B C (owns A D)) Gamma) Gamma2 Delta
                                             (maysay B C Psi)  ->
         entail A (cons (owns A D) Gamma) Gamma2 Delta
                                             (maysay B C Psi).
```

The rule obs-act uses the conclusion derivation function **concl**(). The type of the conclusion derivation is defined by:

```
concl: tm action -> tm agent -> tm policy -> type.
```

Proofs of (`concl A Act Phi`) are either `concl_comm`, or `concl_creates`, corresponding to the conclusions from communication and creation, respectively.

```
concl_comm: concl (comm A B Phi) B Phi.
concl_creates: concl (creates A D) A (owns A D).
```

The refine rule requires the map function, which is defined by:

```
map: list T -> (ptm T -> ptm T1) -> list T1 -> type.
map_nil: map nil F nil.
map_cons: map B F L -> map (cons A B) F (cons (F A) L).
```

The owns-rule makes use of the function **data_aff**. As mentioned in Section 2.4, we can restrict **data_aff**(ϕ) to atomic policies.

```
data_aff: ptm policy -> list_o data -> type.
data_aff_owns: data_aff (owns B D) (cons_o D nil_o).
```

For each additional atomic predicate needed in certain setting, for example *mayRead*(), one additional definition is needed to define which data it affects. In the definition of **data_aff** `list_o` is a type for lists of data, which is identical to the definition of `list`, except for the `_o` subfix and the type of the list elements.

3.2.2 Scenario-specific code

For a given scenario the Twelf signature is extended with scenario-specific predicates and actions. The extension consists of short and simple type definitions, that do not affect the general properties of the logic. Consider the example scenario described in the end of Chapter 2. The scenario-specific definitions are as follows.

```
alice : otm agent.
bob : otm agent.
file : otm data.
reads : otm agent -> otm data -> ptm action.
writes : otm agent -> otm data -> ptm action.
mayread : otm agent -> otm data -> ptm policy.
maywrite : otm agent -> otm agent -> ptm policy.
isusingv4: otm agent -> ptm policy.

data_aff_mayread: data_aff (mayread B D) (cons_o D nil_o).
data_aff_maywrite: data_aff (maywrite B D) (cons_o D nil_o).
```

Example 10 (Dynamic collaborative environment) *We can now check a basic proof. Suppose Alice creates a file* `file` *and she gives Bob a policy that states that Bob can read it, if he is using version 4 of the reader software* $isUsingV4(Bob) \rightarrow mayRead(Bob, File)$. *Bob reads the file, and logs this action together with the condition that he is using version 4 of the document reader* $isUsingV4(Bob)$. *The auditor finds out about Bob reading the file, and asks Bob to account for reading the file. Bob uses his log entry of Alice's communication to prove* $mayRead(Bob, File)$.

```
thm2:  (entail bob
             (cons (isusingv4 bob) nil)
             (cons (comm alice bob (imp (isusingv4 bob)
                                         (mayread bob file))) nil)
             nil
             (mayread bob file))=
(obs_act concl_comm (imp_l init init append_nil)).
```

The proof (right of the equality sign) type-checks with the proposition. Bob has accounted for his action. The auditor has now found evidence of Alice's policy communication. The auditor asks Alice to account for it.

```
lemma2:
  (entail alice
    nil
    (cons (creates alice file) nil)
    nil
     (maysay alice bob (imp (isusingv4 bob) (mayread bob file))))=
 (obs_act concl_creates (owns_maysay (refine (imp_r (perm_g1_2
                                           (owns_l data_aff_mayread
 (map_cons_o map_nil_o) (append_cons append_nil)))) (map_cons map_nil)
                                           (append_cons append_nil)))).
```

The proof of both lemma's were output by the proof finder (see the next section), and checked by Twelf.

Proofs that use many conditions and actions as evidence, require numerous permutations. In our implementation of the proof checker, for the sake of readability, we allow abbreviation of permutation steps using lemma's. For example, `perm_g1_2` (used in the previous example, swaps the first and the second element in the first context, and `perm_g2_5` swaps the first and the fifth element in the second context, and so forth.

3.3 Proof Finding

In the previous section we have proposed a proof checker for the AC^2 proof system, in this section we address proof finding in the AC^2 proof system. In this section we show that the cut-rule is redundant in the AC^2 proof system, which implies consistency, and semi-decidability. The cut-free calculus is also more suitable for mechanical (bottom-up) proof search. In Section 3.3.2 we show an implementation of a proof finder for the AC^2 proof system, written in Prolog.

3.3.1 Cut-elimination

In this section we prove a cut-elimination theorem for the AC^2 proof system, which states that: If we can proof some lemma ϕ and prove a formula ψ using this lemma, then the formula ψ can also be proven directly. Not having this intuitive property would indicate an exotic logical system indeed. Cut-elimination theorems, due to Gentzen [31], are considered a central issue in the field of logics. A cut-elimination theorem exists for first-order logic as well as for a number of other standard logical systems. In our case consistency and semi-decidability follow from cut-elimination.

Below for the sake of readability, we only write the non-linear context Γ, as the other two contexts are irrelevant for this proof. For the same reason, in the sequel we ignore the left and right rules for $\xrightarrow{?}$ and $\xrightarrow{!}$.

The cut-rule is as follows.

$$\frac{\Gamma \vdash_a \phi \qquad \Gamma, \phi \vdash_a \psi}{\Gamma \vdash_a \psi}\ \text{cut}$$

Here ϕ is called the cut-formula. It is is cut out of the premises.

The cut-rule does not satisfy the *sub-formula* property; the cut-formula, ϕ, in the premise may be completely absent in the conclusion. A mechanical proof finder would have to guess it. The sub-formula property is important to be able to implement an efficient proof search.

Theorem 2 (Cut-elimination) *Let the proof system with the cut-rule be denoted with $\vdash_a$ and the proof system without the cut-rule be denoted $\vdash_a^+$, then, for arbitrary Γ and ϕ,*

$$\Gamma \vdash_a \phi \quad \Rightarrow \quad \Gamma \vdash_a^+ \phi \tag{3.1}$$

Proof 2 *For proving the cut elimination theorem for our logic we follow a standard approach [72]: We show by induction (over the length of the formula to be proven) that proofs including a cut rule can be transformed into proofs without this rule.*

Our induction assumption states that, if we have a cut free proof $\mathcal{D}$ for $\Gamma \vdash_a \phi$ and a cut free proof $\mathcal{E}$ for $\Gamma, \phi \vdash_a \psi$ then we also have a cut free proof $\mathcal{F}$ for $\Gamma \vdash_a \psi$. This induction assumption is applied if the cut formula (ϕ) is simplified or if the cut formula stays the same and one of the proofs is shortened (and the other proof is not lengthened).

Below, a formula is called principal *in the rule, if the rule explicitly introduces the formula (either left or right of the $\vdash_a$). Furthermore, we* *will use the fact that any proof for $\Gamma \vdash_a \phi$ can be* weakened *to a proof for $\Gamma, \psi \vdash_a \phi$ by using the same rules but simply adding ψ in each step.*

The proof is by case-analysis over the last rule used in the proofs $\mathcal{D}$ and $\mathcal{E}$. For clarity we show a table with the cases for $\mathcal{D}$ and $\mathcal{E}$. The table refers to parts of the proof below, and pr. denotes principal.

	$\mathcal{D}$ *init(I)*	$\mathcal{D}$ *owns-L*	ϕ *not pr. in* $\mathcal{D}$	ϕ *pr. in* $\mathcal{D}$
$\mathcal{E}$ *init(I)*	*1*	*2*	*2*	*2*
$\mathcal{E}$ *owns-L*	*1*	*3*	*5*	*4*
ϕ *not pr. in* $\mathcal{E}$	*1*	*6*	*5*	*6*
ϕ *pr. in* $\mathcal{E}$	*1*	*3*	*5*	*7*

The rules init(I) and owns-L are the base-cases of the induction over the length of the derivation, so they are done first. As mentioned we leave out the rules concerning use once and use many obligations, $\overset{!}{\hookrightarrow} R$, $\overset{!}{\hookrightarrow} L$, $\overset{?}{\hookrightarrow} R$ and $\overset{?}{\hookrightarrow} L$, as well as the $\top R$ rule, which amount to trivial cases below.

1. $\mathcal{D}$ ***ends in I.*** *When $\mathcal{D}$ consist of a single init rule,*

$$\mathcal{D} : \frac{}{\Gamma', \phi \vdash_a \phi}\ I$$

(i.e. $\Gamma = \Gamma', \phi$) then applying contraction to $\Gamma', \phi, \phi \vdash_a \psi$, which is the conclusion of $\mathcal{E}$, gives us the required sequence $\Gamma', \phi \vdash_a \psi$. Thus $\mathcal{E}$ followed by contraction is a cut-free proof for this sequent.

2. $\mathcal{E}$ ***ends in I.*** *When $\mathcal{E}$ consists of a single init(I) rule and ϕ is used,*

$$\mathcal{E} : \frac{}{\Gamma, \psi \vdash_a \psi}\ I$$

then the cut-formula is ψ and a cut-free derivation of ψ is simply $\mathcal{D}$. Otherwise, if ϕ is not used,

$$\mathcal{E} : \frac{}{\Gamma', \psi, \phi \vdash_a \psi}\ I$$

(i.e. $\Gamma = \Gamma', \psi$), then a cut-free proof for the required sequent $\Gamma', \psi \vdash_a \psi$ is a single application of the init rule.

3. $\mathcal{D}$ ***ends in owns-L.*** *When $\mathcal{D}$ consists of a single application of the owns-L rule, then ϕ is atomic,*

$$\mathcal{D} : \frac{}{\Gamma', owns(a,d) \vdash_a \phi}\ owns\text{-}L$$

so if ϕ is principal in the last inference in $\mathcal{E}$ then this inference must use rule init, covered in 2, or owns-maysay or owns-L. In the latter two cases, one can simply contract the context to obtain the required sequent. In case ϕ is not principal in $\mathcal{E}$'s last step, then the induction assumption for a smaller proof $\mathcal{E}$ is used, see case 6.

4. ***$\mathcal{E}$* ends in owns-L.** *When $\mathcal{E}$ is owns-L and ϕ is used, then ϕ is an owns() predicate.*

$$\mathcal{E} : \frac{}{\Gamma, \mathit{owns}(a, d) \vdash_a \psi} \; \mathit{owns\text{-}L}$$

There are no cases for ϕ principal in $\mathcal{D}$'s last step except init and owns-L, both treated in the cases 1 and 3. In case ϕ is not principal in $\mathcal{D}$'s last step, then the induction assumption for a smaller proof $\mathcal{D}$ is used, see case 5. Otherwise if ϕ is not used in owns-L,

$$\mathcal{E} : \frac{}{\Gamma, \phi, \mathit{owns}(a, d) \vdash_a \psi} \; \mathit{owns\text{-}L},$$

then a cut-free derivation of ψ is a single application of the owns-L rule.

5. ***ϕ* is not principal in *$\mathcal{D}$*.** *The cut-formula is not principal in the derivation $\mathcal{D}$ if the derivation ends in one of the (left) rules: $\rightarrow$L, $\forall$L, $\wedge L_1$, $\wedge L_2$, obs-act, owns-maysay.*

   All the cases for the different left rules are similar. As an example we show the case for $\wedge L_1$.

   If the proof $\mathcal{D}$ consists of proof $\mathcal{D}_1$ followed by $\wedge L1$:

$$\mathcal{D} : \frac{\Gamma', \phi_1 \vdash_a \phi}{\Gamma', \phi_1 \wedge \phi_2 \vdash_a \psi} \; \wedge L_1 \qquad \mathcal{E} : \Gamma', \phi_1 \wedge \phi_2, \psi \vdash_a \psi'$$

   then by weakening $\mathcal{D}_1$ with $\phi_1 \wedge \phi_2$ and weakening $\mathcal{E}$ with ϕ_1 one get proofs for $\Gamma', \phi_1, \phi_1 \wedge \phi_2 \vdash_a \psi$ and $\Gamma', \phi_1, \phi_1 \wedge \phi_2, \psi \vdash_a \psi'$ thus by induction (the weakened $\mathcal{D}_1$ is shorter than $\mathcal{D}$ and the weakened $\mathcal{E}$ is the same length as $\mathcal{E}$), there is a cut-free proof for $\Gamma', \phi_1, \phi_1 \wedge \phi_2 \vdash_a \psi'$. By applying $\wedge - L1$ and then contraction one derives the required sequent $\Gamma', \phi_1 \wedge \phi_2 \vdash_a \psi'$.

   The cases for the other left rules are done in the same way.

6. ***ϕ* is not principal in *$\mathcal{E}$*.** *One can apply the induction assumption on the $\mathcal{D}$ and $\mathcal{E}_1$, to obtain a cut free proof $\mathcal{F}_1$ and then apply the same right rule as the righthand side of the sequents proven by $\mathcal{E}_1$ and $\mathcal{F}_1$ are the same.*

7. ***ϕ* is principal in both *$\mathcal{D}$* and *$\mathcal{E}$*.** *This is the most elaborate case. We must split cases for the different forms of the cut-formula and use the induction assumption for a sub-formula of the cut-formula.*

   (a) ***Subcase* $\phi = \phi_1 \rightarrow \phi_2$.** *There is one case for the last inference of $\mathcal{D}$:*

$$\mathcal{D} : \frac{\Gamma, \phi_1 \vdash_a \phi_2}{\Gamma \vdash_a (\phi_1 \rightarrow \phi_2)} \rightarrow L$$

and $\mathcal{E}$'s last inference must be →L:

$$\mathcal{E} : \frac{\Gamma \vdash_a \phi_1 \qquad \Gamma, \phi_2 \vdash_a \psi}{\Gamma, (\phi_1 \to \phi_2) \vdash_a \psi} \to L$$

One can apply the induction assumption on the premise in $\mathcal{D}$ and the first premise in $\mathcal{E}$ to obtain a cut-free proof for $\Gamma \vdash_a \phi_2$ and again use the induction assumption on this proof and the second premise in $\mathcal{E}$ to obtain a cut-free proof the required sequent. (Both cases use a simpler cut formula.) The cases for ϕ with the connectives $\wedge$ and $\forall$ are done in the same way.

(b) ***Subcase** $\phi = maySay(b, c, \phi_1)$**.*** *There is one case for the last inference in $\mathcal{D}$:*

$$\mathcal{D} : \frac{\Gamma' \vdash_a \phi_1}{\Gamma'', maySay(b, c, \Gamma') \vdash_a maySay(b, c, \phi_1)} \; refine$$

Then $\mathcal{E}$'s last inference must be refine:

$$\mathcal{E} : \frac{\Gamma', \phi_1 \vdash_a \psi_1}{\Gamma'', maySay(b, c, \Gamma'), maySay(b, c, \phi_1) \vdash_a maySay(b, c, \psi_1)} \; refine$$

then the induction assumption can be applied for the proofs $\mathcal{D}_1$ and $\mathcal{E}_1$ of the premises to reach the required sequent without the use of either refine-rule.

(c) ***Subcase** ϕ **is atomic.*** *There are two cases for the last step in $\mathcal{D}$ (where ϕ is principal), being init and owns-L, which were treated in the cases 1 and 3.*

This completes the proof.

Cut-elimination implies both consistency and semi-decidability. In sequent calculus consistency is shown by proving that *falsity* can not be derived: $\nu \vdash \bot$. Now the AC^2 policy language does not have a separate predicate for falsity, but in AC^2 $\forall_d\ owns(a, d)$ behaves much like falsity. We show that it can not be derived.

Corollary 1 (Consistency) *For any $a \in \mathbf{AG}$, $\nu \vdash_a \forall_d\ owns(a, d)$ cannot be derived.*

The proof is as follows. Cut-elimination states that for every derivation in $\vdash_a$ there is also a derivation of the same proposition in the cut-free calculus $\vdash_a^+$. Suppose there is a derivation of $\forall_d\ owns(a, d)$ in $\vdash_a$, then there must be one also in $\vdash_a^+$. However there is no right rule for introducing *owns*(), hence $\vdash_a$ is consistent.

Corollary 2 (Semi-decidability) *Given a proposition* Γ*, a, and* ϕ*, provability of* $\Gamma \vdash_a \phi$ *is semi-decidable.*

To show that provability of $\Gamma \vdash_a \phi$ is semi-decidable we outline a procedure, that finds a proof of $\Gamma \vdash_a \phi$, if $\Gamma \vdash_a \phi$ holds.

First of all, cut-elimination implies that we can restrict proof search to the cut-free sequent calculus $\vdash_a^+$. The proof rules in $\vdash_a^+$ satisfy the subformula property in the sense that every derivation (of $\Gamma \vdash_a^+ \phi$ only contains subformulas of the formulas in Γ or ϕ. This is the main reason why bottom-up proof search is efficient in the sequent calculus [71]. Except for the contraction rule, in each rule the premises contain structurally smaller formulas than those in the conclusion. Hence without contraction proof search would terminate, but of course, without contraction, it would not be complete. The contraction rule is dealt with separately as follows.

The first step of the procedure consists of searching for proofs in $\vdash_a^+$ that use the contraction rule at most n times. This step terminates. If a proof is found, then $\Gamma \vdash_a \phi$ is provable. Otherwise the procedure continues with a new proof search for proofs in $\vdash^+$ that uses contraction $n + 1$ times.

The procedure hence terminates, and decides that $\Gamma \vdash_a \phi$ holds if it holds. But the procedure never decides that the proposition is not provable though, if it is not. In the next section we show an implementation of this procedure using Prolog.

3.3.2 Prolog code

To find justification proofs in the AC^2 framework we have implemented a proof finder in Prolog, which we call PFF (Prolog Proof Finder). PPF finds (automatically) an AC^2 justification proof, from a given log, for a given action, *if such a proof exists.* If no proof exists, PPF may not terminate. PPF can be used by agents during an audit, when a justification proof must be presented to an auditor (see Figure 6). Alternatively, PPF can be used before performing an action, if the performing agent is unsure about whether or not the action is allowed.

PPF is written in SWI Prolog (a public and ISO compliant Prolog system). PPF was designed to output proof terms that can be checked directly by TPF, the proof checker written in Twelf (see Section 3.2). This is an important design choice. It makes the proof finding procedure more complicated, yet on the other hand this allows us to skip the verification of the proof finder code. Instead of proving that the proof finder works correctly, we simply verify its output by using the Twelf proof checker.

PPF is not a state-of-the-art theorem prover, but only a proof of concept. PPF does not implement the rules for the linear context Δ for example. We discuss possible improvements and alternatives in the conclusions of this chapter.

Let us introduce the Prolog code of PPF:

- Predicates, connectives and policies of the AC^2 language, are represented as (nested) lists. For example, the predicate $\mathit{mayWrite}(a, d) \wedge \mathit{mayRead}(a, d)$ is written as a nested list

  ```
  [and, [maywrite, a, d], [mayread, a, d]].
  ```

- Entailment ($\vdash$) is represented by the Prolog predicate `entail\5`. Finding a proof of $\Gamma \vdash_a \phi$ is thus reduced to solving the Prolog goal, `entail(a, gamma, phi, proof)`. The last argument is output: `proof` is the proof term, a string that is a proof of $\Gamma \vdash_a \phi$, which can be checked directly by our proof checker.

- The proof rules of the proof system of AC^2 are implemented by logic programming clauses, where the conclusion of the proof rule forms the head of the clause, and the premises of the proof rule form the body of the clause. Let us give a simplified example, by showing the $\wedge L_1$ rule in formal notation and its translation into Prolog. Here we ignore the additional contexts to illustrate the basic idea. The rule $\wedge \mathrm{L}_1$,

  $$\frac{\Gamma, \phi_1 \vdash_A \psi}{\Gamma, (\phi_1 \wedge \phi_2) \vdash_A \psi}\ \wedge\mathrm{L}_1$$

  is translated to:

  ```
  entail(A,[[and,Phi1,_] | Gamma1,Psi,[' (and_l1',Proof,')']):-
     entail(A,[Phi1|Tail],Psi,Proof).
  ```

  Now, PPF works as follows: Suppose we want to find a proof of the expression $\Gamma \vdash_{\mathrm{a}} \phi$, then we ask Prolog to solve the query:

  ```
  entail(A, Gamma, Phi, Proof)
  ```

  All variables are bound to values, except `Proof`, so if Prolog resolves the query, it will come up with a value for it, which is a proof term. Prolog resolution thus amounts to bottom-up proof search for $\Gamma \vdash_A \Phi$ in the AC^2 proof system.

- Translating the rules for universal quantification in Prolog is complicated. Prolog variables can not be used directly, because we want to be able to control variable renaming, and separate true variable from meta-variables. In PPF we implement a unification procedure based on Paulson's article on first-order logic theorem proving [69]: Quantified expressions are written as `[foral,x,[mayRead, x, d]`, without using Prolog's built-in variables.

  In the $\forall L$ rule the bound variable y in $\phi(y)$ can be replaced by any value x, but because we do bottom-up proof search - to avoid having to guess a good value for x - we postpone this choice until we arrive at the top rules (such as init (I) and owns-L). We unify variables in the top-rules (I, and $owns(-)L$) and also in the refine rule, where we unify agent variables in $maySay()$.

We only implement unification for atomic formulas which is sufficient because we can always decompose formulas left and right of the $\vdash$ to atomic formulas (with left and right rules, or refine in case of *maySay*).

In the $\forall R$ rule, x must be a fresh value, that was not used before. Because, as mentioned above, we postpone unification in the $\forall - L$ rule (see above), fresh values depend on the meta-variables that are still to be unified. In PPF fresh values are terms of the form `param(name,[...])`, where the list is used to do an *occurs-check* in the unification predicate `unifiabl\3`.

- In the refine-rule,

$$\frac{\Gamma_1 \vdash_a \psi}{\Gamma_1', maySay(b, c, \Gamma_1) \vdash_a maySay(b, c, \psi)} \text{ refine,}$$

since we do bottom-up proof search, we must match sender-receiver variables of the *maySay* predicates on both sides of the $\vdash$, before proceeding. If there are meta-variables in these places (see above) we have to unify them here and proceed with the assigned values. The Prolog code of the refine rule is as follows.

```
entail(Agent,Gamma,[maysay,B,C,Psi],Proof,Env):-
   sublist_of_maysay_formulas(Gamma,GMaysay),
   length(GMaysay,N),
   maysayinverse(GMaysay,G,List1),
   pad1([B,C],N,List2),
   unifiabl(List1,List2,Env1),
   entail(Agent,G,Psi,Pf,Env2),
   append(Env1,Env2,Env),
   perm(GMaysay,_,Gamma,PfPerms,PfBras),
   mapmaysaypf(G,GMaysay,PfMapMaysay),
   concat_atom_f([PfPerms,
         ' (refine',Pf,PfMapMaysay,')',PfBras], Proof).
```

Basically, in the first subgoal we select a sublist of `Gamma`, `GMaysay`, that contains only formulas of the form `[maysay, ., ., .]`. The third, fourth and fifth subgoal take care of the unification step for agent variables b and c left and right of the $\vdash$. The sixth subgoal tries to close the branch by proving the premise of the refine rule, and the nineth subgoal produces a proof (term) of the proposition that the sublist `GMaysay` is a map, under *maySay*, of the list `G`. The last subgoal prints the proof term needed for proof checking.

- The owns-left rule,

$$\frac{\mathbf{data\_aff}(\phi) \subseteq \{d_1, \ldots, d_n\}}{\Gamma_1, \mathit{owns}(a, d_1), ..., \mathit{owns}(a, d_n) \vdash_a \phi}\ \text{owns-L},$$

  is again a top-rule. In PPF the rule is implemented as:

```
entail(Agent,Gamma,Psi,_,Proof,Env):-
   data_aff(Psi,Datalist,PfOmega),
   pad2(Agent,Datalist,List1),
   sublist_of_owns_formulas(Gamma,GOwns),
   ownsinverse(GOwns,List2),
   unifiabl(List1,List2,Env),
   mapownspf(Agent,Datalist,GOwns,PfMapowns),
   perm(GOwns,_,Gamma,PfPerms,PfBras),
   concat_atom_f([PfPerms,'
         (owns_l',PfAffData,PfMapowns,')',PfBras], Proof).
```

  The first subgoal checks if the *affected data* (**data_aff**) is defined for the formula `Psi`, and which data must be owned by agent A for the owns-rule to apply. The second, third and fourth subgoal select *owns* predicates from the context `Gamma` and check whether these *owns* predicates can unify in the right way. The fifth subgoal prints a proof term that proves that the sublist of `Gamma` containing the *owns* predicates is indeed a sublist of `Gamma`. The final subgoal concatenates the full proof term for this branch.

For reasons of efficiency we do not directly translate the permutation rules in Prolog. The permutation rule for the first context is written as follows.

$$\frac{\Gamma_1, \phi_1, \phi_2, \Gamma_1'; \Gamma_2; \Delta \vdash_a \psi}{\Gamma_1, \phi_2, \phi_1, \Gamma_1'; \Gamma_2; \Delta \vdash_a \psi}\ \text{P-L}_1$$

Permutation allows the use of left-rules or top-rules, by positioning elements at the first position of the list of hypotheses. In PPF we replace the permutation rules by a predicate `perm\5` in every left (L) and top-rule. The predicate `perm\5` that checks membership of a formula in a list (using `member`), and determines the permutations needed to place the formula to the front of the list. (In addition it returns the newly ordered list where the required element is at the first place, as well as a proof term that describes the use of the permutation rule.) This predicate can be seen in the $\wedge L_1$ rule for example:

```
entail(Agent,Gamma,Psi,Proof):-
   perm([[and,Phi1,_]],Tail,Gamma,PfPerms,PfBras),
   entail(Agent,[Phi1|Tail],Psi,Pf),
   concat_atom_f([PfPerms,' (and_l1',Pf,')',PfBras], Proof).
```

This implementation of permutation is complete, since permutations are transitive, and because permutations commute with the right (R) rules. The predicate `perm`) actually output abbreviated proof terms, by using permutation lemma's.

3.3.3 Proof search

Prolog's resolution procedure is *depth-first*, which means that, given a proposition, it will try to prove it by following the first (leftmost) branch. Prolog does not move to the next branch unless it reaches a failure point, i.e. a point in which the selected atom does not unify with the head of any clause. If a branch is infinite, then Prolog does not move to the next branch. The problem here is that if the proof is not found in the infinite branch, Prolog does not explore other perhaps more fruitful branches.

Proof rules that do not have the so-called *sub-formula property* cause infinite branches. Bottom-up proof search only works if all the rules have the sub-formula property. There are two proof rules that are problematic from this point of view: the cut-rule, and the contraction rule.

- The cut-rule would be translated in Prolog as follows:

  ```
  entail(A,Gamma,Psi,Proof):-
                  entail(A,Gamma,Phi,PfC1),
                  entail(A,[Phi|Gamma],Psi,PfC2),
                  concat_atom_f([' (cut',PfC1,PfC2,')'], Proof).
  ```

  For any `entail(Agent,Gamma,Psi,Proof)` the cut-rule can be used, any number of times. There are infinitely many values for `Phi` for which the subgoal `entail(A,Gamma,Phi,PfC1)` can be solved, because the formula `Phi` does not occur anywhere in the conclusions of the proof rule (in the head of the Prolog clause). Prolog will try all the possible values (infinitely many). We have showed, by the cut-elimination theorem, that the cut-rule is redundant, so we do not have to encode the cut-rule in PPF.

- The contraction rule would be translated in Prolog as follows:

  ```
  entail(Agent, [Phi|Gamma], Psi, Proof):-
                  entail(Agent,[Phi|[Phi|Gamma]], Psi, Pf).
                  pfappend([Phi],[Phi|Gamma],Psi,Pf2),
                  concat_atom_f([' (c_l1',Pf,Pf2,')'],Proof).
  ```

  Again, as with the cut-rule, the contraction rule always applies, and it causes each branch to become infinite. Now, in propositional logic it can be shown that after applying contraction three times for each formula in Γ, no new proofs are found. This is a way of showing that *propositional logic* is decidable. In predicate logic, because of the $\forall L$

rule, such a bound on the use of contraction does not exist. Therefore, to avoid the infinite branches caused by the contraction rule, in PPF we control its use by limiting the number of times it can be used in a given branch. We increase this limit only when no proof is found. The limit is implemented by adding an integer variable to `entail`, and decrease it each time the contraction rule is applied.

```
entail(Agent,[Phi|Gamma],Psi,Lim,Proof):-
   Lim > 0, LimNew is Lim - 1,
   entail(Agent, [Phi|[Phi|Gamma]], Psi, LimNew, Pf),
   concat_atom_f([PfPerms, ' (c_l', Pf,
         '(append_cons append_nil)', ')', PfBras],Proof).
```

Restricting the contraction rule in this way forces a *breadth-first* search (across the branches that use contraction `Lim` times). Thus ensuring a complete proof search, that is, if there is a proof of a proposition, then it will be found by PFF.

3.4 Related Work

Closely related to our prototype is the BLF system [90], which uses a policy language based on Binder [30] (a predicate Horn logic). BLF focusses on providing a framework for checking semantic properties of software code (proof-carrying code). In BLF proofs about the correctness of software are generated using Prolog, and checked using Twelf (like in the AC^2 framework). The Binder language used in BLF is a subset of the AC^2 language. In particular it is restricted to Horn clauses, and hence common access control policies such as $\forall x.isdoctor(x) \rightarrow read(x, file) \wedge write(x, file)$ can not be expressed in BLF. Proof finding in Horn clauses is more easy, and decidable. Furthermore, although Binder features a *says* predicate, related to our *maySay* predicate, it does not allow nested delegation, for the sake of tractability. The latter is a severe restriction, in settings with chains of delegations such as those occurring in dynamic collaborative environments where users delegate rights to other users, who in turn delegate to other users. We have shown that in our logic, nesting of the *maySay* predicate does not yield intractabilities.

PCA is a system to check access requests of web clients for protected resources on web servers. The proof checker of PCA is implemented in Twelf, while a tactical proof finder is implemented in Java. Manual proof finding is necessary because, unlike AC^2, the policy language of PCA includes higher-order expressions. This feature provides greater flexibility than in AC^2. For instance, any of the proof rules in the AC^2 proof system can be expressed and issued by a user as a normal policy. On the other hand, proof finding becomes intractable, and properties of the proof system, such as consistency, can not be proven once and for all, but they depend on which policies are issued by the users, at runtime. The PCA prototype, implemented in Twelf, introduces a type 'world' which is similar to the logical contexts (Γ) that usually occur in sequent calculi. Finally, the PCA proof system is a classical logic, while in AC^2 we use an intuitionistic proof system, to avoid certain indirect justification proofs (that use the rule of *excluded middle*).

SD3 is a trust management system that uses, like AC^2, separate tools for finding and checking proofs [51]. Like in AC^2, the safety of the SD3 system does not depend on the correctness of the proof finding algorithm, but only on the proof checker. SD3 uses Prolog however for proof checking, while we use Twelf's type checker for that purpose. The proof finder for SD3 (referred to as 'certified evaluator') is straightforwardly written in Prolog, since SD3 uses only Datalog expressions. Like Binder, Datalog can not express some common policies (see above).

Our implementation in Twelf is based on the LF signature for intuitionistic logic introduced by Harper [41]. We refine his approach by distinguishing also the subtypes for agents and data, and those for actions and policies. Harper's LF signature implements a first-order logic proof checker using natural deduction. As mentioned earlier, natural deduction is not directly suitble for mechanical proof finding, as opposed to the sequent calculus that we use. Moreover, the intuitionistic sequent calculus we use is explicit about the assumptions used in the proofs, which makes the use of the agent's log entries explicit in the propositions.

PPF is inspired by Folderol, a prototype proof finder written by Paulson in ML [69]. Paulson mentions that Prolog is indeed the natural choice for a mechanical theorem prover, but, he chooses ML to avoid the technicalities involved with fixing Prolog's unsound unification procedure. Paulson does refer repeatedly to a Prolog version of the Folderol theorem prover, but this version was never published (nor could Paulson provide us the code). PPF extends Folderol in the sense that PPF apart from finding the proofs, also outputs the syntax of the proof itself. Instead of merely checking whether a proposition holds or not. PPF generates proofs that can be checked with the Twelf implementation of the corresponding proof system.

3.5 Conclusions

We have developed two tools (TPC, and PPF) that implement the AC^2 framework. Our proof checker TPC is implemented in Twelf, while our proof finder (PPF) is implemented in Prolog. We have also proven that a cut-elimination theorem holds for the AC^2 proof system. Cut-elimination is an important theorem for showing consistency, and for allowing mechanical proof search. PFF is based on the cut-elimination theorem for the AC^2 proof system.

Twelf is a convenient tool for experimenting with a logic. Twelf's built-in type checker refuses to accept conflicting or ambiguous type declarations. At the same time, Twelf is flexible enough to implement even higher-order logics. The implementation of the policy language and proof system of AC^2 consists of under a hundred lines of Twelf code. The safety of the audit process (which relies on rejecting false proofs) depends only on the Twelf signature, which is short and straihtforward to verify.

Prolog is a natural choice when implementing a theorem prover. Our prototype theorem prover is a proof-of-concept and not a state-of-the-art theorem prover, and we do not focus on performance or efficiency issues. The fact that Prolog's own unification algorithm is unsound (which allows it to be more fast), complicates proof finding in a predicate logic like ours. Alternatively, instead of using a full-fledged theorem prover, agents could also use standard templates for proofs of common permissions from common actions. A second possibility may be to use lean theorem proving [19], which is fast at finding proofs for simple propositions, but slower for more complex propositions. For future research it would be interesting to see whether the lean theorem prover iLeanTap for intuitionistic logic can be converted to proofs checkable by Twelf's typechecker, and be extended with the AC^2 rules owns-L and refine. More in general we believe that lean theorem proving is an interesting future possibility for theorem proving in access control logics.

Chapter 4

Electronic Health Records

4.1 Introduction

Traditional access control mechanisms aim to prevent illegal actions a-priori occurrence, i.e. before granting a request for a document. There are scenarios however where the security decision can not be made on the fly. For these settings we developed AC^2 a policy language and a framework for *a-posteriori access control*. In Chapter 2 we have shown that AC^2 provides a flexible system for use in a dynamic collaborative environment in a consultancy firm, where consultants create and exchange confidential documents. In this chapter we focus on the setting of an Electronic Health Record (EHR) system in a hospital.

Roughly speaking, Electronic Health Record (EHR) systems must fulfill two requirements [85]: (1) To provide high-quality health care, the EHR must be immediately available, preferably across the boundaries of the different hospitals and abroad. (2) To protect the patient's privacy, the EHR must remain confidential and should be disclosed only according to the law and/or the patient's explicit consent. The first requirement is crucial to improve the quality of health care. Medical errors are often caused by lack of information, or erroneous information and paper-based processing is slow, and error prone. The second requirement is important to protect the privacy of patients, and medical staff, especially in countries such as The Netherlands, where medical insurances are privatized and medical records have become valuable business information.

Fulfilling both requirements is hard. To fulfill the first requirement, the mechanism should be relatively simple and fast. The second requirement however states that access should only be granted under precise conditions and circumstances. Considering the complexity of the medical work flow, the large number of health records and the variety of institutions, users and systems involved, checking these circumstances and conditions is not simple or fast.

Besides the problem of designing a fast access control mechanism that at the same time takes into account complex conditions [9, 18, 40], it is considered impossible, in general, to design an access control mechanism that models *every* circumstance perfectly [74]; in other words, there will always be exceptional, unforeseen circumstances. This is an important issue in the EHR setting, given the mobility of patients and staff, and the urgency of health care. We believe that at least it should be possible for medical staff to self-authorize exceptions to rules, while leaving the process of justifying the exceptions for later.

In this chapter we argue that many of the legal requirements for EHR systems are addressed directly in AC^2, and we argue that AC^2 is suitable for use in an EHR system. We illustrate our case by describing a simple scenario involving medical personnel and health records to show how AC^2 can be used in a hospital.

4.2 Scenario

We describe a simple scenario involving a hospital's EHR system, the hospital's privacy officer, patients, and medical personnel.

4.2.1 General setting

The agents involved here are patients, doctors, nurses and administrative employees; the users of the EHR system. The data consists of the medical records and the actions we consider are reading and updating a medical record, and administering and billing drugs. Medical records are updated through *appending* a block of new information digitally signed by the agent that updates the record.

The form of the medical records we consider is inspired by the legal directives on privacy of health records [84, 85]. A medical record is divided into two parts: first, the personal information, PI, records all non-medical information related to the patient, like billing information, and information regarding the patient's family members (which may have medical records of their own). Second, the medical data (MD) gathers all the medical information of the patient, like diagnoses and given prescriptions.

Additionally, several auditors have the mandate of controlling that the medical records are used appropriately, i.e. the hospital's internal auditor, a governmental body and a patient union representative. Independently, these auditors may audit different sections of the organization.

The hospital has defined a general policy, ϕ_h, to protect the privacy of patients and allows medical personnel to access and handle the necessary health information. The policy is shown in Figure 8.

h1: Patients may read and update their record's PI section and authorize others to do so.

h2: Patients may read and update their record's MD section and authorize others to do so.

h3: Doctors may read the PI section of their patients health records.

h4: Doctors may read and update the MD section of their patients health records.

h5: Doctors may give drugs to their patients.

h6: Doctors can delegate to nurses on their staff the administering of drugs.

h7: Administrative employees may bill patients for every drug administered.

Figure 8: The hospital's policy.

We assume that whatever is not explicitly mentioned in the policy, is not permitted, and that each policy added entails *more* permissions (monotonicity). Additional policies may be added later to this general policy by the individual users, for example patient consent given to medical staff, or authorizations among medical staff. We will give examples in the sequel.

We assume users simply send policies to each other by using signed emails. We do not model explicit prohibitions, which would require special procedures for propagation in a distributed setting (See for example the distributed revocation mechanism of SPKI/SDSI [75]).

Let us introduce the scenario-specific AC^2 actions, and predicates needed. The actions in this example are `read`(), `update`(), `giveDrug`(), and `bill`(). The predicates *mayRead*(), *mayUpdate*(), *mayGiveDrug*(), and *mayBill*() are the corresponding permissions. The predicates *isPI*(), and *isMD*() denote classification of the information contained in d_1: $\mathit{isPI}(a, d_1)$ if d_1 is *Personal Information* of agent a and $\mathit{isMD}(a, d_1)$ for *Medical Data*. The predicates *isDoctorOf*(), and *onStaffOf*() denote professional relationship between agents: $\mathit{isDoctorOf}(a, b)$ if a is the doctor of b and $\mathit{onStaffOf}(a, b)$ if a is on the staff of (doctor) b.

Using the scenario-specific actions and predicates we can translate the Hospital's policy into the AC^2 policy language (see Figure 9).

$$\mathrm{h1} = \forall a, d_1 . \mathit{isPI}(a, d_1) \to \mathit{owns}(a, d_1).$$

$$\mathrm{h2} = \forall a . \mathit{isMD}(a, d_1) \to \mathit{owns}(a, d_1).$$

$$\mathrm{h3} = \forall a, b, d_1 . \big(\mathit{isDoctorOf}(b, a) \wedge \mathit{isPI}(a, d_1)\big) \to \mathit{mayRead}(b, d_1)$$

$$\begin{aligned}\mathrm{h4} = {} & \forall a, b . \big(\mathit{isDoctorOf}(b, a) \wedge \mathit{isMD}(a, d_1)\big) \\ & \to \big(\mathit{mayRead}(b, d_1) \wedge \mathit{mayUpdate}(b, d_1)\big).\end{aligned}$$

$$\mathrm{h5} = \forall a, b, c . \mathit{isDoctorOf}(b, a) \to \ \mathit{mayGiveDrug}(b, a, c).$$

$$\begin{aligned}\mathrm{h6} = {} & \forall a, b, c, d_1 . \big(\mathit{isDoctorOf}(b, a) \wedge \mathit{onStaffOf}(c, b)\big) \\ & \to \ \mathit{maySay}(b,\ c,\ \mathit{mayGiveDrug}(c, a, d_1)).\end{aligned}$$

$$\begin{aligned}\mathrm{h7} = {} & \forall a, b, c, d_1 . \mathit{isAdministrative}(c) \\ & \to \big(\mathtt{giveDrug}(b, a, d_1) \overset{!}{\to} \mathit{mayBill}(c, a, d_1)\big).\end{aligned}$$

Figure 9: The hospital's policy of Figure 8 written in AC^2's policy language.

4.2.2 Examples

Let us instantiate the general setting to a concrete instance, and give some examples of how AC^2 would work in this setting. We have patient Paris, doctor Alice, Dave, nurse Bob, and administrative employee Charlie.

Below we use the following notation: When an action, say `read`(), occurs in the system we write a separate line

`act`$_i$: `read`(),

where `act`$_i$ is an instance of an action of the type `read`(), requiring the permission *mayRead*().

Example 11 (Patient consent) *Paris starts to visit the hospital, where she visits doctor Dave. Paris becomes a patient of doctor Dave by explicitly acknowledging the patient-doctor relationship in a policy communication.*

`act`$_1$*:* `comm`(*Paris*, *Dave*, *isDoctorOf*(*Paris*, *Dave*))*.*

Doctor Dave logs this action. Dave is now allowed to read and update the health record of Paris. He has retrieved the two parts PI_Paris and MD_Paris. Doctor Dave reads the PI section of Paris's record, to remind himself of Paris's personal details, and updates the MD section of her record.

$\texttt{act}_2$*:* $\texttt{read}(\mathit{Dave}, \mathit{MD\_Paris})$*.*
$\texttt{act}_3$: $\texttt{update}(\mathit{Dave}, \mathit{MD\_Paris})$*.*

Now doctor Dave needs to get a second opinion from another doctor. He asks Paris for consent. Paris agrees and prepares the policy ϕ_{Paris} for Dave, where ϕ_{Paris} is,

$$\forall b, d_1 \; \big(\mathit{isMD}(\mathit{Paris}, d_1) \wedge \mathit{isDoctor}(b)\big) \rightarrow \; \mathit{maySay}(\mathit{Dave}, \; b, \; \mathit{mayRead}(b, d_1)).$$

Paris's policy, hence, is more specific, and more permissive, than the hospital's policy. The new policy permits doctor Dave to delegate, to any doctor, the permission to read the MD section of Paris's medical record. The PI section of her record remains private. Paris communicates her new policy ϕ_{Paris} to doctor Dave:

$\texttt{act}_4$*:* $\texttt{comm}(a, b, \phi_{Paris})$*.*

Dave logs this action.

Doctor Dave logs the actions $\texttt{act}_1$, and $\texttt{act}_4$ because the evidence of these events may be useful for him later on, in case of an audit. For example, Dave could use $\texttt{act}_4$ later on to prove that he was allowed to show the health record of Paris to another doctor. Actions $\texttt{act}_2$ and $\texttt{act}_3$ are of no interest to him.

Auditors on the other hand may keep an independent (e.g. random) track of actions, in so-called *audit trails.* They might be interested rather in $\texttt{act}_2$, and $\texttt{act}_3$ because they are actions that actually affect the privacy and integrity of health records.

The hospital's privacy officer is one of the auditors in the EHR system. It monitors queries to the health record database, both to detect anomalous behavior and to ensure that the hospital's policy is adhered to. The communications between doctor Dave and his patient Paris (containing AC^2 policies) are of no direct concern to him. These actions may become known to him in the course of an audit, being used in some justification proof. Independently, an external auditor controls the financial accountability of the hospital, i.e. to ensure that only actual costs are billed to patients. This external auditor is not interested in access to health records, but rather in which drugs have been given to Paris, and how often.

Example 12 (Internal hospital privacy audit) *The hospital's privacy officer asks Dave for a justification for having accessed Paris's record (actions* $\texttt{act}_2$*, and* $\texttt{act}_3$*). We abbreviate Paris, Dave, and MD_Paris, by a, b, and d_1 in this example. To give a justification Dave (agent b) uses h_3 from the hospital policy, his logged evidence of* $\texttt{act}_1$*, and a proof of*

$$[h_3, \mathit{isMD}(a, d_1)]; [\texttt{act}_1]; \nu \vdash_b \mathit{mayRead}(b, d_1).$$

Dave's proof is as follows.

$$\dfrac{\dfrac{\dfrac{\pi_1 \qquad \pi_2}{[\mathit{isMD}(a, d_1)]; [\texttt{act}_1]; \nu \vdash_b \mathit{isDoctorOf}(b, a) \wedge \mathit{isMD}(a, d_1)} \wedge R \qquad \dfrac{}{[\mathit{mayRead}(b, d_1); \nu; \nu \vdash_b \mathit{mayRead}(b, d_1)} I}{[(\mathit{isDoctorOf}(b, a) \wedge \mathit{isPI}(a, d_1)) \rightarrow \mathit{mayRead}(b, d_1), \mathit{isMD}(a, d_1)]; [\texttt{act}_1]; \nu \vdash_b \mathit{mayRead}(b, d_1)} \rightarrow L}{[h_3, \mathit{isMD}(a, d_1)]; [\texttt{act}_1]; \nu \vdash_b \mathit{mayRead}(b, d_1)} (\forall L)^3$$

Here π_1 and π_2 are proofs of isMD(a, d_1), and isDoctorOf(b, a), using the proof rules init (I), obs-act, and $\texttt{act}_1$*, the patient consent.*

The auditor can check Dave's proof, and concludes that, indeed, Dave can account for action $\texttt{act}_2$*, and* $\texttt{act}_3$*. For technical implementation (in Twelf) of proof checking we refer to the previous chapter.*

In this example *patient consent* is literally translated to a policy communication. At the same time, the auditor only observes the actions that affect the health records, not the policy communications. The auditor can be initially unaware of the policy communications between Paris and Dave. AC^2 supports delegation without the need to keep track of it centrally.

Example 13 (Urgent recovery) *Unexpectedly, Paris arrives injured at the hospital. Her doctor Dave is off duty, but doctor Alice is in, on a shift with nurse Bob. Alice treats Paris immediately. Let us go over the actions during Alice's shift. First, Bob starts his shift with doctor Alice. Alice affirms Bob is a nurse on her shift.*

$\texttt{act}_5 : \texttt{comm}(\mathit{Alice}, \mathit{Bob}, \mathit{isOnStaffOf}(\mathit{Bob}, \mathit{Alice}))$.

Bob logs this for later. After a while in the shift, Paris arrives injured at the hospital.

Informally Paris asks for treatment. Alice reads and updates the MD section of Paris's health record:

$\texttt{act}_6 : \texttt{read}(\mathit{Alice}, \mathit{MD\_Paris})$.
$\texttt{act}_7 : \texttt{update}(\mathit{Alice}, \mathit{MD\_Paris})$.

Informally doctor Alice tells nurse Bob to give Paris the drug Qurol (a general purpose medicine), and Bob gives the drug.

$\texttt{act}_8 : \texttt{giveDrug}(\mathit{Bob}, \mathit{Paris}, \mathit{Qurol})$.

For the purpose of billing, Bob has notified administrative employee Charlie that Qurol was given to Paris:

$\texttt{act}_9 : \texttt{notify}(\mathit{Bob}, \mathit{Charlie}, \mathit{Qurol}, \mathit{Alice})$.

Charlie logs this for later, and bills Paris (or her health care plan) for the drug. Charlie sends the bills.

$\texttt{act}_{10} : \texttt{bill}(\mathit{Charlie}, \mathit{Alice}, \mathit{Qurol})$.

A couple of hours later, when Paris is feeling a bit better, Alice comes to see Paris and explains to her that she will have to remain a bit longer in the hospital, for medication and observation. Alice also asks Paris for her formal consent to being her patient.

$\texttt{act}_{11} : \texttt{comm}(\mathit{Paris}, \mathit{Alice}, \mathit{isDoctorOf}(\mathit{Alice}, \mathit{Paris}))$.

Afterwards when the shift is over, Alice files the drug prescriptions for Paris, and she authorizes Bob, a-posteriori, to give the drug to Paris.

$\texttt{act}_{12} : \texttt{comm}(\mathit{Alice}, \mathit{Bob}, \mathit{mayGiveDrug}(\mathit{Bob}, \mathit{Paris}, \mathit{Qurol}))$.

Figure 10 shows the sequence of actions shown in the last example. Both Alice and Bob initially acted without formal authorizations, but only with informal ones. The formal authorizations are given a-posteriori (actions $\texttt{act}_{11}$ and $\texttt{act}_{12}$).

Example 14 (Auditing hospital shifts) *The hospital's privacy officer audits key actions in the EHR system, and some random ones, after each shift. The auditor may ask doctor Alice to account for having accessed Paris's health record (actions $\texttt{act}_6$ and $\texttt{act}_7$). The justification for this action relies on a later action ($\texttt{act}_{12}$) and the hospital's policy. For example, when Alice is asked to account for giving Qurol to Paris, she can send her log of action $\texttt{act}_{12}$, together with a proof of*

$$\nu; [\texttt{act}_{12}]; \nu \vdash_{\mathit{Bob}} \mathit{mayGiveDrug}(\mathit{Bob}, \mathit{Paris}, \mathit{Qurol}).$$

The auditor accepts this proof, and concludes that Bob complies to hospital policies.

Alice and Bob, in this example, can treat the patient immediately and deal later with the necessary details for administration and accountability. Initially not all the authorizations were available, so Alice and Bob have to arrange for authorizations *a-posteriori*. Charlie, on the other hand, did not perform any actions that rely on a-posteriori authorizations. Both types of justifications can be used in AC^2.

Example 15 (Billing) *The next morning Bob gives another dose of Qurol, in line with what Doctor Alice prescribed.*

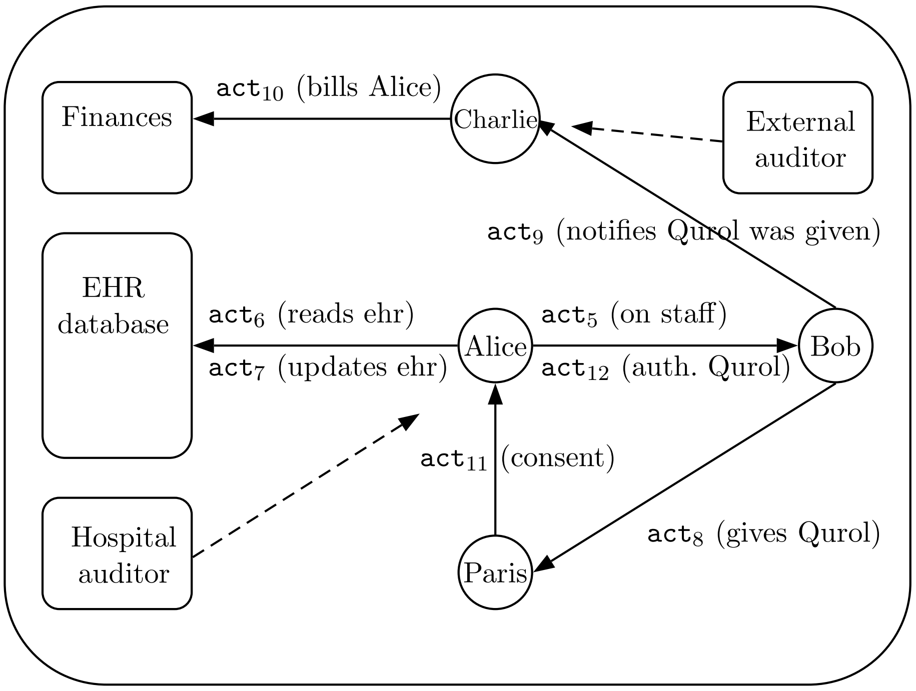

Finances
$\texttt{act}_{10}$ (bills Alice)
Charlie
External auditor
$\texttt{act}_{9}$ (notifies Qurol was given)
EHR database
$\texttt{act}_{6}$ (reads ehr)
$\texttt{act}_{7}$ (updates ehr)
Alice
$\texttt{act}_{5}$ (on staff)
$\texttt{act}_{12}$ (auth. Qurol)
Bob
$\texttt{act}_{11}$ (consent)
Hospital auditor
Paris
$\texttt{act}_{8}$ (gives Qurol)

Figure 10: The sequence of actions involving the patient Paris. The actions are numbered according to the order of their execution. So the actions $\texttt{act}_{12}$, denoting doctor Alice's authorization for nurse Bob to give the medicine Qurol, and $\texttt{act}_{11}$ denoting Paris's formal consent to being treated by doctor Alice, occur after the key actions $\texttt{act}_{6}$ to $\texttt{act}_{10}$. They serve Alice and Bob for later justification, for example to the Hospital's auditor.

$\texttt{act}_{14}$: *Bob administers the medicine.*

$\texttt{act}_{15}$: *Bob notifies Charlie that Qurol was given to Paris.*

Charlie logs this for later.

$\texttt{act}_{16}$: *Charlie bills Paris a second time for the drug.*

Charlie logs this, together with a reference to Bob's notification.

In this example we show AC^2's use-once obligations. The policy $h7$ allows Charlie to bill Paris each time someone gives her a drug. Charlie, when adding to the bill of Paris, needs to reference the corresponding notification by Alice. AC^2's proof system prevents from using the same notification twice. An external auditor can check financial accountability in this way.

It is important that all the critical actions that involve medical records are caught in the audit trail, to allow auditors to perform meaningful audits.

Let us assume for the sake of example that the hospital has outsourced the work of billing to a private enterprise, called *Clearinghouse X*). Charlie who works there and is responsible for billing the patients, has a regular office with a computer and an internet connection. Charlie could send an email to a friend outside the hospital attaching the notification by nurse Bob, that Paris was treated with Qurol. To prevent this from happening, the hospital could require access to the audit-trail of Clearinghouse X, that includes the email traffic by Charlie, and audit these actions too. Or it could rely Clearinghouse X to take measures, for example access control (audit-based or not), combined with legal clauses in employee contracts.

4.3 Legislation

Not only does AC^2 provide flexible access control for an EHR system in case of medical urgencies. We argue that AC^2 has a number of features that address the key requirements from legislation about EHR systems [84, 85] as well. Consider for example the Health Insurance Portability and Accountability Act (HIPAA) [85].

Example 16 (HIPAA) *The general principal for use and disclosure is stated as follows: A covered entity may not use or disclose protected health information, except either: (1) as the Privacy Rule permits or requires; or (2) as the individual who is the subject of the information authorizes in writing.* Covered entities *are health care providers (hospitals), health care plans (insurances), and health care clearinghouses (organizations that process health information). HIPAA permits disclosures of an individual's health records in the following cases:*

1. *Disclosure to the individual.*
2. *Disclosure for the purpose of treatment, payment, or medical operations.*
3. *Disclosure permitted informally by the individual.*
4. *Incidental disclosures, that occurs despite reasonable safeguards.*
5. *Disclosure for the public interest and benefit, for example prevention, donation, or law enforcement.*

Besides the 'permitted disclosures' listed here, patients can authorize additional disclosures for certain purposes (research, marketing, etc.). In all these cases, the covered entity must observe 'the principle of minimum necessary disclosure'. HIPAA also states a number of rights that patients have. The rights are called:

- *Privacy practice notice: the right to be informed of the covered entity's privacy practice.*
- *Access to the health records: the right to review and obtain a copy of their health records*
- *Amendment: the right to have errors in health records amended.*
- *Disclosure accounting: the right to an accounting of disclosures of their health records in the last six years.*
- *Restriction request: the right to request a restriction of disclosure of their health records.*
- *Confidential communication requirements: the right to confidential communication.*

The rest of HIPAA contains administrative requirements, organizational options, provisions for representatives and minors, and some US specific legal matters.

AC^2 addresses many of the requirements in HIPAA.

- The general priniciple of HIPAA states that both legislation, and patient consent should be taken into account. Patient consent is an important requirement in privacy legislation on health records. AC^2 supports patient consent directly by the *maySay*() predicate. Moreover it is easy to combine individual authorizations with general policy in AC^2 (see Example 11).
- HIPAA allows disclosures permitted *informally* by the patient. AC^2 supports informal consent by patients. Let us go back to the example: Paris consents first informally to being treated. Alice starts the treatment and deals with Paris's formal consent only after the treatment. This allows medical employees in such situations to continue with their work, without delay.
- HIPAA act stresses that patients have the right to justifications of past disclosures of their medical records (Disclosure Accounting). AC^2 provides a formal definition of this accounting and the tools to automate such accounting. Although not required in the HIPAA act, automation is convenient to be able to run audit tools without human supervision, enhancing both the privacy and the efficiency of the audits. This also means that the logging of actions, that is a requirement in the AC^2 framework, is already required by HIPAA.

- Finally, the HIPAA act states that *The Privacy Rule does not require that every risk of an incidental use or disclosure of protected health information be eliminated.* Therefore, the main drawback of AC^2, i.e. that with a-posteriori access control we can not completely exclude misuse, is in principle allowed by HIPAA.

On the other hand there are some HIPAA requirements that are not addressed by AC^2.

- HIPAA allows users to file a restriction request. AC^2 does not include a revocation mechanism. Combining communication and revocation of policies would be an interesting possibility for future work. A possible solution is to extend AC^2 with a distributed revocation mechanism, similar to that of SPKI/SDSI [75].

- HIPAA states that the user is always in control about his own health record. In AC^2 the user that creates a piece of data is owner of that data. In health care this is perhaps not always appropriate. Intuitively a doctor, when a new patient comes, creates a new file and fills it with health information he acquires along the way. A possible solution is to allow the doctor to create a new file, with only a placeholder for the owner, in such a way that eventually the patient, by uttering some kind of approval, becomes the owner of the file.

4.4 Related Work

The Cassandra system [18] was designed to implement an a-priori access control mechanism in an EHR system. The authors test the expressivity of the Cassandra policy language by expressing existing policies regarding access to medical data and activation of medical roles. The policy languages used in the Cassandra system are different flavors of *Datalog with constraints.* While Cassandra's policy language is in theory undecidable, it is argued that this is not problematic in practice. The AC^2 policy language that we use is not based on Datalog, but on a fragment of first-order logic (see Chapter 2), which is semi-decidable.

Rissanen et al. [74] address the issue of how to override safely the decisions of a preventive access control system called the Privilege Calculus. At each override a procedure starts to find the appropriate authority which is notified to audit the override. In our approach there is only a minimal preventive access control mechanism, which can not be overriden. Moreover, in our approach it is up to the auditors to decide when and which users to audit.

A related problem is the enforcement of copyrights in content-sharing systems (DRM). It has been argued that DRM can be used to enforce privacy policies. For instance, Conrado et al. [24] propose to use DRM to enable privacy in content-sharing systems and vice versa to use privacy as a driver for a wider use of DRM enabled devices. DRM however, unlike the AC^2, requires special (compliant) hardware or software at the application layer. This makes standard DRM unsuitable for EHR systems or enterprise-privacy systems. In the context of DRM, a type of a-posteriori access control was proposed by Shmatikov and Talcott [81]. There, a reputation-based trust management (TM) model is presented, in which the reputation of individual agents is determined by the fulfilment or violation of (DRM) licenses. We believe that Trust Management [62], coupled with auditing, may be an interesting solution, especially in large distributed EHR systems, for instance across a continent.

4.5 Conclusions

AC^2 is different from conventional access control, in that it allows the health care professionals to go ahead with their duties, without the need to obtain full authorization. The details of authorization are dealt with at a later (more convenient) time. Conventional a-priori access control system do not offer this kind of flexibility. In a conventional access control system expired, or incomplete authorizations block access to the data completely. In the hospital examples we gave, the quality of health care for Paris would be less.

Logging is a central part of AC^2. Useful events or performed actions have to be logged by the medical staff, for the purpose of accountability (a HIPAA [85] requirement). By keeping logs of such events and actions, doctors and nurses can account to multiple auditors that audit different actions at different times. In conventional a-priori access control system, on the other hand, the authorizations are only checked once by a single authority, i.e. at the moment access is requested. While additional logging and auditing is considered essential also in conventional access control [76], these audits are conducted often *without* using formal procedures. AC^2 provides a formal and automated auditing procedure. Automation allows for fast routine audits, and is more privacy-friendly than auditing by hand.

In the EHR setting, privacy is an important issue for both patients and medical staff. A-posteriori access control is in theory more intrusive to the user's privacy than a-priori access control. A-priori access control however is also coupled with audits of logs and user actions [76, 77]. An automatic audit procedure not only enhances the privacy of the patients, but also that of the medical staff, wary perhaps of human auditors that go through the logs by hand. Formal and public auditing procedures make the privacy protection mechanism also more transparent to the patients as well as to the medical staff. For future work it is interesting to investigate how to control the actions of the auditors in turn and how we can achieve maximal privacy of the (honest) users with respect to those auditors.

A-posteriori access control has a characteristic drawback: it does not prevent misbehavior, hence it does not give the robust security guarantees that are required in e.g. military information systems. This is allowed by HIPAA (see Section 4.3). In many settings, a-posteriori access control can not replace preventive access control at all because the costs of incidental misuse are higher than the costs of a (too) preventive security mechanism. Furthermore, in AC^2 it is important that users can be held accountable for their actions, i.e. that a user will not vanish after executing (illegal) actions. While not explicitly mentioned in HIPAA, it is already common in medical settings to ask staff to account for past actions, for legal, medical, or financial reasons.

A more complex aspect is *mutual trust*. We assumed for simplicity that all users were equally trusted to utter security statements. Consider the case where some foreign doctor, from another hospital, made changes to the drug prescriptions for Alice. Strictly speaking, Dr. Dave can trust another doctor, however, Dr. Dave's employer may require him to make a more involved trust decision that involves checking the foreign doctor's reputation and expertise area. Making such decisions can be supported by using trust management (TM) systems [62]. On the one hand, the TM system should give a full view of how users can be trusted, and on the other hand, when an auditing authority finds that a user is not accountable, it should report this to the TM system to decrease part the user's reputation.

Chapter 5

Privacy Policies

5.1 Introduction

In the previous chapters we have shown how AC^2 can be used in fairly closed settings - in a dynamic collaborative environment in a consultancy firm (in Chapter 2), and in a hospital (in Chapter 4). In this chapter we argue, by comparing AC^2 to two existing (and well-known) privacy solutions, that AC^2 also provides a good solution in a third scenario: An enterprise that collects (private) customer data on its website, processes it and then shares it with partner enterprises.

Today, there is a widespread use of internet services such as online shops, or customer support websites. Often, these internet services collect private data about the user, such as the name, address, purchase history and so on. Usually this data is collected for a particular purpose, for instance to provide a service such as parcel delivery. However, because the customer has no control, or clue, about how the data is processed or stored, the data could also be used for other purposes. Let us give an example.

Example 17 (Cross marketing) *Alice buys vegetables everyday at the local market. She also orders, for her neighbor, a box of chocolates every week from an enterprise called 'Store' via their online website. The enterprise 'Store' has collected private data about Alice (such her name, address and purchase history), to ensure the correct delivery of the chocolates.*

The enterprise has a partnership with a health insurance company. The insurance company sponsors certain healthy products (such as low-fat vitamin enriched butter): Customers who regularly buy healthy products are asked if they are interested in receiving a discount coupon from the partner enterprise that offers health plans. In that case the relevant customer data (name, address, amount of butter purchased) is sent to the health insurance company, and the discount coupon is sent to the customer.

The insurance company is keen on increasing profits, and it uses several tools to do so. One mechanism is to give incentives (like sponsoring butter), another mechanism is to assign risk profiles to customers and customers of partner enterprises, and certain insurance plans are only offered to people with low risk.

One day, by mistake, the insurance company receives the purchase history of customers who bought chocolates. The insurance company decides to change the risk profile for Alice from 'neutral' into 'risky', because of the chocolates. She will be excluded from discount offers, or, if she already has plan, she may even be excluded from extending her existing plan.

In this example, Alice's privacy is breached, with direct consequences for her insurance plan. Fortunately today there exist laws that force enterprises to comply with precise privacy regulations [83, 84]. For instance, the European Union in 1995 issued a directive to its member states that regulates the collection, storage and processing of personal data across the EU [83]. In 2002 this directive was extended to adapt to the ongoing changes in the electronic communications sector [84]. Among other things, the directives demand that enterprises only collect private data for specified and legitimate purposes and that the data may not be processed in ways incompatible with those purposes [83].

To see an example of how the EU directives translate to requirements for computer systems, consider the previous example. On the checkout page of their website, Store collects the private data of customers making a purchase, such as their name and address. The first requirement, that follows from the directive, is that the checkout page contains explicit statements about the purposes for which the private data are being collected. The second requirement is that the enterprise and her partners do not process the private data for purposes other than those stated on the checkout page. In Example 17 then both Store, and their partner insurance company would have been violating the EU directive.

We distinguish three stages (see Figure 11) in the life-cycle of the private data in this example. The first stage is the moment of collection by the enterprise. In the example the private data is collected via a website. The second stage is the processing inside the enterprise. In the example the private data is used for sending out the parcels, and billing. The third stage is the processing outside the enterprise, at the partner sites. In the example the private data is used for offering discounts, and assigning risk profiles to customers.

In this chapter we show how AC^2 can be used to express and enforce privacy policies, and we compare AC^2 to two (well-known) privacy solutions (E-P3P, and P3P). In Section 5.2, we discuss P3P which is widely used in stage 1 (in Figure 11), where the enterprise collects the data from the customer. In Section 5.3, we discuss E-P3P, which is designed for privacy

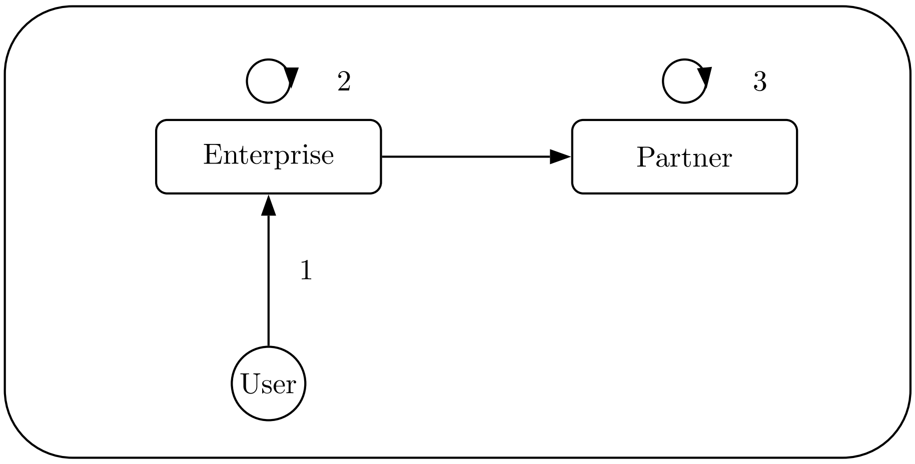


Figure 11: The lifecycle of the private data in e-commerce.

protection of customer data inside an enterprise (stage 2). In Section 5.4 we argue that AC^2, can be used in all three stages, and is particularly suitable for the protection of private data across enterprise boundaries (stage 3).

5.2 Platform for Privacy Preferences

In most countries, privacy legislation demands that websites provide a *privacy statement* explaining which private data is collected, who will see it, for how long it will be stored, et cetera [83, 43]. However, privacy statements and disclaimers printed on websites are often long and complex, and difficult to understand quickly by the ordinary internet user. Platform for Privacy Preferences (P3P) [29] - a W3C recommendation - aims to solve this problem by supporting automatic analysis of privacy statements. P3P allows enterprises to translate their privacy statements into a standardized XML-based format, using a common vocabulary, to be placed on their websites. When a website supports P3P, a visitor can employ an automatic tool to analyze the website's privacy statement and quickly decide if they are satisfactory.

To illustrate how P3P works, let us see an example.

Example 18 (Online privacy statements) *Alice visits an online store and after choosing a product she goes to the checkout page. Here she fills out a form with some private data: i.e. her name and credit card number. The store states in a privacy statement that it will use these data only to complete the transaction. In addition, the checkout form has a non-obligatory field for the customer's email address. The store states in a second privacy statement that the data will be used by the insurance company for discount offers. Both*

```
<policies xmlns='http://www.w3.org/2000/P3Pv1'>
1)<policy name='checkout'
         entity='Store, 5th Avenue, Manhattan, PO 10001, USA'>
2) <disputes>service='PrivacySeal.orG/DisputeResolution'</disputes>
3) <access><none/></access>
4) <statement>
     <purpose><current/></purpose>
     <recipient><ours/></recipient>
     <retention><stated-purpose/></retention>
     <data-group>
     <data ref='#user.name'/>
     <data ref='http://www.store.com/p3pvoc#card.number'/></data-group>
    </statement>
    <statement>
     <purpose><contact required='opt-in'/></purpose>
     <recipient><ours/></recipient>
     <retention><stated-purpose/></retention>
     <data-group>
     <data ref='#online.email'/></data></data-group>
    </statement>
   </policy>
  </policies>
```

Figure 12: A sample P3P policy

privacy statements can be translated in P3P and the resulting P3P policy, containing the two statements, is shown in Figure 12.

The example allows us to see the different (XML) elements in a P3P policy:

1. The policy element specifies a name for the policy, and the *entity* which indicates the issuer of the policy.
2. The *disputes* element describes how possible conflicts over the privacy policy may be resolved (e.g., by which court, or other entity). This is not binding, in the sense that the enterprise is still subject to the legal ways to resolving a privacy dispute.
3. The *access* element indicates whether the submitted data may be accessed by the subject after it has been collected. This can be used for instance to verify the accuracy of the collected data. This policy states that access is not possible.
4. The key elements of the P3P policy are the *statements* which describe, per data item (the element *data*), for which *purpose* it is collected, who is allowed to access it (in the *recipient* element), and for how long it will be stored (in the *retention* element).

- The purposes are respectively *current*, which refers to the online purchase and *contact*, which indicates that the information can be used to contact the user for *marketing of services or products*. The purpose element may also contain an attribute indicating how a user can express his consent to the purpose. In this case, explicit *opt-in* is required for the purpose contact.
- The recipient value *ours* means that the data can only be accessed by the store (e.g. it will not be given to third parties), while the retention value *stated-purpose* means that the data will only be retained for a period needed for the stated purpose.
- The *data* element is specified by a reference to an element in a so-called P3P data schema, e.g. `#online.email`, in P3P's default data schema, and `http://www.store.com/p3pvoc#card.number`, in the store's custom vocabulary.

In the example above, if Alice's browser supports P3P it can compare the policy with Alice's privacy preferences automatically. One of these preferences states that she wants to be warned when sites request information for purposes other than *current*. In this case the browser, can notify her that she *may or may not supply her email address for marketing of services or products*. Her advantage is that she does not have to read the site's privacy statement to find out what they mean and which fields are optional.

Since its introduction in 1997, P3P has received considerable attention [43]. P3P adoption was particularly stimulated by the introduction in 2001 of a privacy slider in Microsoft's Internet Explorer 6. This privacy slider allows the user to determine which websites may set and retrieve *cookies*, according to their P3P policies. Cookies from websites with no P3P policies (or with an unsatisfactory one) are blocked by the browser.

A drawback of P3P is that, despite its simplicity, P3P policies can be ambiguous [91]. For instance, one could refer to the same data element twice with different retention periods, within the same policy. Ambiguities may result in legal risks for the issuers as their policies may be interpreted in unexpected ways [80]. This also makes the development of P3P compliant browsers more difficult. As a matter of fact, despite the fact that P3P was designed to be interpreted by browsers, there is no definition of how a browser should interpret policies, and there are no guidelines for writing 'browser-friendly' policies [80].

We should mention that while P3P addresses the problem of representing a website's privacy policy, it does not address the problem of enforcing them. The use of P3P alone does not give assurance about the *actual* privacy practices in the backend of the website. Critics have even suggested that the online industry, by adopting P3P, is only giving an appearance of protecting privacy, to avoid stricter legislation [23].

5.3 Platform for Enterprise Privacy Practices

As mentioned earlier, in many countries, legislation regulates the collection and the use of private data. This requires enterprises to enforce privacy policies that prescribe for example when certain data should be deleted, by whom it may be accessed and for which purposes. As we saw in the previous section, P3P can be used to represent such privacy policies on websites, but it does not address the problem of enforcing them inside the enterprise. The Platform for Enterprise Privacy Practices (E-P3P) - introduced in 2002 by Karjoth et al. - addresses this [53]. E-P3P provides an XML-based language to express privacy policies as well as a framework with specific privacy functionality to enforce these policies.

In the E-P3P architecture an enterprise collects private data at so-called *collection points*. Here individuals, e.g. customers, submit private data to the enterprise, after agreeing with the enterprise's privacy statements. Each collection point has a *form* which associates the private data with its subject, declares its *type*, e.g. credit card or email address, and the subject's *consent choices*. This association remains intact in the enterprise's backend and it may even travel to another enterprise. In E-P3P this is called the *sticky policy paradigm* [53]. The sticky policy does not refer to enterprise policies but refers to the privacy statements and the filled in consent choices on the data collection form that stick to the private data.

The privacy officer of the enterprise declares, by using E-P3P's policy language, who can access which type of data for which purposes. The privacy policy can also refer to the subject's consent choices and to certain privacy obligations, e.g. *delete the data in 30 days.* Operations in the enterprise's applications are then mapped to terminology used in the privacy policies, and vice versa. For example, the 'send' operation of a mass-mailer system, used in the marketing department, is mapped to the term *read for the purpose of marketing* in the privacy policy. The term *delete the subject's email* in the privacy policy is implemented as an 'unsubscribe' operation of a mailing list system.

Finally, access to the private data of a subject is granted in two steps. The access to the legacy enterprise application is evaluated by an access control system, for instance taking into account employee roles, which is independent of the E-P3P system. Then, the legacy application makes an access request to a *privacy enforcement system* for the subject's private data. The privacy enforcement system decides whether access should be granted, by evaluating the enterprise policy and by matching against the subject's consent choices. If access is granted, then the privacy enforcement system also executes possible privacy obligations specified in the enterprise policy.

Example 19 (Enterprise privacy policies) *Consider the previous example of an online store collecting private data on its checkout page. The*

```
1) <ep3pPolicy version = '1.2'  issuer = 'Store'
     vocabulary-ref = 'http://www.Store.com/Voc'
2) default-ruling='deny'>
3) <rule>
     <dataCategory>allData.creditCardData</dataCategory>
     <purpose>business.billing</purpose>
     <userCategory>employees.billing</userCategory>
     <ruling>allow</ruling>
     <action>read</action>
     <obligation action=deleteWithIn(30)</obligation>
     <condition/>
   </rule>
4) <rule>
     <dataCategory>allData.contactData</dataCategory>
     <purpose>business.marketing</purpose>
     <userCategory>employees</userCategory>
     <ruling>allow</ruling>
     <action>read</action>
     <obligation>
     <condition>optInToMarketing=true</condition>
   </rule>
```

Figure 13: A sample E-P3P policy

enterprise that owns the online store wants customers to trust its privacy practices. To this end, it has published privacy statements on the checkout page and uses E-P3P to ensure that enterprise systems, and employees, behave according to them. These privacy statements specify that Alice's credit card number may be accessed by the employees from the Billing department provided that the purpose is billing *and that the data is subsequently deleted. In addition, employees may use Alice's email, address for marketing purposes, if Alice opted in to this purpose.*

The corresponding E-P3P policy is shown in Figure 13. We go over the details of this policy in the next paragraph. Now an employee of the marketing department wants to send an email with promotions to a number of customers (including Alice), by using a mass-mailing system. The mass-mailing system, after checking that the employee is authorized to use the system, sends a request to the E-P3P privacy enforcement system to see whether access should be allowed on the basis of the enterprise's privacy policy. The request has the following form:

```
<ep3pQuery>
 <userCategory>employees</userCategory>
 <dataCategory>allData.contactData</dataCategory>
 <purpose>business.marketing</purpose>
 <action>read</action>
</ep3pQuery>
```

The E-P3P policy enforcement engine (or reference monitor) decides whether to grant this request or not, based on the E-P3P policy. Let us go over the E-P3P policy of this example.

1. *The top E-P3P policy contains the E-P3P version number, the name of the issuer of the policy, the vocabulary used, and*
2. *the so-called default-ruling, which specifies the default decision that should be taken, when no rule in the policy applies to the request.*
3. *The first rule contains elements that specify that reading creditcard data, for the purpose of billing, provided the requester deletes the data within 30 days.*
4. *The second rule allows requests for reading contact data of a customer, for the purpose of marketing, provided the customer has opted-in to marketing.*

The E-P3P policy enforcement engine will allow the request and grant access (`allow`*) to the contact data, provided the condition* `optInToMarketing` *holds. A request for creditcard data with the purpose billing would also be allowed. Other requests would be denied (by the default-ruling* `deny`*).*

This example shows the key elements of an E-P3P policy: a reference to the vocabulary used, the policy's default-ruling and the policy's ruleset. The ruleset is a list of E-P3P rules that declares which user categories can perform which actions on which data categories and for which purposes. The vocabulary allows one to define *hierarchies* of data categories, purposes, and data users, which are convenient to refine a privacy policy in a hierarchical sense. For example, the ruling `ALLOW` inherits downwards in the hierarchies: When a rule allows a request for `allData`, then a request for `allData.creditCardData` is also allowed. Denials, on the other hand, are inherited both downward and upward. For example if a rule denies access to allData.creditCardData, then the requests for allData or allData.creditCardData.cardType are also denied.

E-P3P was introduced by Karjoth et al. [53], while the full XML-based language and semantics for E-P3P policies was defined by Ashley et al. [14]. EPAL [13], a language similar to (and derived from) E-P3P was submitted by IBM to the W3C for standardization, has not yet been endorsed. EPAL does not have some of the advanced features of E-P3P such as hierarchies for obligations and conditions, or procedures for the composition of policies. EPAL has been implemented by IBM in the *IBM Tivoli Privacy Manager*, a system providing automatic management of private data to bring down the enterprise's costs of privacy management and to decrease the risks of unauthorized disclosures.

It is worth remarking that, although the names of E-P3P and P3P are similar, they are used in different stages. One is used to manage privacy rules internal to an enterprise, the other is used to communicate, in a standardized way, privacy policies to internet users. Karjoth et al. [52] have linked the two by proposing a system that publishes P3P policies directly from internal enterprise policies. Yu et al. [91] on the other hand argue that P3P policies should be more long-term promises, which should not change each time an internal enterprise practice changes.

5.4 Audit-based Compliance Control

P3P and E-P3P offer methods for specifying privacy policies and enforcing them inside an enterprise. The issue of protecting private data across enterprise boundaries remains open, however. This is an important problem, given that outsourcing of business processes is increasingly common.

In this section we describe how AC^2 provides an alternative approach to privacy protection. AC^2 does not only allow to specify privacy policies, like P3P, it also provides a framework for auditing compliance to these policies. AC^2 addresses the issue of compliance to policies for data that move across different security domains.

Example 20 (AC^2 enterprise privacy policy) *Let us return to the enterprise privacy policy of our previous example. It can be expressed as two AC^2 policies, as follows:*

$$\begin{aligned}\phi_1 = &\forall_{a,d_1}\big(\mathit{employee.billing}(a) \wedge \texttt{purchaseorder}(d_1)\\ &\qquad \to \mathit{mayRead}(a, d_1, \mathit{billing})\big).\end{aligned}$$

$$\begin{aligned}\phi_2 = &\forall_{a,b}\big(\mathit{employee}(a) \to (\texttt{optin}(a, \mathit{marketing})\\ &\qquad \overset{?}{\hookrightarrow} \mathit{maySend}(b, a, \mathit{marketing}))\big).\end{aligned}$$

The policy ϕ_1 states that employees at the billing department can read the purchase orders, for the purpose of billing. Here the purpose is a parameter of the predicate mayRead(). The action $\texttt{optin}(a, \mathit{marketing})$, in policy ϕ_2 is a use-many obligation. The policy employee can send marketing content to Alice if she has opted in to marketing.

When Alice makes her purchase, she posts a form with the 'agree to marketing' checkbox marked. The store logs her action:

$\texttt{act}_1$*:* $\texttt{optin}(\mathit{Alice}, \mathit{marketing})$

Now employee Charlie sends a marketing email to Alice.

$\texttt{act}_2$*:* $\texttt{sends}(\mathit{Alice}, \mathit{marketing})$

The action is not, like in conventional access control, checked a-priori against the store's privacy policy, but Charlie can justify action $\texttt{act}_2$*, a-posteriori, to an auditor in the enterprise, a by referring to Alice's opt-in (*$\texttt{act}_1$*), and the store's policy* ϕ_2*.*

In AC^2 , compliance to policies is not enforced in a conventional way but instead users may have to justify their actions a-posteriori, i.e. users may be audited. By holding the users accountable for their actions afterwards, policy violations are prevented by deterrence rather than by prevention. In order to do this, auditors must dispose of meaningful (and tamper-proof) audit trails. Such audit trails are already present in enterprises for the purpose of accountability, and are also (though implicitly) assumed in the P3P and E-P3P frameworks.

Suppose the enterprise has outsourced marketing mailings to a specialized marketing agency. When customer data leaves the security domain of the store, it can not use conventional access control [50, 68, 76] anymore to protect the data, since the access control system of the store does not extend to the marketing outsourcing. Let us show how AC^2 can be used in such a setting.

Example 21 (Outsourcing) *The enterprise Store has outsourced marketing to another enterprise called Marketeer. Instead of giving Marketeer a list of email addresses of customers that have opted in to receiving marketing, Store wants to keep track of how often customers receive mailings, because they do not want to annoy them, and they want to know if additional marketing is appropriate or not.*

The enterprise Store has specified a privacy policy that Marketeer has to comply to. The policy is as follows:

$$\phi_3 = \forall_{a,b}\big(\texttt{notifies}(\mathit{Store}, \mathit{Marketing}, a) \xrightarrow{!} \mathit{maySend}(b, a, \mathit{marketing})\big)$$

An employee, Carol, of the enterprise Marketeer has just sent a marketing mail to Alice, and notifies the Store that Alice was sent a marketing mailing: $\texttt{act}_3$*:* $\texttt{sends}(\mathit{Carol}, \mathit{Alice}, \mathit{marketing})$

$\texttt{act}_4$*:* $\texttt{notifies}(\mathit{Store}, \mathit{Marketing}, \mathit{Alice})$
Carol logs $\texttt{act}_3$ *together with a reference to* $\texttt{act}_4$*, indicating which (use-once) obligation has been consumed. Suppose Store, or another entity, wants to audit the enterprise Marketeer, Carol can justify* $\texttt{act}_4$ *by using the policy* ϕ_3*, and an excerpt of her log containing* $\texttt{act}_3$*.*

The latter example shows an important advantage of using AC^2 for enterprise privacy policies, as opposed to E-P3P, or a conventional access control mechanism. Even if the enterprise Marketeer uses E-P3P (or another conventional access control system) to enforce the policy ϕ_3, only the security officer of Marketeer who implements E-P3P will acquire guarantees about data-protection. The enterprise Store on the other hand can only hope that the right E-P3P policies are in place, and that they are enforced continuously, because the enforcement point is out of the control of Store. AC^2 also does not require, like traditional access control systems, a trusted enforcement point that enforces compliance to policies. AC^2 allows both enterprise to adopt their own systems for data-protection.

Moreover AC^2 allows multiple auditors to audit actions concurrently and independently. This is in line with the common practice in which different stakeholders (such as internal auditors, customer organizations, and corporate accounting bureaus) audit compliance to regulations, and policies, independently of each other (for example, at different times, and targeting different actions).

A drawback of AC^2 is that misuse is not prevented but only *deterred*. Compliance to policies is only checked *a-posteriori* by the *auditing authorities*. This implies that all the users of the system are *auditable* and that complete *audit trails* are available to the auditors. This fits well with e.g. hospitals or enterprises, where users, can be held accountable for their actions and audit trails are often already part of the (security) requirements. It may be hard to realize these requirements in more open settings. Alternatively, instead of using the audit trails inside the enterprise, one could use external audit trails, for example by asking users directly to supply evidence of a postal mailing or a received call.

5.5 Related Work

An extensive survey of social, legal and technical aspects of P3P was given by Hochheiser [43]. In a more technical approach, Yu et al. [91] investigate the semantics of P3P: they find several inconsistencies and show how to restrict the language to avoid them. Byers et al. [22] survey the use of P3P on a large number of websites. They argue that a large number of websites is not compliant with the P3P specifications, and that this may yield legal problems for these websites. The P3P Preference Language (APPEL) [28] was developed by Cranor et al. to allow users to express preferences about P3P policies. With APPEL, users can specify which P3P policies they find acceptable and which not. Yu et al. [91] develop another kind of P3P preference language. This approach is based on the semantics of P3P, unlike APPEL, which is based on the syntax of P3P. Related to P3P is the Resource Description Framework (RDF) [57]. RDF was developed to represent information on the web in a machine-readable format. Although it is not specifically intended to be used for privacy practices, it may be used to express P3P policies. RDF has been proposed as an alternative for APPEL [28].

E-P3P is an extension of Jajodia et al.'s Flexible Authorization Framework (FAF) [49]. Like in E-P3P, FAF provides a policy language that can specify both positive and negative authorizations and uses hierarchies for objects and users. However, FAF does not allow the use of obligations, and does not include a special construct to express the purpose of an access request. The notion of privacy obligations in E-P3P is similar to the provisions in Jajodia's Provisional Authorization Specification Language (pASL) [48]. In pASL a principal is granted access to an object if it causes certain conditions to be satisfied. In E-P3P, obligations are treated opaquely, as methods that are called and return a value, while in pASL obligations are treated more in detail by using a temporal logic. E-P3P shares some similarities with XACML [66], an OASIS standard for access control systems. XACML is XML-based and uses object and data hierarchies, as well as conditions and obligations. XACML is also inspired by FAF [49], and, although it is not specifically intended for enterprise privacy policies, it can be used for protecting private data inside an enterprise. As an example of this a policy for the protection of medical records is shown [66]. Although XACML does not have a special *purpose* construct, like the one in E-P3P, it has been added in XACML's so-called privacy profile. Anderson [10] compares EPAL [13] and XACML and concludes that EPAL corresponds mostly to a subset of XACML and that it lacks certain features required for access control and privacy. Stufflebeam et al. [82] present a practical case study of E-P3P and P3P. Here the authors implement a number of health care policies in both EPAL and P3P. Among other things, they conclude that many promises expressed in natural language privacy policies are neither expressible in P3P nor enforceable with EPAL. More closely related to the AC^2 policies, Originator Control (ORCON) policies [67] were introduced as an alternative for discretionary and mandatory policies. In ORCON policies, the original owner of the data can always change the access rights on the data, while the current owner of the data can not do so. This fits well with the privacy regulations in which the subject should retain control over its personal data [83]. ORCON however stores policies centrally, whereas AC^2 allows policies to be communicated between users and systems.

5.6 Conclusions

In the P3P system, privacy statements are formatted using XML and a common vocabulary, to allow for automatic analysis of the statements. P3P is well-established, in the sense that there are many popular websites that use P3P [22]. Also there are a number of tools that generate natural language statements from P3P statements [43]. A drawback of P3P is that it does not distinguish between different types of access. For example, it is impossible to specify that certain employees may update personal data, while others may not. This makes it cumbersome to use for access control inside an enterprise.

The E-P3P system addresses this. E-P3P distinguishes between different types of access and enables the use of obligations and conditions. Although E-P3P itself is not (yet) an endorsed standard, it corresponds to a subset of an OASIS standard, i.e. the XACML access control language [66]. In a way they are complementary because E-P3P assumes the existence of access control policies, independent of the privacy policies. E-P3P policies can contain prohibitions, i.e. rules that deny access, which makes the language more expressive than the AC^2 policy language. However it seems complicated to move E-P3P policies from one enterprise to another. The new policy may cause conflicts and it may even be bypassed altogether due to other policies that are incompatible [15]. Moving policies may be needed in enterprise collaborations where private data are exchanged, guarded by policies. Furthermore, the use of E-P3P can only give assurances to other enterprises, when they assume that the enterprise is trusted to implement E-P3P correctly [53]. This may be a too strong assumption in the setting where enterprises dynamically form coalitions to exchange private data.

In AC^2 this assumption is relaxed. Here it is assumed that the enterprise can misbehave, while compliance to privacy policies can be verified by (external) auditors, through a formal auditing procedure. AC^2 is designed for a distributed setting, and it is easy to move policies across enterprise domains for instance accompanying private data. When policies are sent from one enterprise to another, the question is raised whether one can trust the sender of the policy. For example, a rogue enterprise could be set up for the purpose of distributing fake customer consent statements to other enterprises. To address this problem one could extend AC^2 with a trust management system to facilitate trust decisions about the sources of policies. Etalle and Winsborough show how AC^2 could be combined with the trust management language RT [32]. Furthermore, it may be interesting to couple the reputation of enterprises to the outcome of past audits, like in reputation-based systems [81].

Part II

Extending Role-based Access Control

Introduction

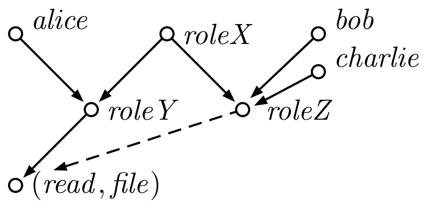
In the first part of this thesis we have proposed AC^2, a new framework for specifying and enforcing discretionary access control policies. We have shown by giving examples, how AC^2 can be used in different types of dynamic collaborative environments where private data is collected and processed. At the same time we are aware of the fact that changing to a new access control paradigm represents a big step for any organization, and that AC^2 is not likely to be adopted as-is by e.g. hospitals. In Part II of this thesis we take a more evolutionary approach to designing a flexible access control system more suitable for dynamic collaborative environments: We extend an *existing*, widely deployed, access control model (RBAC), to make it more suitable for dynamic collaborative environments.

Role-based Access Control (RBAC) [11] is a well-known standard for access control, used in many different organizations. To see an example, let us go back to the simple collaboration in the introduction, where the peers Alice, Bob and Charlie cooperate to produce a certain (confidential) document. Suppose that Alice, Bob and Charlie work in organizations where RBAC is used for the protection of documents. Suppose the RBAC policy is as follows.

$$\begin{tikzcd}
\mathit{alice} \arrow[dr] & & \mathit{roleX} \arrow[dl] \arrow[dr] & & \mathit{bob} \arrow[dl] \\
 & \mathit{roleY} \arrow[dl] & & \mathit{roleZ} & \mathit{charlie} \arrow[l] \\
(\mathit{read}, \mathit{file}) & & & &
\end{tikzcd}$$

Alice, like before, wants to benefit from Bob's help, but Bob does not have access to *file*. Alice would like to give Bob access to the document. RBAC however, does not allow Alice to make changes to the policy. Alice must therefore talk to an administrator to obtain a policy change, which is time-consuming for Alice, and discourages (ad-hoc) collaboration. In such a scenario it is tempting for Alice to show the document to Bob, and

Charlie in a way that circumvents the access control system (for example by using a USB stick, or by sharing her password). Ideally one would like to extend RBAC here, in such a way that it allows Alice to make a certain policy changes (the dashed edge depicted below).



Now, suppose Charlie works in a different branch of the organization, on a different system than Alice, or even in another organization. The RBAC standard does not specify how to deal with policy changes across different systems. One would like to have a model that specifies how to update the system used by Charlie, in such a way that Charlie, after the policy change by Alice, gets access to the documents.

A number of researchers have recently proposed RBAC extensions to allow delegation and decentralization of administrative authority [17, 27, 34, 78, 93]. We make two contributions to contribution to this line of research in Chapters 6 and 7.

- In Chapter 6 we define a new administrative model for RBAC, that gives more flexibility to the users than existing proposals - without being less safe. This kind of flexibility allows users to change policies better according to the day-to-day practical needs. We propose a formal definition of *administrative refinement* and define an ordering on the administrative privileges in RBAC. We also show that the privilege ordering is decidable.

- In the Chapter 7 we propose a model for the administration of RBAC in a distributed system and propose an administration procedure supporting the principle that different systems protect different sets of objects. We demonstrate that our administration procedure fulfills formal requirements deriving from safety and availability, and we show how it can be translated to a practical implementation. Finally, we extend the model to multiple decentralized administrative systems.

Remark 6 (RBAC versus AC^2) *Let us conclude this intermezzo by discussing the most striking differences between RBAC and AC^2.*

1. *In RBAC, access control decisions are made the moment a user requests access. In AC^2 on the other hand user requests are always granted, but justifications can be requested later on by an auditor.*

2. *The policy language used in RBAC is less expressive than the policy language of AC^2. The RBAC policy language is an instance of propositional logic, while AC^2 is based on first-order predicate logic. For instance AC^2 policies can contain conditions, or obligations that can not be expressed in RBAC. On the other hand, RBAC is decidable, which is important because the RBAC reference monitor must make access control decisions on the fly.*

3. *RBAC does not allow ordinary users to change or deploy policies, like in the AC^2 framework. In RBAC the policy can be set, and changed only by system administrators.*

Chapter 6

Refinement for Administrative RBAC Policies

6.1 Introduction

Role-based access control (RBAC) [11] is a well-known standard for access control, devised to make the assignment of users to privileges easy. In practice however, for example in hospitals or enterprises, RBAC policies can be large and dynamic, consisting of thousands of roles [34], and changing frequently. In such cases *policy management* can be a daunting task. The obvious approach to this problem is to divide the work and to delegate (bits of) administrative authority to other users. The advantage is that other users can adapt the access control policy to changing circumstances more easily, reducing administrative bottlenecks. Not only does this reduce the cost of maintaining the access control policy, it also avoids dangerous practices, such as sharing passwords or keys that should really remain secret. For example, it may be convenient to allow the head nurse to delegate database access to other nurses when they need it for particular tasks, without having to refer back to the hospital security officer. On the other hand, this kind of flexibility also introduces security risks, because users make the wrong changes to the RBAC policy.

The issue of designing *flexible* yet *safe* policy administration mechanisms for RBAC has received considerable attention recently [17, 27, 34, 78, 93]. To mention some of the research: In ARBAC [78] *administrative privileges* are assigned to a separate hierarchy of *administrative roles* and defined by specifying a range of roles that can be changed. Crampton and Loizou [27] take a more general approach, by using the same hierarchy for both administrative privileges and ordinary user privileges. Using the concept of *administrative scope*, they define which roles should have administrative privileges over other roles. In a similar approach, Wang and Osborn [88] divide the role-graph (a type of RBAC policy) into administrative domains. Each administrative domain has one administrator with privileges about the (roles in the) domain. In the Role-Control Center [34], administrative privileges over roles are defined in terms of *views*, which are subsets of the role hierarchy, and they can only be assigned to users that are assigned to these roles. There seems to be no consensus (yet) about which administrative privileges belong to which roles; each of the above mentioned frameworks differs on this issue. Some models motivate their choice by considerations that include the meaning of a *role* in a company, or the concepts of ownership, and responsibility, as one would find it in a company. On the other hand, Li et al. argue that interpreting the RBAC role hierarchy as a business organization chart can be misleading [58]. To give a simple example, in a business organization chart it is common to place the business manager on top. If we interpret this as an RBAC role hierarchy this would imply that the business manager can play all the roles in the organization, and consequently acquire all privileges. This is often inappropriate. For example, a hospital's manager typically does not perform medical operations, nor does she have the privileges of doctors, although the manager tops the doctors in the hospital's organization chart.

This chapter is a contribution to the above-mentioned lines of research on management of RBAC policies. We introduce the concept of *administrative refinement*, which is a relation on the set of administrative policies. Basically, an administrative refinement of a policy allows the same or fewer users to perform the same or safer administrative actions, as the policy itself. The safer administrative actions yield policies where users have fewer privileges. We use the concept of administrative refinement as a foundation for an ordering on the administrative privileges. We give the formal proof that the ordering is decidable, and how it can be used in an RBAC reference monitor to allow for an administrative RBAC model which is more flexible and more safe. In this chapter we will not assume any features that go beyond the *General Hierarchical RBAC model* (like constraints), which is the most commonly used RBAC model. In this way, our results can be used in wide a range of different RBAC models [17, 27, 34, 78, 93] that are based on the General Hierarchical RBAC model.

6.2 Preliminaries

We first introduce briefly the *General Hierarchical RBAC* model, as defined in the ANSI RBAC standard, because it is the most commonly used RBAC model [11, 34]. In Section 6.3 we extend this model with administrative privileges, yielding a general class of administrative policies.

The goal of an RBAC policy is to specify which users are permitted to perform which actions on which objects. The standard assumes the existence of a set names for users, U (ranged over by u, u',$\dots$), a set for roles R (ranged over by $r, r', \dots$), for actions A and for objects O. Privileges in RBAC are permissions to perform actions on objects (being data or other resources). The privileges form a set $P \subseteq A \times O$ (ranged over by p, p'), $(\mathit{read}, \mathit{table}_1)$, $(\mathit{print}, \mathit{colorA4})$. We refer to these privileges as *user privileges* (as opposed to the administrative privileges, that will be introduced below).

A standard RBAC policy is a triple $(\mathit{UA}, \mathit{RH}, \mathit{PA})$, where $\mathit{UA} \subseteq U \times R$, $\mathit{RH} \subseteq R \times R$, and $\mathit{PA} \subseteq R \times P$. The set of policies is denoted Φ. For the sake of brevity we treat $\phi \in \Phi$ as a single directed graph (a single set of edges $\mathit{UA} \cup \mathit{RH} \cup \mathit{PA}$).

Contrary to the RBAC standard, we do not require that the RH relation is transitive, or that the graph of the RH relation is acyclic.

We believe that the transitivity requirement complicates administration unnecessarily, in agreement with Li et al. [58]. Let us give an example. Suppose that the doctor role inherits the nurse role, and the nurse role a number of small roles (for tasks e.g.). Transitivity requires that the doctor role is also assigned to each of the roles of the nurses. Now if the tasks of the nurse change, not only the nurse role must be changed but also the doctor role.

Cycles in RBAC policies are sometimes considered to be redundant, but there are no strong reasons for excluding such policies.

Definition 7 (RBAC Policies) *Let U, R, and P be sets of users, roles, and user privileges, an RBAC policy ϕ is a tuple*

$$\phi = (\mathit{UA}, \mathit{RH}, \mathit{PA}),$$

where $\mathit{UA} \subseteq U \times R$ determines which users are member of which roles, $\mathit{RH} \subseteq R \times R$ determines which roles are assigned to other roles, and $\mathit{PA} \subseteq R \times P$ determines which roles have which privileges.

The set of RBAC policies is denoted $\Phi_{U,R,P}$. To simplify our exposition we treat a policy ϕ as a *directed graph*, defined by the *set* of directed edges $\mathit{UA} \cup \mathit{RH} \cup \mathit{PA}$. If there is a path from one vertex v to another v' we write $v \to_\phi v'$. Below we sometimes omit the subscript ϕ when the policy is clear from the context.

The RBAC *reference monitor* uses the policy ϕ as follows. Any user u can start a *session* [11]. The reference monitor allows the user to *activate* a role r in a session iff. $u \to_\phi r$. The privileges of the user's session are all the privileges $q \in P$ such that $r \to_\phi q$ for some role r activated in the session. A session can be regarded as a new temporary user with temporary assignments to roles (the active roles in the session). Sessions are an important safety mechanism, allowing users to apply the *principle of least privilege.* In the sequel however, for the sake of simplicity, we ignore the details about sessions and assume that users always activate all their roles.

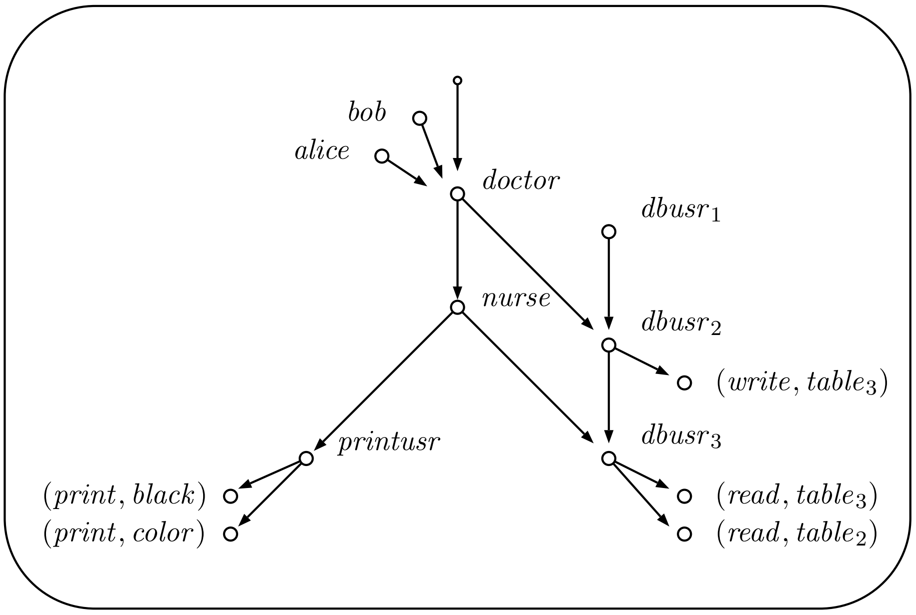


Figure 14: Example of a standard (ANSI) RBAC policy.

The user acquires the privileges of his own role, but also the privileges of the roles r' such that $r \to_\phi r'$ (the roles 'below' r). This is known as *privilege inheritance* and it is a well-known feature of ANSI RBAC.

Remark 7 (Privilege inheritance) *Inheritance distinguishes RBAC from other mandatory access control systems, such as the Bell-LaPadula model, where certain write privileges are assigned to low roles, but not inherited by users with high roles.*

We conclude the preliminaries by giving an example of a standard RBAC policy.

Example 22 (Standard RBAC) *Consider the setting of a hospital, where a database system* dbms *stores electronic health records in a number of tables $table_1$, $table_2$, $table_3$ etc. The health records can only be seen or changed by authorized personnel.*

Charlie, the security officer of the hospital, uses RBAC for access control in the dbms*. The policy is depicted in Figure 14. It allows Alice to activate the role* nurse *and the role* staff*. As a nurse she can only* read *the tables $table_2$ and $table_3$, while as a doctor she can also* write *in the table $table_3$.*

6.3 Administrative Policies

The RBAC standard specifies a number of *administrative functions and controls*, which can be used by an administrative authority to make policy changes [11]. Here we express administrative authority in terms of administrative privileges to model which users (or roles) can make which policy changes. There are two types of privileges: privileges for making new edges (denoted here by $\Box$), and privileges for removing edges (denoted here by $\Diamond$). We assign the administrative privileges to roles just like the user privileges are assigned to roles in standard RBAC. This approach is also advocated in the literature and the intuition behind it is that the RBAC policy can also be used to specify who can change the RBAC policy [27, 88]. The literature focusses on defining constraints on which roles can have which administrative privileges. For example, to prevent low roles from obtaining privileges about higher roles [27]. Here we do not make choices with respect to such constraints.

Administrative privileges are an *infinite* set, even if we assume that the sets of users, roles and user privileges are *finite*. The reason is that administrative privileges over administrative privileges are also administrative privileges. For example, consider the privilege to give someone else the privilege to change the members of a role. The number of *administrative levels* (the number of nestings of the $\Box$ connective to be introduced below) is often restricted in existing literature (sometimes to one [79] or to two levels [93]). We agree that in some settings multiple levels of administration are not useful. In practice the security officer of the organization could fix a bound on the number of administrative levels, depending on the needs of the organization. However, here we prefer to take a general approach, leaving it up to security officers to choose which administrative privileges to use in their systems.

We formalize the full set of privileges by defining a *grammar* that encompasses both user privileges and administrative privileges.

Definition 8 (Privilege Grammar) *Let U, R, P be sets of users, roles and user privileges, the set of all privileges $P \cup P^{\circ}$ is defined by the following grammar:*

$$p \ ::= \ (o,a) \mid \Box(u,r) \mid \Diamond(u,r) \mid \Box(r,r') \mid \Diamond(r,r') \mid \Box(r,p) \mid \Diamond(r,p).$$

where $u \in U$, $r, r' \in R$, and $(o,a) \in P$.

Each administrative privilege corresponds to an administrative command in a straightforward way (see the definition of administrative commands below). For example, the privilege $\Box(u,r)$ allows to add a member u to the role r. The privilege $\Diamond(u,r)$ allows to remove a member u from the role r.

We do not model privileges to change the sets U, R, or P, and we assume that they are chosen sufficiently large and fixed. The rationale is that changes to U, R, or P do not actually change the policy, rather they change which policies are well-formed. For example, in practice the set of users U could be defined as *all strings starting with 'uid'*. Independent of the fact that at any time only a limited number of users is assigned to roles in the RBAC policy.

In the administrative privilege grammar, $\Box$ and $\Diamond$ are connectives and the set P° is infinite. (Even if the sets U, R, P are finite.) For example, one could have an expression $\Box(r, \Box(u, r'))$, which expresses the privilege to give to role r, the privilege to assign user u to the role r'. We can now define administrative policies.

Definition 9 (Administrative Policies) *Let U, R, P be sets of users, roles and user privileges,* an administrative RBAC policy *ϕ is a tuple*

$$\phi = (UA,\ RH,\ PA^\circ),$$

where $UA \subseteq U \times R$ are the user assignments, $RH \subseteq R \times R$ is the role ordering, and $PA^\circ \subseteq R \times P \cup P^\circ$ are the assignments to user or administrative privileges.

The set of administrative policies is denoted $\Phi^\circ_{U,R,P}$, which is a superset of the policy set $\Phi_{U,R,P}$ from standard RBAC (see Definition 7). Let us give an example of an administrative policy.

Example 23 *Let us return to Charlie, the security officer of the previous example. Charlie wants to delegate some of his administrative authority to the employees of the Human Resource department (HR). He would like* HR *to appoint new staff members or nurses directly, without having to refer to Charlie each time. Charlie uses an administrative policy to give administrative privileges to other users. Figure 15 shows the policy: Members of* HR *can assign any user (denoted by $*$) to* staff *and* nurse *roles, and they can revoke users from the nurse role.*

Additionally, to protect the confidentiality of health records in the tables $table_2$, and $table_3$ Charlie has given a revocation privilege about the role $dbusr_3$ to the role $dbusr_1$. The administrative policy hence not only describes who can access which resources, but also which roles have privileges to change the policy itself.

Our model does not require that administrative privileges be assigned to separate parts of the policy. They can be assigned to roles just as ordinary privileges.

Administrative policies allow users to make policy changes. We model this formally by defining a set of *administrative commands.* We write $\blacksquare$ and $\blacklozenge$ to denote assignments and revocations, respectively. As mentioned when introducing our definition of administrative privileges, we do not model changes to the sets U, R, and P, as we assume they are sufficiently large.

Definition 10 (Administrative Commands) *Let U, R, P, be sets of users, roles and user privileges, the administrative commands form a set*

$$\{\blacksquare_u(u', r) \mid \blacklozenge_u(u', r) \mid \blacksquare_u(r, r') \mid \blacklozenge_u(r, r') \mid \blacksquare_u(r, p) \mid \blacklozenge_u(r, p)\}$$

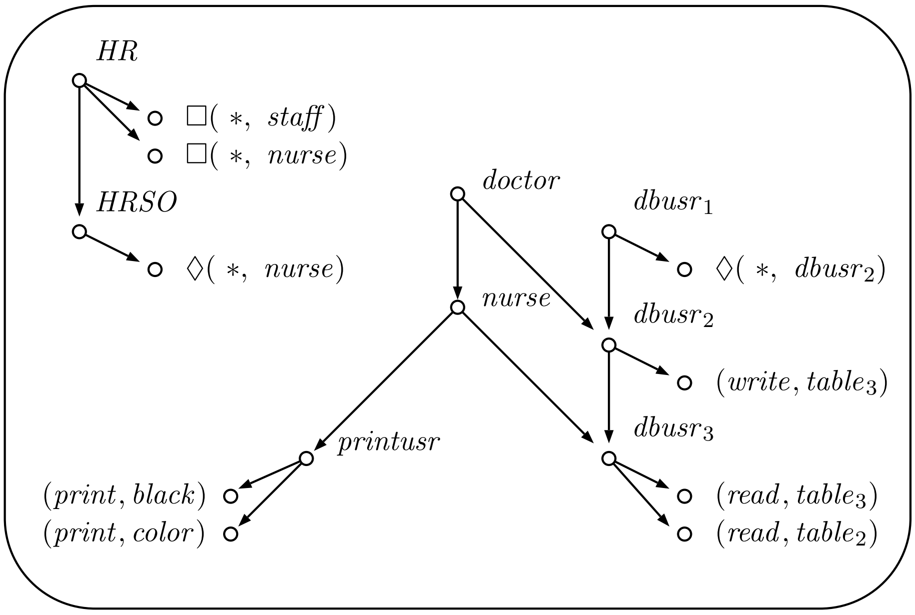


Figure 15: An administrative RBAC policy, deployed by Charlie, the security officer of the hospital.

where $u, u' \in U$*,* $r, r' \in R$*, and* $p \in P \cup P^{\circ}$*.*

For example, the command $\blacksquare_u(u', r)$ denotes that user u assigns user u' to the role r. We illustrate this definition with a basic example: Consider a policy containing the edges (u, r) and $(r, \Box(r, r'))$, as depicted in Figure 16. This policy allows user u to add an edge from role r to role r', i.e. it allows the command $\blacksquare_u(r, r')$.

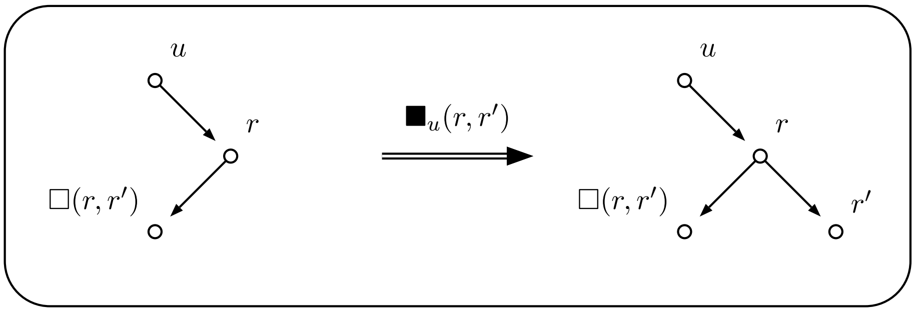


Figure 16: An administrative policy and an administrative action.

A *command queue* is a list of administrative commands. The set of command queues is denoted *CQ*. The empty command queue is denoted ε, and the list constructor is denoted :. The administrative functionality of the RBAC reference monitor is modeled by a command queue, and an administrative RBAC policy. The RBAC reference monitor changes the policy by executing administrative commands in the command queue. We formalize this by a transition function.

Definition 11 (Administrative Transition) *Let $cq \in CQ$ be a command queue, and $\phi \in \Phi^\circ$ an administrative policy, the* administrative transition function*, denoted* $\Rightarrow$: $CQ \times \Phi^\circ \to CQ \times \Phi^\circ$*, is*

$$\begin{array}{rcl} \langle \blacksquare_u(v,v') : cq,\ \phi\rangle & \Rightarrow & \langle cq,\ \phi \cup (v,v')\rangle,\ \textit{if}\ u \to_\phi \Box(v,v'). \\ \langle \blacklozenge_u(v,v') : cq,\ \phi\rangle & \Rightarrow & \langle cq,\ \phi \setminus (v,v')\rangle,\ \textit{if}\ u \to_\phi \Diamond(v,v'). \end{array}$$

If an administrative command is not allowed by the policy ϕ, then the command is removed from the queue, without changing the policy ϕ. Below, a sequence of executions of commands in the queue is called a *run*, denoted by $\Rightarrow^*$.

6.4 Administrative refinement

In existing literature [17, 27, 34, 78, 93], administrative privileges for RBAC policies are treated just like ordinary user privileges (without taking into account that they allow for policy changes). In this section we show that a different approach that is more flexible yet safe. First we formalize the notion of *administrative refinement*. In section 6.4.1 we show that *administrative refinement* yields an ordering for the administrative privileges for assignment ($\Box$), and in section 6.4.2 we show how the privilege ordering can be used practically in an RBAC reference monitor, and we show that the privilege ordering is decidable.

Ignoring administrative policies for the moment, an RBAC policy ψ is safer than a policy ϕ, if ψ grants users fewer privileges than ϕ does. In access control this is sometimes referred to as *refinement* (see for example [14]) . We call this *non-administrative refinement*, to distinguish it from *administrative refinement* to be introduced below.

Definition 12 (Non-Administrative Refinement) *Let $\phi, \psi \in \Phi^\circ$ be two RBAC policies. We say that ψ is a non-administrative refinement of ϕ, denoted $\phi \succeq \psi$, iff for any $v \in U \cup R$ and any* user privilege $p \in P$*, $v \to_\psi p$ implies $v \to_\phi p$.*

We give a basic example to illustrate this definition.

Example 24 (Refinement) *In general, by removing edges from an RBAC policy one obtains a safer policy, hence a refinement. Consider for example the (non-administrative) RBAC policy depicted in Figure 14. By removing Diana from the* staff *role one obtains a refinement.*

There are other more fine-grained type of refinements. For example, if we replace the edge between *Diana* and *staff* with an edge between *Diana* and *nurse*, then one also obtains a refinement according to Definition 12. On the other hand, if we replace the edge between *nurse* and *dbusr1* with an edge between *nurse* and *dbusr2*, we do not obtain a refinement, as nurses get more privileges.

We can now define *administrative refinement*. The goal of an administrative policy is to allow certain policy changes. Basically, an administrative refinement of a policy is a policy that allows *safer* policy changes. Policy changes made by one user may allow other users to make new policy changes, and so on. Therefore, we must take into account which users are performing administrative actions, and in which order. We formalize administrative refinement as follows.

Definition 13 (Administrative Refinement) *Let $\phi, \psi \in \Phi^\circ$ be administrative RBAC policies. We say that ψ is an* administrative refinement *of ϕ, denoted $\phi \succeq^\circ \psi$, if, for any queue of administrative commands $cq \in CQ$, there is a queue of administrative commands $cq' \in CQ$, such that $\phi' \succeq \psi'$, where $\langle cq, \phi\rangle \Rightarrow^* \langle \varepsilon, \phi'\rangle$, and $\langle cq', \psi\rangle \Rightarrow^* \langle \varepsilon, \psi'\rangle$, and cq' is such that, it contains the same number of commands, and the n-th command in cq and the n-th command in cq' is also performed by u, where n ranges over the number of commands in the queue cq.*

The definition states that, if ψ allows a certain policy change then either the same policy change is also allowed by the policy ϕ, or it is a policy change that results in a *safer* policy. It is easy to see that administrative refinement implies non-administrative refinement; take $cq = cq' = \varepsilon$. In other words, if $\phi \succeq^\circ \psi$ holds then also $\phi \succeq \psi$ holds.

Remark 8 (Safety in the HRU model) *In the HRU model [42] it is assumed that there is a group of untrusted users who can collude in any order. Taking into account the order, like we do, is more precise, in the sense that in the HRU model one can not distinguish the policy $\big(lowrole \to \Box(r,p)\big)$ from $\big(highrole \to \Box(r,p)\big)$, while the latter is more safe.*

6.4.1 Ordering administrative privileges

In this section, we introduce a privilege ordering on administrative privileges and then we show that this ordering yields administrative refinements in RBAC. At the end of this section we show how the privilege ordering can be used in practice to allow more flexible RBAC administration.

Consider a simple setting where a sub-administrator has the explicit right to assign a user u to a *high* role in the role ordering. There is no reason to forbid the sub-administrator to assign the user to a lower role. This can be seen as follows. If u becomes a member of the high role, then u

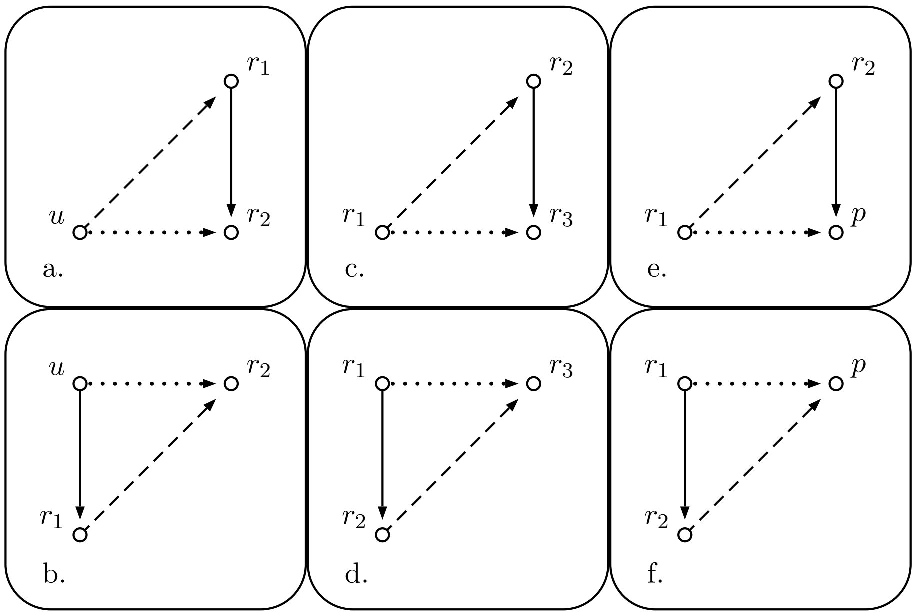


Figure 17: The right to add the dotted edge follows (implicitly) from the right to add the dashed edge. The corresponding ordering relation, given a policy ϕ, is denoted with $\leadsto_\phi$.

can activate also the lower roles and obtain their privileges, as if u was assigned to it explicitly. In existing RBAC literature administrative privileges are not interpreted in this way. For example, in Figure 17a, if the policy includes the edge from r_1 to r_2, that is, r_1 is a higher role than r_2, then the (administrative) privilege to assign user u to role r_1 (the dashed edge) yields the privilege to assign user u to role r_2. It is safer to use the latter privilege instead, because user u will be assigned to a lower role, and hence have fewer privileges. The six figures show the different base cases of the ordering on the administrative (assignment) privileges. Let us define the full ordering relation formally. When one administrative privilege p yields another administrative privilege p', denoted $p \leadsto_\phi p'$, we say that p is *stronger* than p', and p' is *weaker* than p. Intuitively, administrative policies with stronger administrative privileges are less safe, than administrative policies with weaker privileges. We will formalize the ordering relation as follows.

Definition 14 (Privilege Ordering) *Given an RBAC state $\langle UA, RH, PA\rangle$, let q be a privilege in P, and p_1, p_2 privileges in $P \cup P^\circ$, and let r_1, r_2, r_3, r_4 be roles in R. The relation $\leadsto$ is defined as the smallest* *relation satisfying:*

$$
\begin{aligned}
q &\rightsquigarrow_\phi q, && \textit{if } q \in P && (6.1)\\
\Box(u, r_1) &\rightsquigarrow_\phi \Box(u, r_2), && \textit{if } r_1 \rightarrow_\phi r_2, && (6.2)\\
\Box(r_1, r_2) &\rightsquigarrow_\phi \Box(u, r_3), && \textit{if } r_2 \rightarrow_\phi r_3 \textit{ and } u \rightarrow_\phi r_1, && (6.3)\\
\Box(r_2, r_3) &\rightsquigarrow_\phi \Box(r_1, r_4), && \textit{if } r_1 \rightarrow_\phi r_2 \textit{ and } r_3 \rightarrow_\phi r_4, && (6.4)\\
\Box(r_2, r_3) &\rightsquigarrow_\phi \Box(r_1, p_2), && \textit{if } r_1 \rightarrow_\phi r_2,\ r_3 \rightarrow_\phi p_1, \textit{ and } p_1 \rightsquigarrow_\phi p_2, && (6.5)\\
\Box(r_2, p_1) &\rightsquigarrow_\phi \Box(r_1, p_2), && \textit{if } r_1 \rightarrow_\phi r_2 \textit{ and } p_1 \rightsquigarrow_\phi p_2, && (6.6)
\end{aligned}
$$

where p_1 in line 5 is an arbitrary privilege such that $r_1 \rightarrow_\phi p_1$ holds.

The ordering $\rightsquigarrow$ is both reflexive and transitive, because the premises in the definitions of Definition 14. In practice the privilege ordering can be used to allow users, with administrative privileges, to be implicitly authorized for weaker administrative privileges.

It can be shown (see the Theorem 3 below) that by replacing an administrative privilege by a weaker one (with respect to the ordering), one obtains an administrative refinement of the policy. In other words, giving administrative users also the weaker administrative privileges allows them to perform also *safer* administrative operations than the ones originally allowed.

Theorem 3 (Ordering and refinement) *Let $\phi \in \Phi^\circ$ be an administrative policy, let $(r, p) \in \phi$ be a privilege assignment, and let p' be a privilege such that $p \rightsquigarrow_\phi p'$, then the policy $\psi = (\phi \setminus (r, p)) \cup (r, p')$ is an administrative refinement of ϕ, that is $\phi \succeq^\circ \psi$.*

Proof 3 *The proof is by induction over the structure p', i.e. the number of nestings of $\Box$. The base cases are when p' contains one or fewer nestings of $\Box$.*

Let us go over the base cases. If the first rule applies, then p' is a user privilege from P. The theorem holds trivially, since the relation $\succeq^\circ$ is reflexive. For the second rule of Definition 14, take a policy ϕ where $r_1 \rightarrow_\phi r_2$ and $(r, \Box(u, r_1)) \in \phi$ for some role r. Let ψ be the same policy except for the privilege assignment $(r, \Box(u, r_1)) \in \phi$ which is replaced by $(r, \Box(u, r_2))$. We have to show that $\phi \succeq^\circ \psi$: The policy ϕ allows the command

$$\blacksquare_{.}(u, r_1),$$

which changes ϕ to $\phi' = \phi \cup (u, r_1)$, while ψ allows the command

$$\blacksquare_{.}(u, r_2),$$

which changes ψ to $\psi' = \psi \cup (u, r_2)$. To show that $\phi \succeq^\circ \psi$ it suffices to show that $\phi' \succeq \psi'$: In ψ' u has the privileges of r_1, but in ϕ' u has the same

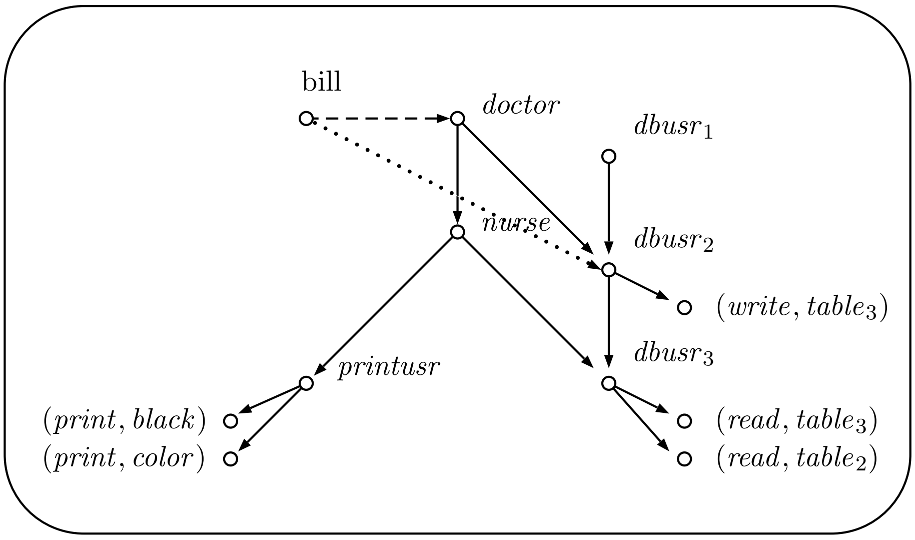


Figure 18: A practical example of the use of administrative refinement.

privileges, due to the edge $r_1 \rightarrow_\phi r_2$. Hence, the policy ψ is a refinement of the policy ϕ. The cases where the rules 3 thru 6 apply are proven similarly.

For the induction step, let p' be an administrative privilege of the form $\Box(r, p''')$ where $p''' \in P \cup P^\circ$ (i.e. a privilege with $n \gg 1$ nestings of $\Box$). Either rule 5 or 6 of Definition 14 can apply. Let us go over the first case (rule 5) here. Take a policy ϕ and privilege p'', such that $r_1 \rightarrow_\phi r_2$, $r_3 \rightarrow_\phi p''$, $p'' \rightsquigarrow_\phi p'''$ and $(r, \Box(r_2, r_3)) \in \phi$. Let ψ be the same policy, except that the privilege assignment $(r, \Box(r_2, r_3)$ is replaced by $(r, \Box(r_1, p'''))$. We have to show that $\phi \succeq^\circ \psi$. The policy ϕ allows the command

$$\blacksquare_.(r_2, r_3),$$

which changes ϕ to $\phi' = \phi \cup (r_2, r_3)$, while ψ allows the command

$$\blacksquare_.(r_1, p'''),$$

which changes ψ to $\psi' = \psi \cup (r_1, p''')$. In the policy ψ' the role r_1 has the privilege p''', while in the policy ϕ' the role r_1 has privilege p''. By induction hypothesis, since p''' is structurally smaller than p' (i.e. with $n-1$ nestings of $\Box$), the policy ψ' is an administrative refinement of ϕ'. The case where rule 6 of Definition 14 applies is proven in a similar way.

Let us give an example of how the privilege ordering can be used in a practical situation.

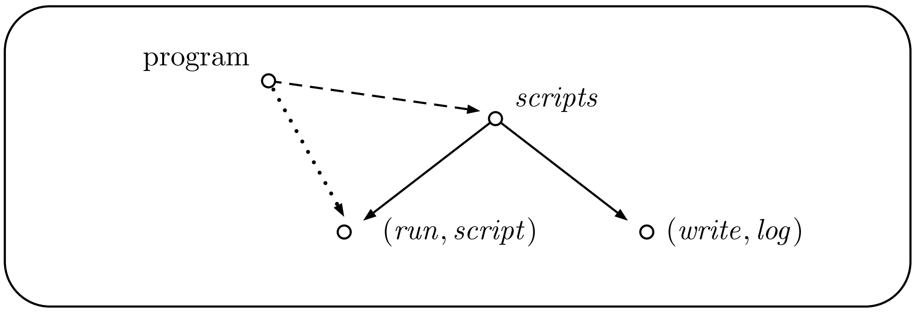
Example 25 (Contractor) *Bill arrives at the hospital and his job is to put some order in the health record database. To do the job he needs to get*

$dbusr_2$ privileges. Anna, an employee of the HR *department, can give the necessary clearance to Bill (and add the dashed edge in Figure 18). However she must urge Bill to apply the principle of least privilege, by activating only the role $dbusr_2$, and not e.g. the doctor or the nurse role, which would yield excessive privileges. But Anna can only hope that Bill does so.*

The privilege ordering implies that Anna can assign Bill directly to the $dbusr_2$ role (the dotted edge in Figure 18) because *of her privilege to assign Jack to the doctor role. In a way, Anna, instead of hoping that BIll applies the principle of least privilege and does not take privileges that are not needed for his duties, she applies it herself.*

Here we have defined a policy to be safer when the policy gives users fewer privileges. The principle of least privilege, and the way it is supported by the RBAC session mechanism, is a well-known example of the usefulness of this definition.

Remark 9 (Non-monotonic design) *One could perhaps argue that there could be practical situations where having fewer privileges is not more safe. For example one could imagine a privilege to append to a log file. Removing this privilege could cause programs to run unsafely, that is without writing logs. We depicted such a policy in the figure below: The administrator can assign some user named program to a role named scripts that has two privileges: for execution of a script and for writing the log. Assigning program directly to $(run, script)$ could cause the program to run unsafely.*



One could address such issues at the application layer, for example by changing the program so that it halts when no logs can be written. Alternatively, one could extend our model to prevent this: Assume we mark *this privilege. The definition of the privilege ordering can now be extended with a side-condition that takes these marks into account, such that the privilege $(run, script)$ cannot be assigned separately.*

Remark 10 (Ordering revocation privileges) *Our privilege ordering does not include revocation privileges, $\Diamond$. Unlike with assignment privileges, safety considerations can not be used to establish refinements of revocation* *privileges. From the point of safety, there is no need to even model revocation privileges. Every revocation makes the policy more safe. From the viewpoint of availability, however, revocations privileges should not be given to all users. So one could use availability consideration to extend the privilege ordering also to revocation privileges. We sketch the idea briefly:*

From the viewpoint of availability revoking all edges directed to a vertex v is equivalent to removing all edges directed from the vertex v. Therefore, if a user can revoke all edges directed to a vertex v, then she can also remove any of the edges directed from the vertex v. Vice versa, if a user can revoke all edges directed from a node v, then she can also revoke any of the edges directed to the vertex v.

We do not pursue this idea further, because it leads to a definition based not on the presence of certain edges (assignments) in a policy, but on the absence *of edges. This would have a profound impact on our model. RBAC does not allow to specify the absence of an assignment. Also checking that assignments are absent requires special care (particularly in distributed settings), and finally it is not clear how future assignments should be dealt with.*

6.4.2 Decidability

In this section we address an important practical issue. We prove that the ordering relation (Definition 14) is decidable. Since the full set P of privileges is infinite, this result is not immediate. For instance, a naive forward search does not necessarily terminate (see the example at the end of this section). The proof indicates how a decision algorithm, deciding which privileges are to be given to which roles, can be implemented at an RBAC reference monitor.

Lemma 1 (Decidability of the Ordering Relation) *Given an RBAC state S, and two privileges p, p', it is decidable whether $p \rightsquigarrow p'$.*

Proof 4 *The proof is by structural induction over p', i.e. the number of nestings of $\square$.*

The base cases are when p' has one or fewer nestings of $\square$. We show that for these base cases $p \rightsquigarrow p'$ is decidable:

- *Either p' is a user privilege from P. In this case $p \rightsquigarrow p'$ holds only when $p = p'$ (see rule (1) in Definition 14).*

- *Or p' is of the form $\square(u, r)$ in which case rules (2) and (3) must be checked, which both have finite premises.*

- *Or p' is of the form $\square(r, r')$, in which case the rule (4) must be checked, which has finite premises.*

- *Or p' is of the form $\Box(r, q')$ where $q' \in P$, in which case rules (5) and (6) apply. In both cases the premises are finite, because for a $q' \in P$, $q \rightsquigarrow q'$ only holds for $q = q'$.*

For the induction step, suppose that p' is $\Box(r', p''')$, for some role r' and privilege p''' with $n-1$ nestings of $\Box$. Now, $p \rightarrow p'$ can only hold if the premises of rule (5) or (6) hold. In both cases, the premises are decidable, either because they are finite, or because the induction hypothesis is applicable, since in $p'' \rightarrow p'''$, p''' is structurally smaller than p'.

Let us show how the proof above can be used in practice, as a procedure for checking whether one privilege is weaker than another.

Example 26 (Using the privilege ordering) *Consider Example 25 again. Can Anna assign Bill to the dbusr$_2$ role? We should decide whether*

$$\Box(\mathit{bill}, \mathit{doctor}) \rightsquigarrow \Box(\mathit{bill}, \mathit{dbusr}_2).$$

We have to check that the role doctor inherits the privilege $\Box(\mathit{bill}, \mathit{dbusr}_2)$. Using the second line of Definition 14, we find that this is the case.

To give a more involved example, we suppose that the hospital's security officer Charlie has the privilege $\Box(\mathit{doctor}, \Box(\mathit{doctor}, \mathit{dbusr}_2))$. Can Charlie give doctors the privilege $\Box(\mathit{doctor}, \mathit{dbusr}_3)$? We have to check whether

$$\Box(\mathit{doctor}, \Box(\mathit{nurse}, \mathit{dbusr}_2)) \rightsquigarrow \Box(\mathit{doctor}, \Box(\mathit{doctor}, \mathit{dbusr}_3)).$$

This is indeed the case by using line (6) first, and then line (2).

Now, for the sake of exposition, let us remove the edge between the doctor and the role dbusr2 . Let us show how to determine that the previous relation does not hold: Now only line (6) can apply. So we must decide whether $\Box(\mathit{nurse}, \mathit{dbusr}_2) \rightsquigarrow \Box(\mathit{doctor}, \mathit{dbusr}_2)$. This is a base case of the induction described in the proof of Lemma 1: Only lines 2, 3, and 4 remains to be checked and than we can conclude that it does not hold.

It could be useful to find *all* the privileges p' weaker than a given p. Perhaps surprisingly, in some cases the set of all privileges p' weaker than a given privilege p, is infinite. (In fact, our proof of decidability of $p \rightsquigarrow p'$ is over the structure of p'.) Let us give an example.

Example 27 (Infinitely many weaker privileges) *Consider a policy where $(r_2, \Box(r_1, r_2)) \in PA$. We should stress here that this is not an artificial, or peculiar policy: Members of r_2 can make members of r_1, member of r_2 too.*

Suppose we are interested in finding all the privileges weaker than $\Box(r_1, r_2)$. The first weaker privilege we discover by applying rule (2) in definition 14:

$$\Box(r_1, \Box(r_1, r_2)).$$

Using this result in rule (3), we find another weaker privilege,

$$\Box(r_1, \Box(r_1, \Box(r_1, r_2))),$$

and we can use this again in rule (3), and so on.

The outer nesting in the last term in the this example is in a sense redundant. Instead of assigning the privilege $\Box(r_1, r_2)$ to r_1 directly, one assigns the privilege to do so, to r_1. This only requires the users in role r_1 to perform an extra administrative step, which is cumbersome for the users in r_1, does not provide any safeguards. For all practical purpose one could stop after N_{RH} applications of rule (3), where N_{RH} is the length of the longest chain in RH. Informally, the argument goes as follows. Consider a privilege of the form $\Box(r, \Box(r', \Box(r'', p)))$. If r is a higher role than r' (and not the same role) then the privilege expression is practically useless. Otherwise it is not (as discussed). The number of useful nestings is hence at most N_{RH}.

The computational complexity of making an access control decision based on a RBAC policy ϕ depends on the structure of ϕ. To enable fast access control decisions in a practical application, regardless, it would be convenient to be able to calculate for each role r all the privileges p for which $r \rightsquigarrow_\phi p$, beforehand. However as shown in the previous example, this is in principle impossible. A possible solution to this problem is to find all T pairs v, v' such that $v \rightarrow_\phi v'$, and use the T pairs to make the actual access control decision, by applying Definition 14.

6.5 Related Work

The problem of administration of an RBAC system was first addressed by Sandhu et al. [79]. Later, numerous articles have been published extending or improving the administration model proposed there [17, 27, 34, 78, 87, 92, 93]. We discuss the seminal works.

Barka et al. [17] distinguish between original and delegated user role assignments. Delegations are modeled using special sets, and different sets are used for single step and double step delegations (which must remain disjoint). A function is used to verify if membership to a role can be delegated. Privileges can also be delegated, provided they are in the special set of delegatable privileges belonging to the role. In their work, each level of delegation requires the definition of tens of sets and functions, whereas in our model administrative privileges, of an arbitrary complexity, are simply assigned to roles, just like the ordinary privileges. The PDBM model [93] defines a cascaded delegation. This form of delegation is also expressible in our grammar (by nesting the $\Box$ connective). In the PDBM model, however, each delegation requires the addition of a separate role, which is cumbersome given the fact that there are already many roles to manage. In our model the administrative privileges are assigned to roles just like the ordinary privileges. It is not required to add any additional roles.

A number of proposals define general constraints on the administrative privileges. For example, the constraint that a user must first have a privilege, before being allowed to delegate it to other users. Note that, as mentioned earlier, here no particular choice is made with respect to such constraints. Zhang et al. [92] implement rule based constraints on delegations. They demonstrate their model using a Prolog program. Basically, they analyze the properties of a centralized RBAC system, focussing on so-called *separation of duty* policies. Crampton [27] defines the concept of administrative scope. Basically a role r is in the scope of a role r' if there is no role above r' that is not below r. They show how administrative scope can be used to constrain delegations to evolve in a natural progression in the role hierarchy. Bandman et al. [16] use a general constraint language to specify constraints on who can receive certain delegations.

6.6 Conclusions

The issue of designing policy administration mechanisms for RBAC has received considerable attention recently [17, 27, 34, 78, 93]. With our model we contribute to these lines of research. We introduce the notion of administrative refinement of policies, and we show how it can be used to allow more flexible management of the RBAC policy. Concretely, our contribution is the definition of a general class of administrative policies, and a formal definition of administrative refinement. We have shown that there is a natural ordering for administrative privileges which yields administrative refinements of policies, and we have given the proof that this ordering is decidable. We also showed how useful our extension is in practice. Our approach allows administrative users to be implicitly authorized for weaker administrative operations, which is thus more flexible and more safe as well.

Revocation privileges are included in our model, but we have not identified (yet) a separate ordering for revocation privileges (as discussed in the remark at the end of Section 6.4.1). We believe that this is an interesting possibility for further research.

Chapter 7

RBAC Administration in Distributed Systems

7.1 Introduction

In the previous chapter we have proposed an administrative model for the ANSI RBAC standard. In this chapter we address the issue of administration of RBAC policies in a distributed system.

Large and distributed information systems are increasingly common. For example, most large hospitals run a variety of systems that process medical data. Access control is needed across these systems because (by law) health records must be protected from unauthorized access. RBAC is often used in such settings, and is designed to simplify the specification of access control policies. Still, practice has pointed out that - in many organizations - RBAC policies can become large, involving hundreds of roles [34], sometimes as many roles as users. This makes administration of an RBAC-based system a difficult task.

Consider for example a hospital with a distributed system composed of various subsystems that store and process confidential medical data. Both *safety*, and *availability* are key in this setting. Let us suppose that the hospital's security officer, to fulfill data-protection requirements, has deployed a set of different RBAC policies at different subsystems in the hospital. Over time some of these policies need to change. For example, a nurse may need to be assigned to a new role because of a changed hospital shift, or a database role may need to get access to additional tables, because some database application changed. Now who can make the policy changes? Which subsystems need to update their access control policies following policy changes? How can the update of the various subsystems take place efficiently? How can multiple administrative systems be used concurrently?

Although administration of RBAC has received considerable attention recently, and numerous researchers have proposed different ways of choosing administrative RBAC policies [17, 26, 27, 3, 34, 60, 78, 87, 88, 92, 93], there is no literature on the more practical issue of administration of a distributed RBAC system. The RBAC standard does not address this either. We propose a model for the administration of a distributed RBAC system and we show how it can be translated to a practical implementation.

- We define a distributed system model with a central administrative system, while the objects and the RBAC policy are distributed across the different subsystems. A key component of our model is a *mapping* based on the fact that different subsystems protect different subsets of data, and that therefore only some policy changes are relevant to certain subsystems. We use this in Section 7.2 to define formal *safety and availability requirements* for the administration of an RBAC policy across the subsystems.

- We derive an administration procedure, which is efficient in the sense that subsystems are only updated about *relevant* policy changes, and correct in the sense that it preserves the formal safety and availability requirements (Section 7.3).

- We translate the administration procedure to practical *pseudo-code* to demonstrate how it can be implemented (Section 7.3.1).

- We show how our model can be extended with *multiple* administrative subsystems (Section 6) and we sketch the additional steps that are required here. This addresses advanced settings with for example a human resources system for assigning users to roles, and a database management system (DBMS) for assigning database privileges to database roles.

7.2 Distributed System Model

In this section we define a basic distributed system model. The model consists of a central administrative subsystem and a number of non-administrative subsystems. In section 7.4 we show how it can be generalized to a system with multiple administrative subsystems.

We restrict our attention to *General Hierarchical RBAC* model, as defined in the ANSI RBAC standard, extended with the general class of administrative policies (as defined in Sections 6.2, and 6.3 in the previous chapter).

Consider a heterogeneous distributed system composed of databases, file systems and so one, like one may find in organizations such as hospitals. In such a distributed system it is not convenient to use a central reference monitor to decide all user access requests, as each request would involve contacting the central reference monitor creating a bottleneck and a single point of failure. On the other hand, when each subsystem has its own reference monitor, and a separate policy, then one needs to manage those policies consistently. For example, if a user is assigned to the role of employee, all subsystems in the organization should allow the user to use the privileges of the employee role and vice versa when a user is revoked from a role. Inconsistencies in the definition of roles across the distributed system cause confusion and may affect safety and availability of data across the system.

So one could argue that a procedure is needed that maintains exact copies of a single system-wide policy at the different subsystems, and that updates all the subsystems after each policy change. At the same time, it is unnecessary to send updates to, say, a printer about changed database table privileges, particularly given the fact that in practice RBAC policies can be large and policy changes frequent [34]. Additionally, for example in health care, policy definitions may even be *privacy-sensitive*. For example, a policy that states that a specialist on skin diseases has full access to a health record of a certain patient, reveals a lot of information about the patients health. In this section we define a distributed system model, and basic safety and availability requirements, allowing us to derive a more efficient administration procedure (in Section 7.4).

The privilege mapping pm is a key component of our model, allowing us to capitalize on the fact that different subsystems offer access to different (largely disjoint) subsets of the resources, and avoid excessive updates about irrelevant policy changes. Formally it is defined as follows.

Definition 15 (Privilege Mapping) *The* privilege mapping*, denoted pm, is a mapping $pm : S \to \mathcal{P}(P)$. We say that subsystem s* protects *object o, if $(a, o) \in pm(s)$.*

The privileges in $pm(s)$ are referred to as the *relevant user privileges* for subsystem s. We do not require that $pm(s)$ and $pm(s')$ are disjoint for different subsystems s and s' (see also remark 11). The privilege mapping can be used as a tool for the security officer to evaluate and implement policy changes. Let us give a practical example.

Example 28 (Distributed hospital system) *A hospital has a network consisting of a database named Sqil, a medical system Sqan, a printer Inq, and an administrative system HSO for administrative tasks, such as policy changes. The system is depicted in Figure 19, where the dots denote ordinary users of the system.*

The hospital's security officer enforces a number of RBAC policies across the different subsystems that protect resources such as an electronic health record table of Sqil denoted ehrtable*, and a scan job of Sqan called* job*. The*

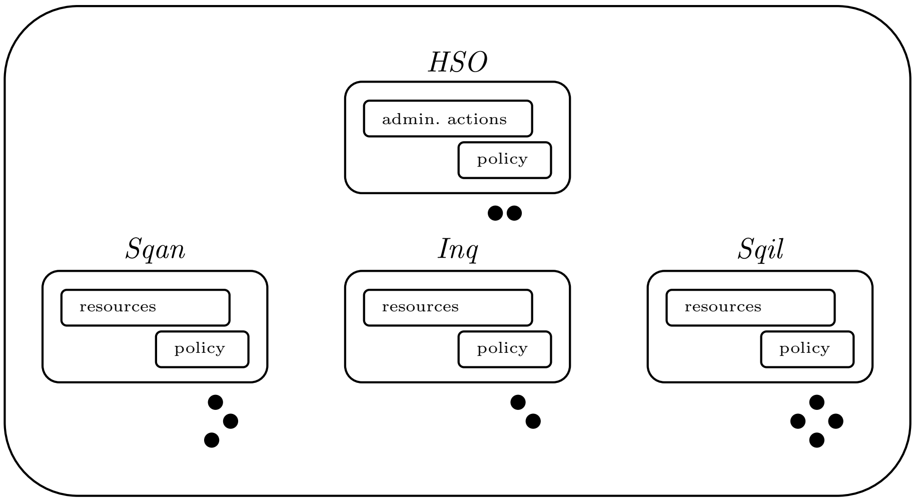


Figure 19: A hospital's distributed system, where the dots are users, and *HSO* is the system used by the hospital security officers to issue policies.

hospital's security officer has defined the following privilege mapping:

$$\begin{aligned} pm(\mathit{Sqil}) &= \{(\mathit{ehrtable}, \mathit{view}), (\mathit{ehrtable}, \mathit{insert})\} \\ pm(\mathit{Sqan}) &= \{(\mathit{job}, \mathit{halt}), (\mathit{job}, \mathit{start})\} \\ pm(\mathit{Inq}) &= \{(\mathit{black}, \mathit{print}), (\mathit{color}, \mathit{print})\} \end{aligned}$$

Basically, for each subsystem, pm() gives the list of privileges that are relevant for that subsystem, i. e. regarding actions and object protected by that system.

In the rest of this chapter we will use the privilege mapping to define formal properties, and also an administrative procedure, to deal with policy deployment, and policy changes. We will assume, in our running example, that the security officer has defined such a mapping. One could argue that in some settings it is cumbersome to keep track of which objects are present on which systems. On the other hand, in many practical situations, the object mapping *follows* largely from the names of the objects. Privileges to read ehrtable1, ehrtable2, and so on, would all map to the database system *Sqil*, while privileges to start and halt job1, job2, and so on, would all map to the medical system *Sqan*.

Remark 11 (Privilege mapping) *The privilege mapping could map some of the objects to multiple subsystems. For instance when the same resource is present at multiple subsystems, such as in the case of a cluster of databases that duplicate certain tables.*

There may exist practical settings where it is difficult to keep track of which objects or resources reside on which subsystems, for instance because

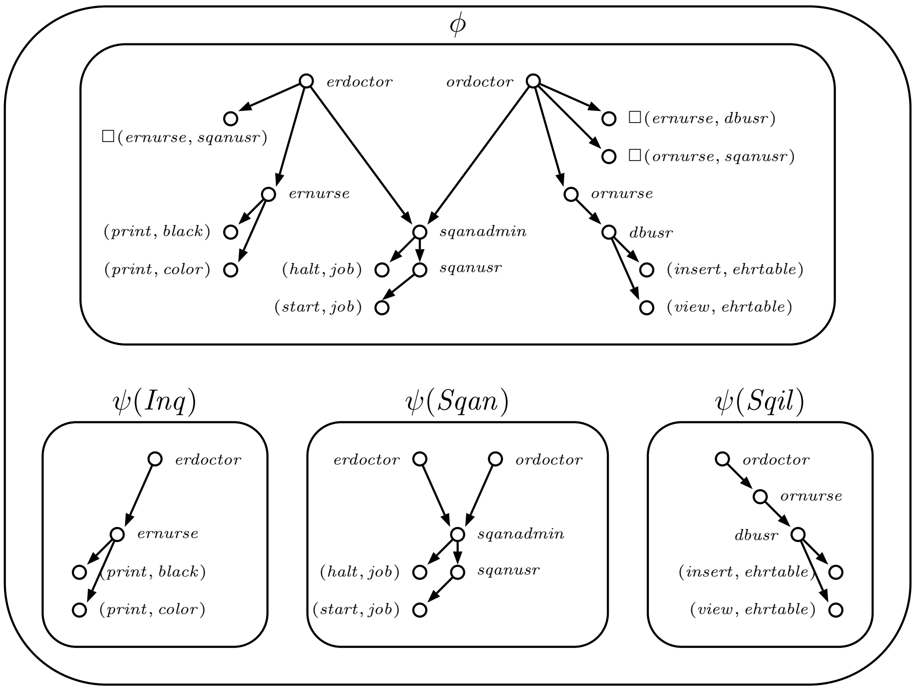


Figure 20: Sound and complete policy distribution.

users can move them freely from one subsystem to another. In such settings it may be more convenient, but in principle less precise, to map the resources to all the subsystems they may be moved to.

Our model of a distributed system comprises a set of systems (each with a reference monitor), denoted S and ranged over by $s_0, s_1, \ldots$, and a single administrative system, denoted s_a, from which (administrative) users can make policy changes. The policy of the administrative system is denoted ϕ, while the policy of subsystem s is denoted $\psi(s)$, where $\psi : S \to \Phi$ defines the distribution of policies across the subsystems.

Definition 16 (Distributed RBAC System) *A* distributed system *is a tuple*

$$(S,\ pm,\ \phi,\ \psi\ ),$$

where S is a set of systems, $pm : S \to \mathcal{P}(P)$ is the distributed system's privilege mapping, $\phi \in \Phi^{\circ}$ is an administrative policy, and $\psi : S \to \Phi$ is a function that maps each subsystem to a non-administrative policy.

Using the privilege mapping we can define two formal requirements for the distribution of policies across the subsystems.

The first is the basic requirement that the vertices in each of the subsystem's policies are included in the policy of the administrative system. This is in the interest of safety and administration, so that the security officer, or some other user of *HSO*, in Example 28, can correctly assess the impact of policy changes. The second captures that the relevant parts of the administrative system's policy should be present at the subsystems. This is in the interest of availability of resources, so that, as in Example 28, if the security officer has given to users the privilege to access a resources, then they are also granted access by the subsystem.

Definition 17 (Soundness and Completeness) *Given a distributed system* $(S,\ pm,\ \phi,\ \psi)$*, we say that* ψ *is* sound *with respect to the central policy* ϕ*, if*

$$\bigcup_{s \in S} \psi(s) \subseteq \phi.$$

On the other hand, we say that the distribution ψ *is* complete *with respect to the central policy* ϕ *if and only if for any subsystem* $s \in S$*, and any privilege* $p \in pm(s)$

$$u \rightarrow_{\phi} p \textit{ implies } u \rightarrow_{\psi(s)} p.$$

Soundness is important from the viewpoint of safety: it ensures that subsystems grant access only when it is allowed by the administrative policy ϕ. It may seem that a weaker requirement suffices: *For any* $s \in S$*, and any privilege* $p \in pm(s)$*,* $u \rightarrow_{\psi(s)} p$ *implies* $u \rightarrow_{\phi} p$*.* However, such a weak requirement would complicate the implementation of policy changes, as will become clear in the next section.

Completeness, on the other hand, is important from the viewpoint of availability: it ensures that the subsystem protecting object o grants access to the object o, whenever it is allowed by the administrative policy. Before defining an administration procedure that implements policy changes, preserving soundness and completeness (in the next section) let us introduce the running example of this chapter, and demonstrate the practical usefulness of the definitions above.

Example 29 (RBAC policies in the hospital) *Let us continue in the setting of Example 28. The security officer has defined a number of roles for staff and nurses of the operation room (OR)* $(\textit{orstaff}, \textit{ornurse})$*, and staff and nurses of the emergency room (ER)* $(\textit{erstaff}, \textit{ernurse})$*. For the sake of brevity we do not elaborate on the exact names of these users, but refer to the roles they are a member of.*

The security officer has prepared the distributed system as shown in Figure 20. The policy ϕ *is the (administrative) policy of system* HSO*, and the policies* $\psi(Sqil)$*,* $\psi(Sqan)$*, and* $\psi(Inq)$ *are the (non-administrative) policies of the database Sqil, the medical system Sqan, and the printer Inq.*

The distribution ψ *is sound with respect to the central policy* ϕ*, because each subsystem policy is a subset of the administrative policy* ϕ*. It is also* *clear that the distribution ψ is complete. So although the subsystems in the hospital do not enforce the hospital's policy in its entirety, this does not affect availability nor safety of the resources. Indeed, most parts of the hospital policy are* in practice *irrelevant for the printer Inq, as Inq does not protect database tables of Sqil, nor the resources of Sqan. Also the administrative privileges $\Box(.,.)$, and $\Diamond(.,.)$ are irrelevant for the subsystems, because they can not be used for administrative actions anyway. These subsystems implement standard RBAC policies from Φ.*

Soundness and completeness are, so to speak, the minimal requirements that must be fulfilled. The *largest* distribution ψ, that is both complete and sound, is the distribution where $\psi(s) = \phi$ for all $s \in S$, where all subsystems have the same policy. We can also define the *smallest* policy distribution that satisfies soundness and completeness.

Definition 18 (Upper and Lower Closure) *The* upper closure *of a vertex v in ϕ, denoted, $(\uparrow_\phi v)$, is $\{(v', v'') \in \phi \mid v'' \rightarrow_\phi v\}$, and the* lower closure *of a vertex v in ϕ, denoted, $(\downarrow_\phi v)$, is $\{(v', v'') \in \phi \mid v \rightarrow_\phi v'\}$.*

The smallest distribution ψ that is sound and complete is such that for every subsystem $s \in S$, the following holds,

$$\psi(s) = \bigcup_{p \in pm(s)} (\uparrow_\phi p).$$

We call this the *lean* distribution. The *lean* policy distribution has the advantage that components of the distributed system have the parts of the policy that are strictly necessary to decide about allowing or denying user actions.

7.3 Centralized Administration

In the previous section we have specified formal requirements for the distribution of RBAC policies across a distributed system (see Definition 17). In this section we define an administration procedure for the administrative reference monitor s_a that *preserves* these requirements.

We model the system s_a by defining a command queue, containing administrative commands ($\blacksquare$, or $\blacklozenge$) for policy changes, and *message commands.* The message commands are needed to model the propagation of policy changes across the subsystems. They form a set $\{\oplus_s(\delta), \ominus_s(\delta)\}$, where $s \in S$ denotes the recipient subsystem and $\delta \in \Phi$ is a policy, and $\oplus$ denotes addition and $\ominus$ denotes removal. Let us first sketch the procedure by giving an example: a user u of the administrative system s_a places the administrative command $\blacksquare_u(r, r')$ in the queue. The administrative subsystem processes it by (1) checking that ϕ allows u to make this policy change, (2) changing its policy ϕ to $\phi \cup (r, r')$, (3) replacing the administrative command with message commands $\oplus_s\big((r, r') \cup (\uparrow_\phi r)\big)$ for each subsystem $s \in S$ that has relevant privileges in the lower closure of r' in ϕ, and (4) processing the message commands. In the sequel we will show that sending $\big((r, r') \cup (\uparrow_\phi r)\big)$ suffices to preserve completeness. Moreover we will show that the procedure preserves soundness. Let CQ denote the set of all command queues. We define the administration procedure by a formal transition function.

Definition 19 (Distributed Administration) *Given a distributed system* $(S,\ pm,\ \phi,\ \psi)$, *let* $cq \in CQ$ *be a command queue, and* N *be the number of systems in* S. *The transition function* $\Rightarrow$: $CQ \times \Phi^\circ \times (\Phi)^N \rightarrow CQ \times \Phi^\circ \times (\Phi)^N$, *is defined as follows.*

$\langle cq,\ \phi,\ \psi\rangle \ \Rightarrow\ \langle cq',\ \phi',\ \psi'\rangle$ *holds when:*

if $cq = [\blacksquare_u(v, v') : cq'']$ *and* $u \rightarrow_\phi \square(v, v')$, *then*
 $cq' = [\oplus_{s_1}\big((v, v') \cup (\uparrow_\phi v)\big) : \cdots : \oplus_{s_k}\big((v, v') \cup (\uparrow_\phi v)\big) : cq'']$,
 where $\{s_1, \ldots, s_k\}$ *are all the subsystems with relevant privileges in the lower closure of* v', *that is*
 $\{s_1, \ldots, s_k\} = \{s \in S \mid \textit{if } \exists p \in pm(s).v' \rightarrow_\phi p)\}$.
 $\phi' = \phi \cup (v, v')$, *and* $\psi' = \psi$.

if $cq = [\blacklozenge_u(v, v') : cq'']$ *and* $u \rightarrow_\phi \lozenge(v, v')$, *then*
 $cq' = [\ominus_{s_1}\big((v, v')\big) : \cdots : \ominus_{s_k}\big((v, v')\big) : cq'']$,
 where $\{s_1, \ldots, s_k\}$ *are all the subsystems.*
 $\phi' = \phi \setminus (v, v')$, *and* $\psi' = \psi$.

if $cq = [\oplus_s(\delta) : cq'']$, *then* $cq' = cq''$,
 $\phi' = \phi$, $\psi'(s) = \psi(s) \cup \delta$ *and* $\psi'(s') = \psi(s')$ *for* $s' \neq s$.

if $cq = [\ominus_s(\delta) : cq'']$, *then* $cq' = cq''$,
 $\phi' = \phi$ *and* $\psi'(s) = \psi(s) \setminus \delta$ *and* $\psi'(s') = \psi(s')$ *for* $s' \neq s$.

The messages are the smallest when the command is a user assignment, since the upper closure of a user is always empty. This is also the most frequently used administrative command [34]. Message commands following an assignment ($\blacksquare$) only involve those subsystems that are 'affected' by it. Revocations ($\blacklozenge$) on the other hand are broadcast to all subsystems to ensure soundness of ψ.

One could make this procedure even more efficient by keeping a *history* of sent policy definitions per subsystem, to avoid sending revocations of definitions to subsystems that never received them (or to avoid sending the same policy definitions twice). We do not go into details about this, for the sake of brevity. Although it is common in literature on distributed systems to use an *expiration mechanism* to reduce the number of revocations [75], we refrain from going into details about time or expiration here, because we believe they are out of the scope of the RBAC standard. We would like to mention however that, because soundness and completeness are preserved

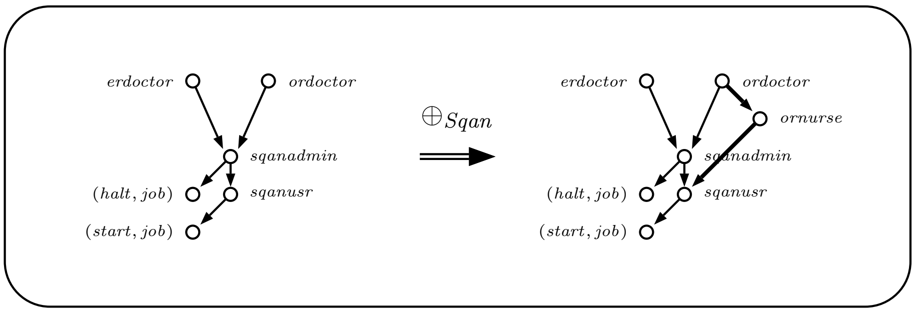


Figure 21: Update for subsystem *Sqan*.

when edges expire, it seems straightforward to add expiration to our model. The procedure preserves soundness and completeness, but it does not preserve leanness, for example. To preserve leanness subsystems could remove irrelevant parts of the subsystem's policy, independently of the administrative reference monitor. System s can check for each edge (v, v') in $\psi(s)$ whether or not $v' \rightarrow_{\psi(s)} p$ for a relevant privilege $p \in pm(s)$.

Let us return to our running example to demonstrate a practical instance of the administration procedure.

Example 30 (Administration in the hospital) *Suppose Bob, a member of* orstaff*, wants to grant all members of* ornurse *the right to use the medical system Sqan, say for a new type of operation. To do so, Bob puts an administrative command in the queue of the administrative system HSO. He is allowed to do so, by the administrative policy ϕ in Figure 20.*

The administrative system HSO now takes the following steps: The command in the queue is

$$\blacksquare_{Bob}(ornurse, sqanusr).$$

After executing this command, the new policy ϕ' contains the new edge $(ornurse, sqanusr)$ and the command on the queue is replaced by the message command

$$\oplus_{Sqan}\big((ornurse, sqanusr) \cup (\uparrow_\phi ornurse)\big).$$

The message command is executed, updating also the policy of Sqan. The new policy for Sqan includes the upper closure of ornurse, i.e. the new edge $(ornurse, sqanusr)$, as well as the 'members' of ornurse. *So Bob's administrative command changes the policies ϕ and $\psi(Sqan)$, but not the policies of Inq or Sqil. The policy changes corresponding to Bob's action are depicted in Figure 21 by dashed edges.*

The administration procedure preserves soundness and completeness. It does so without sending irrelevant parts of ϕ to subsystems. We denote a sequential execution of administrative commands (a *run*) by $\Rightarrow^*$ and an empty queue by ε.

Theorem 4 (Changes preserve soundness and completeness) *Let $(S,\ pm,\ \phi,\ \psi\ )$ be a distributed system. For any command queue $cq \in CQ$ that contains only administrative commands (of the form $\blacksquare_{.}(.,.)$, or $\blacklozenge_{.}(.,.)$), the run to an empty queue*

$$\langle cq,\ \phi,\ \psi\ \rangle \Rightarrow^{*} \langle \varepsilon,\ \phi',\ \psi'\ \rangle,$$

yields a policy ϕ' and a distribution ψ' for which the following statements hold:

1. *If ψ is sound with respect to ϕ, then also ψ' is sound with respect to ϕ'.*
2. *If ψ is complete with respect to ϕ, then also ψ' is complete with respect to ϕ'.*

Proof 5 *We have to show that an arbitrary queue of administrative commands preserves soundness and completeness. We prove the result by induction on the number of commands in the queue. Let us sketch the proof briefly. We assume that the distribution ψ is initially sound and complete with respect to ϕ.*

The base case (the empty queue) is trivial. The induction hypothesis is that soundness and completeness are preserved by queues with n commands, and we show that this holds for queues with $n+1$ commands. Consider a queue containing $n+1$ commands. We enumerate the different possibilities for the first command in the queue.

- *If the first command is of the form $\blacksquare_u(v, v')$, and it is replaced by the message commands $\oplus_{s_1}\big((v,v') \cup (\uparrow_\phi\ v)\big) : \ldots : \oplus_{s_k}\big((v,v') \cup (\uparrow_\phi\ v)\big)$ on the queue, where $\{s_1, \ldots, s_k\} = \{s \in S \mid \textit{if}\ \exists p \in pm(s).v' \rightarrow_\phi p)\}$. The administrative policy is changed to $\phi'' = \phi \cup (v, v')$ (cf. the first item in Definition 19), and after processing the message commands, the distribution changes to ψ''.*

  Soundness *follows since the difference between $\psi(s)$ and $\psi'(s)$ is at most $(v, v') \cup (\uparrow_\phi v)$, which is a subset of ϕ'. The remaining queue is shorter and preserves soundness by induction hypothesis.*

  Completeness *follows trivially for subsystems outside $\{s_1, \ldots, s_k\}$, as the policy change does not affect the upper closure of their privileges. The other subsystems are complete before the change, so they already have the upper closure up till v_2. The update message adds to this also the rest of the upper closure (v_1, v_2), the new edge, and $(\uparrow_\phi v_1)$.*

  The rest of the queue preserves completeness by induction hypothesis.

- *If the first command is of the form $\blacklozenge_u(v, v')$, and it is replaced by the message commands that remove the edge (v, v') from all systems in S.*

Soundness *is straightforward, since the distribution was sound before processing this command, and the edge (v, v') is removed from all the policies of the subsystems.*

For completeness *observe that ψ is initially complete with respect to ϕ, and that by removing an edge the upper closure only shrinks.*

By the induction hypothesis, both soundness and completeness are also preserved by the commands on the remaining (shorter) queue.

This completes the proof.

7.3.1 Implementation

The formal procedure of Definition 19 can be translated into a decision procedure. In this section we describe procedures, by using pseudo code, both for the administrative reference monitor and for the non-administrative reference monitors of the subsystems.

Let us introduce the syntax of the code. Vertices `v1, v2,...` (users, roles, and privileges) are assumed to be unique strings, and edges are pairs of such strings `(v1, v2)`. Policies are represented as *lists* of edges. Below the expression `a in b` checks whether `a` is in the list `b` or not. The functions `add`, `remove`, and `join` denote adding, and removing elements from lists, and joining two lists, respectively, and `[]` denotes the empty list.

We use sub-procedures for finding the upper and lower closure (see Definition 18) of a vertex in a policy, for later use. The function `lower(a, b)` returns a list of elements from the policy `a` which are in the lower closure of `b`, i.e. $(\downarrow_a b)$. Both `lower` and `upper`, its converse, are implemented by a basic depth-first search.

```
procedure lower(policy, v1, visited)
  visited := add(visited, v1).
  list l1 := [].
  for (v1,v2) in policy and v2 not in visited
    list l2 := dfs(policy, v2, visited).
    l1 := join(l1, l2).
  return with l1.
```

The procedure looks for edges starting at `v1`, and follows each of them downwards. To avoid running in circles we mark which vertices where visited already. The upper closure `upper` is the same procedure except for inverting the direction of the edges of `policy`.

Administrative commands are expressed as triples: `(user, action, (v1, v2))`, where the second parameter is either ■ or ♦, and `(v1, v2)` is the edge being assigned or revoked. Queues are lists of administrative commands. We use the operation `shift` that returns and removes the first element of the queue.

The main procedure for the administrative system takes a policy and a queue of commands and returns the policy that results from applying the commands in the queue. It can be written as follows.

```
procedure admin (policy, queue)
  if queue= []
    return with policy.
  endif
  (user, action, (v1, v2)) := shift(queue).
  list lowu := lower(policy, user).
  if action = ■ and □(v1,v2) in second(lowu)
    list uppv1 := upper(policy, v1).
    list lowv2 := lower(policy, v2).
    list dest := [].
    for priv in second(lowv2)
      for s in systems
        if pm(s, priv) and s not in dest
          dest := add(dest, s).
        endif
    for s in dest
      send(s, ⊕, add(uppv1, (v1, v2))).
    return with admin(add(policy, (v1,v2)), queue).
  endif
  if action = ♦ and ♢(v1,v2) in second(lowu)
    for s in systems
      send(s, ⊖, [(v1,v2)])
    return with admin(remove(policy, (v1,v2)),queue).
  endif
    return with admin(policy, queue).
```

Let us explain the procedure in detail. In case the action is ■ it is checked whether or not the user is allowed to perform that command. This is only true when the corresponding privilege □(v1,v2) is in the lower closure of `user`. The function `second`, used here, takes a list of pairs, and returns a list of the second element of every pair. The next step takes care of sending the proper update messages. The list of subsystems is denoted `systems`, and the privilege mapping is a function `pm` that takes a privilege and a system name as input and returns true if the privilege is a relevant privilege for the system. The lower closure of `v2` is used to select which list of subsystems `dest` will receive a message (denoted by `send`). The upper closure of `v1`, on the other hand, constitutes the contents of the update message (cf. Definition 19). The steps for ♦ can be explained in the same way. The procedure recurs through the queue, untill it returns the new administrative policy, or if no command was allowed the same administrative policy.

The procedure for the non-administrative system is more simple. There are two types of commands: A message command by an administrative system, denoted by `receive`, and a basic user command by a user who wants to perform an action on an object, denoted by `do`.

```
procedure subsystem(policy, queue)
  if queue= []
    return with policy.
  endif
  shift(queue) := cmd.
  if cmd = receive(⊕, delta)
    return with subsystem(add(policy, delta),queue).
  endif
  if cmd = receive(⊖, delta)
    return with subsystem(remove(policy, delta), queue).
  endif
  if cmd = (user, action, object)
    list lowu := lower(policy,user).
    if (action,object) in second(lowu)
      do(action, object).
    endif
  endif
  return with subsystem(policy,queue).
```

Note that in this procedure the lower closure of `user` in `policy` is calculated at every user access request. This may be time-consuming (each search involves $O(E)$ steps, where E is the number of edges in $\psi(s)$). One could instead calculate the full transitive closure for the policy once, and update it only when update messages arrive.

7.4 Decentralized Administration

In the previous sections we have modeled systems with a single administrative reference monitor. In this section we show how our model can be extended to deal with systems with multiple administrative reference monitors.

In practice, administrative actions (e.g. assigning a user to a role) are relatively rare; for instance, they are less frequent than ordinary user actions (e.g. accessing a database table). This suggests that for many practical distributed systems a single administrative reference monitor should be sufficient. Still, there may be scenarios where multiple administrative monitors

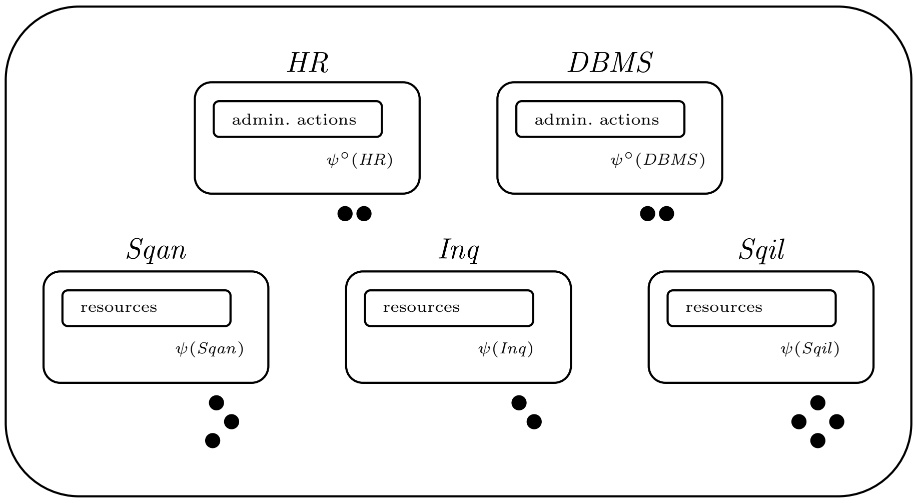


Figure 22: Decentralized administration in a hospital.

are needed. For example when two different organizations share a common infrastructure, and at the same time prefer to use their own separate administrative systems. In this setting, one could use identical administrative subsystems augmented with standard mutual exclusion techniques to coordinate policy changes. More challenging are the settings in which the administrative reference monitors are not identical (i.e. with different administrative policies), for instance because they are not equally trustworthy.

In this section we briefly describe the additional steps needed to extend our distributed model to these settings, and we define an additional requirement for the distribution of policies across the administrative systems. The extension, although not entirely straightforward, makes use of the same formal structure of the previous section.

Let us assume that – in addition to the set of ordinary subsystems S – there exist a set of administrative reference monitors S° (ranged over by $s_a, s_b, \dots$), and an administrative privilege mapping

$$pm^\circ : S^\circ \to \mathcal{P}(P^\circ)$$

As before, we say that $p \in P^\circ$ is a *relevant administrative privilege* for system s, if and only if $p \in pm^\circ(s)$. The mapping pm° corresponds to the intuitive idea that certain administrative reference monitors can only be used for certain administrative actions. A distributed system is defined as a tuple

$$(S^\circ,\ S,\ pm^\circ,\ pm,\ \psi^\circ,\ \psi),$$

where $\psi^\circ : S^\circ \to \Phi^\circ$ is the distribution function of the administrative policies across the administrative reference monitors. Let us see an example (refer to Figure 22): the system *HR* is used at the human resources department for changing user-role assignments, while the system *DBMS* is used at the hospital's data center for changing database privileges. Here, there are multiple distinct administrative subsystems, and multiple distinct non-administrative subsystems, and there is no central administrative system.

Let us define ϕ by

$$\phi = \bigcup_{s \in S^\circ} \psi^\circ(s) \cup \bigcup_{s \in S} \psi(s).$$

The policy ϕ is here no longer the policy of a central administrative system (as in the previous sections), but only an abstract notion of the full system-wide policy. Like before, the subsystems each hold parts of ϕ.

Let us now define requirements for ψ and ψ° concerning safety and availability of objects, similar to the ones proposed in Section 7.2.

The safety requirement (soundness) remains the same, but the availability requirement (completeness) becomes more complex. The policy of an administrative reference monitor should be (1) complete for its relevant administrative privileges, and – in addition – (2) it should contain the parts of ϕ needed to produce the message commands described in Section 7.3.1. We call the first *standard completeness* and the second *administrative completeness*. Let us show a basic example.

Example 31 (Standard and administrative completeness) *Consider an administrative system* $s_b \in S^\circ$*, with* $\Box(v, v') \in pm^\circ(s_b)$*, and* $(r, \Box(v, v')) \in \phi$*.*

- *Standard completeness with respect to the privilege* $\Box(v, v')$ *requires that* ψ *contains* $(\uparrow_\phi r)$*.*
- *Administrative completeness additionally requires that* ψ *contains* $\big((\uparrow_\phi v) \cup (\downarrow_\phi v')\big)$*, i.e. the parts of* ϕ *needed to perform the message commands described in Section 7.3.*

We call $\big((\uparrow_\phi v) \cup (\downarrow_\phi v')\big)$, in the example, the *policy support* of the administrative privilege $\Box(v, v')$. Basically, the policy support of an administrative privilege ensures that the administrative subsystem can perform administrative operations, *and* propagate the relevant additional parts of ϕ to subsystems. Let us show an example of policy support.

Example 32 (Policy support) *Figure 23 shows the policy support of two administrative privileges.*

The edges in the dark gray areas form the policy support of the privilege $\Box(ernurse, dbusr)$*, and the light gray areas the policy support of the privilege* $\Box(ornurse, sqanusr)$*.*

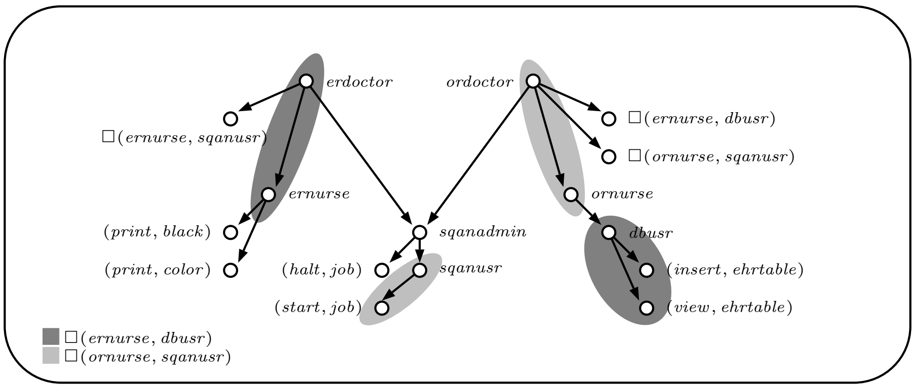


Figure 23: Policy support for different administrative privileges.

It is now clear how a 'correct' and 'efficient' administration procedure can be defined in this distributed model. It depends on the administrative privilege mapping pm° as follows: an administrative subsystems s_a must send an update to an administrative subsystem s_b every time the policy support for relevant administrative privileges of s_b changes.

7.5 Related Work

The administration of RBAC policies is an issue that attracts considerable attention from the research community. In particular, there is a large body of literature on how to choose administrative policies (informally, about which roles should get what authority to change the RBAC policy) [17, 26, 27, 3, 34, 60, 78, 87, 92, 93]. The considerations that motivate the choices in these proposals are diverse. Crampton and Loizou, for example motivate their choice by considering responsibility in a organization hierarchy [27], whereas Li and Mao consider for example flexibility, and psychological acceptability [60]. None of these proposals address how to distribute RBAC policies across a distributed system, in a *correct* and *efficient* way (which is the scope of this chapter). We now consider some of these proposals in more detail.

Wang and Osborn [88] introduce administrative domains for *role graphs*, a class of RBAC policies with a single lowest and single highest role, called *minrole* and *maxrole* respectively. Each administrative domain is defined by one role, and contains all the roles below it, except *minrole*. Administrative domains may not overlap, unless one domain includes the other completely. Wang and Osborn justify this restriction by arguing that it should not be allowed for different domain administrators to make changes to the same roles. On the other hand they also stress that this is a disadvantage of their model, arguing that in practice one would like to have overlapping domains, for example when one resource is shared by different departments (see Figure 20). Implementation of RBAC in distributed systems is not at the basis of this choice. For example, the administrative policy depicted in Figure 20 is not admitted by the administrative domains model of Wang and Osborn. In their model, administrative privileges about the role *sqanusr* can only be assigned to a *domain administrator*, which also has administrative privileges about *ehrstaff* and *orstaff*. Although we agree that there may be practical settings where administrative domains may be useful, we do not adopt such restrictions here.

Closely related is the work by Crampton and Loizou [27], who define the concept of *administrative scope*. Basically a role r is in the scope of a role r' if there is no role above r that is not comparable to r'. They show how administrative scope can be used to constrain delegations to evolve in a natural progression in the role hierarchy. Administrative scopes can be used as a basis for a policy distribution, but this does not yield the sound and complete administration procedures defined in our model.

Similarly, in the ERBAC model, proposed by Kern et al. *scopes* are used to define over which RBAC objects and administrator has authority [54]. The ERBAC model focuses on managing RBAC policies in a commercial enterprise security software. Although ERBAC has been verified against a business case involving multiple remote company sites, the main goal of the administrative component of ERBAC is to allow for delegation of administrative authority, and does not deal with the issue of distributing (parts) of RBAC policies in a proper way.

Li and Mao design three main requirements (flexibility and scalability, psychological acceptability, economy of mechanism) and analyze them in different existing administrative models, and they design UARBAC, a new family of models. As mentioned earlier, none of the above-mentioned models address the issue of distributing (administrative) policies across a distributed system.

Somewhat related to our work is the paper by Bhamidipati and Sandhu which discusses how RBAC can be used in a number of different architectures with multiple servers in a network [20]. They focus on the capabilities of the servers, specifically on whether or not RBAC is *supported*, and treat the role-hierarchy as a central service. In our model on the other hand we distinguish which policy changes are relevant, and we do not update subsystems about irrelevant policy changes. dRBAC is a decentralized trust management and access control mechanism for systems that span multiple administrative domains [36]. It is targeted at settings where independent organizations form dynamic coalitions. Similar settings are also addressed by the TM models to be discussed below. In dRBAC, local policies in one administrative domain can be used in another domain; on the other hand, dRBAC does not address the issue of distributing policies (efficiently) across systems within an administrative domain.

Role-based Trust Management (TM) [63] and distributed certificate systems, such as SDSI [75], are remotely related lines of research. In these systems, a number of agents exchange *security statements* and may create hierarchies similar to those used in RBAC. Issuing TM credentials corresponds to administrative commands in RBAC. In TM however it is generally assumed that users are free to utter security statements, while the focus is on whether to *trust* such statements (which involves some trust calculation by the receiver of such statements). In RBAC this assumption is inappropriate, because policy changes are explicitly guarded by administrative privileges. The central issue in this chapter is the issue of ensuring that users can perform the actions they are allowed to, without broadcasting the entire security policy, has also been researched in TM. In some TM models the user is expected to collect the credentials needed for access by itself. In others *credential chain discovery* algorithms are used, which are automatic procedures to retrieve missing credentials. Here we describe a model that prevents situations where policy definitions must be retrieved ad-hoc by a subsystem, by pushing them to the interested subsystems upon the issuing of policy changes.

7.6 Conclusions

Despite a large body of literature on the administration of RBAC policies [27, 34, 78, 88], there is no proposal for RBAC administration in distributed systems, although these models are concerned with decentralizing authority, which can be used in a distributed setting. In this chapter we propose a model for the implementation of a common RBAC standard in a distributed system. We focus on the formal requirements for such implementation, and we propose an administration procedure for the deployment of policy changes across the distributed system, which is efficient and preserves the formal requirements. A key part of our model is a privilege mapping, which captures the intuitive idea that different systems protect different objects. To demonstrate how the procedure can be implemented in practice, we translate our procedure to practical pseudo-code, and finally we also indicate how to extend our model to cover settings with multiple administrative systems.

Chapter 8

Concluding Remarks

In this thesis we address the question of designing a flexible access control system suitable for dynamic collaborative environments. We take two different approaches. In the first part of this thesis we have proposed a new access control framework AC^2, and we have proposed extensions to an existing access control framework (RBAC) in the second part of this thesis. Each of the chapters include detailed conclusions on the specific techniques of that chapter. In this last chapter we summarize our contributions, comparing both approaches, and we look to forward to how our ideas might be put into practice in the near future.

8.1 Contributions

Let us summarize the contributions in this thesis.

In Chapter 2 we propose a new access control framework called Audit-based Compliance Control (AC^2). AC^2 is targeted at dynamic collaborative environments, such as consultancy firms, or hospitals, where a small group of users exchange, modify and refine a large number of documents and policies (as shown in the examples in Chapters 2, 4, and 5). Unlike conventional access control, we assume that no reference monitor is present to *prevent* unauthorized actions, but that user actions are logged and that users can be asked to justify their actions, *a-posteriori* - just like in society. Our framework uses a simple, but expressive, policy language which is based on first-order predicate logic, extended with an *owns* predicate, a *maySay* predicate, and constructs for pre- and post-obligations. To reason about policies AC^2 uses a formal proof system, for which prove an important logical property: Cut-elimination (Chapter 3. In Chapter 3 we show an implementation of the proof system by programming a proof checker (using Twelf) and a proof finder (using Prolog).

In the second part of this thesis we take a more evolutionary approach to

	AC^2	$RBAC^\circ$
Expressivity	$+$	$-$
Decidability	$-$	$+$
Policy administration	$+$	$-$
Policy distribution	$+$	$-$
Policy enforcement	$+$	$-$
Adoption	$-$	$+$

Table 8.1: Comparison between AC^2 and $RBAC^\circ$.

answering the question of designing a flexible access control system for dynamic collaborative environments, by proposing two extensions for standard RBAC. RBAC is a common access control standard, and is implemented or supported in many systems. Let us briefly go over the conclusions of the second part of our thesis. In Chapter 6 we propose a general class of administrative policies, and a formal definition of administrative refinement. We show that there is a natural ordering for administrative privileges which yields administrative refinements of policies, and that this ordering is decidable. In Chapter 7 we define a model for the administration of RBAC standard in a distributed system, formal safety and availability requirements, and we propose administrative procedures for the deployment of policy changes across distributed systems.

8.2 Comparison

Let us compare the two different approaches, by comparing the main features and drawbacks of the two systems: Expressivity (of the policy language), decidability (of the policy language), policy administration, policy distribution, policy enforcement, and (ease of) adoption. For the sake of brevity, we refer to the extensions proposed in Chapter 6 and Chapter 7 by $RBAC^\circ$. We summarize the comparison in Table 8.1, and go over the comparison in more detail below.

- **Expressivity:** The policy language of AC^2 is an extension of first-order logics, with the *owns* predicates, the connective *maySay* for administrative privileges, and constructs for pre- and post- obligations. Standard RBAC on the other hand is a subset of propositional logic, and allows only policies of the form $user \rightarrow role$, $role \rightarrow role'$, or $role \rightarrow (a, o)$, where a denotes an action, and o denotes an object. $RBAC^\circ$ extends this with administrative privileges, but their expressive power is limited compared to the administrative privileges in AC^2. In fact all policies in $RBAC^\circ$ can be expressed in the AC^2 policy language.

- **Decidability:** As mentioned above, AC^2 is more expressive than RBAC°. This comes at a price. The policy language of AC^2 is only semi-decidable, while the language of RBAC° is decidable. The proof finding procedure for AC^2 (in Section 3.3) is in fact more complicated than the procedure for RBAC°(in Section 6.4.2). The reason is that while AC^2 is based on first-order logic, RBAC° is based on propositional logic. Although, as we have argued in Chapter 2 semi-decidability of AC^2 is not problematic in the setting of AC^2 it is certainly a drawback.

- **Policy administration:** Let us now focus on policy administration, and on how administrative privileges are expressed and obtained. Both AC^2 and RBAC° contain (connectives for) administrative privileges. In AC^2 administrative privileges are expressed using *maySay*, while in RBAC° administrative privileges are expressed using $\Box$ (*assign*). Both AC^2 and RBAC° support administrative refinement, which allows administrators or users to assign a small set of administrative privileges, from which more (weaker) administrative privileges can be derived. The main difference is that while *maySay* can be used with any AC^2 policy ($maySay(a, b, \phi)$ is the privilege for a to say ϕ to b, where ϕ is an arbitrary AC^2 policy), the $\Box$ construct only allows to assign users to roles, roles to roles, or roles to privileges.

  Another difference is that in RBAC° administrative privileges are initially assigned by the system administrator, while in AC^2 administrative privileges can also be derived from the *owns* predicate. Now although in some settings, for example in a hospital, the advantage of this derivation may be limited (doctors do not own the health records). This extra option may provide an important advantage in settings where users create documents often - such as in research and consultancy firm.

- **Policy distribution:** In AC^2 policy distribution is completely decentralized: policies are sent by users to other users, using the action `comm`, and they are provided to auditors by the users, during audits. In RBAC°, on the other hand, users do not need to collect and present (parts of) policies. Users may even be ignorant of the policies that apply, because policies are distributed to and processed by the reference monitor(s), in a centralized (Chapter 6) or decentralized fashion (Chapter 7). In RBAC users only carry authentication credentials. At first sight this may seem more easy for the users. On the other hand, collecting the appropriate (parts of) policies may be difficult, especially in dynamic collaborative environments across organization boundaries when many different policies may apply to an object (or a user action). AC^2 on the other hand is more flexible and allows a mixed set-up where the auditor collects some general policies beforehand, and requires the user to provide only the missing parts.

- **Policy enforcement:** In AC^2 policy enforcement is based on a-posteriori compliance audits. Auditors check logs of the user actions, and they can decide to ask the user for justification proofs. The auditor checks a-posteriori whether there are policies that justify the user's action. Inappropriate behavior of users is hence only deterred, but not prevented. In RBAC$^\circ$, on the other hand, policy enforcement is done in a conventional a-priori way: Prior to granting the action the RBAC reference monitor checks its policy to see whether the action should be allowed. Logs of user actions are only checked manually by system administrators to see whether or not the reference monitor has been working correctly. Checking a-posteriori gives more flexibility to users, and it allows to cope with unexpected circumstances more easily.

- **Adoption:** In AC^2 users can behave badly. This is very different from RBAC$^\circ$ where the reference monitor *prevents* users from behaving badly. In fact a crucial requirement to deploy AC^2 successfully is that the users actions are logged, and that the users can be held accountable for their actions. As discussed in Chapter 2, these requirements exclude certain settings. RBAC$^\circ$ can be deployed for instance in more open settings, where users can not be held accountable for their actions.

  Moreover, RBAC is already used in many commercial, and open source products, while AC^2 is a new access control framework. The RBAC$^\circ$ extensions could be deployed on top of standard RBAC, making existing RBAC systems more flexible.

Having compared the two approaches let us briefly discuss how the two approaches could be combined. RBAC is already widely deployed, and supported in existing systems and applications, and RBAC$^\circ$ policies can be expressed using the AC^2 policy language. The main drawback of RBAC$^\circ$ is that it does not allow a-posteriori enforcement of policies. This makes it difficult to use RBAC$^\circ$ in dynamic collaborative environments. It would be interesting to see whether in closed settings the a-priori enforcement that is typical of RBAC$^\circ$ could be removed partially or temporarily, and replaced by an AC^2 auditing procedure. Recently it has been suggested that such an approach would allow to implement RBAC policies in a dynamic service-oriented architecture (SOA) [55]. We foresee a number of issues that would have to be resolved here: 1) User actions would have to be logged. 2) Users would have to be held accountable for their actions, and 3) RBAC$^\circ$ policies would have to be known to the users and systems that are performing the actions, a-priori. The first two issues seem relatively easy to solve: As mentioned in Chapter 2, user actions are often logged already, and users are often held accountable for their actions. The third issue may be more challenging: While general usage policies are often communicated to users (say employees or doctors), the details of RBAC role and privilege assignments are usually hidden from the users. Using an AC^2 audit procedure in an RBAC system would require a kind of proof finder that explains to the user which RBAC° role or privilege assignments may be missing, or needed to justify a certain action. This seems to be an interesting direction for future research.

8.3 Outlook

Let us conclude by looking at some recent developments that show how our ideas might be put into practice in the near future.

- It has been argued that AC^2 has advantages over RBAC for implementing access control in the recently built Service Oriented Architecture (SOA) of a Dutch insurance company [55]. Koot states that, because the SOA contains systems and data from so many different organizations, RBAC is difficult to implement. The reason is that all the different organizations issue different RBAC policies, that change often and unexpectedly. Keeping the different policies realigned is a difficult task. Koot argues that, since the users and systems in the SOA are authenticated and to some extent trusted, AC^2 could provide a better solution to enforce detailed policies based on roles. We look forward to seeing whether this would simplify the implementation of access control in this setting.

- Recently it has become clear that a simple form of a-posteriori access control will be used in the future Dutch electronic health record system (AORTA) [46]. AORTA allows doctors and other health care professionals to find and access health records of patients, through a national register of health records. The register consists of pointers to health records at the various medical subsystems used across the country. A central node in AORTA provides access to health records, logging, and it checks if requests where issued by doctors (who all have an access card, to sign requests). All the doctors are expected to only access health records when this is allowed by the existing medical regulations. Technically speaking there are no access control policies in place that enforce all or parts of these regulations. The reason is that the data exchange within and across medical organizations is too complex, and unpredictable, to allow a successful deployment of a conventional (a-priori) access control. AORTA, however, does feature an audit log, where access to health records is logged, and doctors can be held accountable a-posteriori, for justifying their actions. Also patient consent is still implemented in a very rudimentary way: The Dutch minister of Health, Welfare and Sports has recently asked patients to opt-in or out of the national register, where opt-out means that the record will not be listed in the register. The AORTA specifications mentions more detailed policies, such as 'Only access for doctor X', or 'Notify patient before each access', but they will not be implemented in the first releases, because they are technically challenging. Flaws in the policy enforcement could have dire consequences. We believe that when more detailed privacy policies will be introduced in future AORTA releases, then automated audit procedures and machine-readable justification proofs (like those used in AC^2) could be helpful to enforce policy compliance. Automated audit procedures would allow the auditors to automatically check a large number of user actions for compliance, ask the users involved for further justifications. AC^2 would provide a way to check compliance against detailed access control policies, without running the risk that health records are unavailable due to outdated or incorrect policies.

- The Dutch government has recently built an online repository, called GMV, for authorizations, where citizens and enterprises can store delegations for transactions [45]. GMV holds triples of the form (a, s, b), where a and b are names of citizen or enterprise, and s the name of a service such as *tax declaration 2008*. For example a may be citizen Alice, and b accountant Bob who helps out with Alice's tax declarations. The GMV triple is later retrieved by the system providing service s at the moment that b requests s on behalf of a. The semantics of a GMV triple is as follows: a says b can consume s on behalf of a. While GMV is currently only being used for a couple of basic permissions for services, the future releases will probably contain sets of permissions (roles) like *all tax declarations*. Let us see how this relates to the AC^2 framework introduced in Chapter 2: The authorization triples in GMV could be mapped to AC^2 policies by the conclusion derivation function yielding the permission $s(a)$. Let $s'(x)$ denote the more general permission to do all tax declaration on behalf of citizen x. In AC^2 this would be expressed as $\forall_x.s'(x) \rightarrow s(x)$. AC^2 can also be used to model how the repository decides to accept a triple or not: The policy $\phi = \forall_{x,y}.maySay(x, y, s'(x))$ is the permission for users to authorize other for their tax declarations s' on their behalf. In AC^2 $maySay(()a, b, s(a))$ can be derived from ϕ using the refine rule. In this setting the audit procedure of AC^2 could be used to check that b was allowed to request service s on behalf of a. First the system providing s could be audited, and in this step the auditor would obtain the triple (a, s, b). Then the auditor could check whether a was allowed to issue the triple. In this step the auditor would check the more general policy ϕ, and the evidence that a said $s(a)$ to b. After these two steps the auditor would have established that the action by b complies to the policies. If not b can be held accountable for his actions.

We believe that there are still other settings where our ideas could be put into practice in the near future. There are many settings where detailed access control policies must be enforced, but where conventional access control mechanisms are cumbersome to setup and use. To name a couple, in dynamic military coalitions, and in large enterprises that have to adhere to SOX legislation. We look forward to seeing future application of our ideas, and we look forward to future research into flexible access control for dynamic collaborative environments.

Bibliography

Author references - refereed

[1] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, and J. I. den Hartog. An audit logic for accountability. In W. Winsborough and A. Sahai, editors, *Proc. of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 34–43. IEEE Computer Society Press, 2005.

[2] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini. Audit-based compliance control. *International Journal of Information Security (IJIS)*, 6(2-3):133–151, 2007.

[3] M. A. C. Dekker, J. Cederquist, J. Crampton, and S. Etalle. Extended privilege inheritance in RBAC. In R.H. Deng and P. Samarati, editors, *Proc. of the Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 383–385. ACM Press, 2007.

[4] M. A. C. Dekker, J. Crampton, and S. Etalle. RBAC administration in distributed systems. In *Proc. of the ACM Symposium on Access control models and technologies (SACMAT)*, pages 93–102. ACM Press, 2008.

[5] M. A. C. Dekker and S. Etalle. Refinement for administrative policies. In W. Jonker and M. Petkovic, editors, *Proc. of the International Workshop on Secure Data Management (SDM)*, volume 4721 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin, 2007.

[6] M. A. C. Dekker, S. Etalle, and J. I den Hartog. Privacy policies. In W. Jonker and M. Petkovic, editors, *Security Privacy and Trust in Modern Data Management*, pages 383–398. Springer Berlin, 2007.

[7] M. A. C. Dekker and Sandro Etalle. Audit-based access control for electronic health records. *Electronic Notes in Theoretical Computer Science*, 168:221–236, 2007.

[8] M. A. C. Dekker, P. J. M. Veugen, and S. Etalle. Audit-based access control voor de zorg. *het vakblad Informatiebeveiliging*, 2007.

Other references

[9] M. Abadi. Logic in access control. In P. G. Kolaitis, editor, *Proc. Symposium on Logic in Computer Science (LICS)*, pages 228–233. IEEE Computer Society Press, June 2003.

[10] A. Anderson. Comparison of two privacy languages: EPAL and XACML. Sun Technical Report TR-2005-147, 2005.

[11] RBAC Standard, ANSI INCITS 359-2004, 2004.

[12] A. W. Appel and E. W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proc. of the Conference on Computer and Communications Security (CCS)*, pages 52–62. ACM Press, 1999.

[13] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. Enterprise privacy authorization language (EPAL 1.2) - W3C member submission 10 november 2003.

[14] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-P3P privacy policies and privacy authorization. In P. Samarati, editor, *Proc. of the Workshop on Privacy in the Electronic Society (WPES)*, pages 103–109. ACM Press, 2002.

[15] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In D. Gollmann and E. Snekkenes, editors, *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 162–180. Springer Berlin, 2003.

[16] O. L. Bandmann, B. Sadighi Firozabadi, and M. Dam. Constrained delegation. In M. Abadi and S. M. Bellovin, editors, *Proc. of the Symposium on Security and Privacy (S&P)*, pages 131–140. IEEE Computer Society Press, 2002.

[17] E. Barka and R. S. Sandhu. Framework for role-based delegation models. In J. Epstein, L. Notargiacomo, and R. Anderson, editors, *Annual Computer Security Applications Conference (ACSAC)*, pages 168–176. IEEE Computer Society Press, 2000.

[18] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In R. Focardi, editor, *Proc. of the Computer Security Foundations Workshop (CSFW)*, pages 139–154. IEEE Computer Society Press, 2004.

[19] B. Beckert and J. Posegga. leantap: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995.

[20] V. Bhamidipati and R. Sandhu. Push architectures for user role assignment. In *Proc. of the National Information Systems Security Conference (NISSC)*, pages 89–100, 2000.

[21] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the Symposium on Security and Privacy (S&P)*, pages 164–173, 1996.

[22] S. Byers, L. F. Cranor, and D. Kormann. Automated analysis of P3P-enabled web sites. In *Proc. of the International Conference on Electronic Commerce (ICEC)*, pages 326–338, 2003.

[23] J. Cattlet. Open letter to P3P developers, 1999.

[24] C. Conrado, M. Petkovic, M. van der Veen, and W. van der Velde. Controlled sharing of personal content using digital rights management. In E. Fernández-Medina, J. C. Hernández, and L. J. García, editors, *Proc. of the International Workshop On Security in Information Systems (WOSIS)*, pages 173–185, 2005.

[25] R. Corin, S. Etalle, J. I. den Hartog, G. Lenzini, and I. Staicu. A logic for auditing accountability in decentralized systems. In T. Dimitrakos and F. Martinelli, editors, *Proc. of the IFIP Workshop on Formal Aspects in Security and Trust (FAST)*, volume 173, pages 187–202. Springer Berlin, 2004.

[26] J. Crampton and H. Khambhammettu. Delegation in role-based access control. In D. Gollmann and A. Sabelfeld, editors, *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, LNCS, pages 174–191. Springer, Berlin, 2006.

[27] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information System Security (TISSEC)*, 6(2):201–231, 2003.

[28] L. Cranor, M. Langheinrich, and M. Marchiori. A P3P preference exchange language 1.0 (APPEL 1.0), 2002.

[29] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P 1.0) specification - W3C recommendation 16 april 2002. http://w3.org/TR/P3P, 2002.

[30] J. DeTreville. Binder, a logic-based security language. In M. Abadi and S. M. Bellovin, editors, *Proc. of the Symposium on Research in Security and Privacy (S&P)*, pages 105–113. IEEE Computer Society Press, 2002.

[31] E. M. Szabo, editor. *The Collected of works Gerhard Gentzen*. North Holland, 1969.

[32] S. Etalle and W. H. Winsborough. A posteriori compliance control. In *Proc. of the 12th ACM Symposium on Access control models and technologies*, pages 11–20, New York, NY, USA, 2007. ACM.

[33] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proc. of the NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[34] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based Access Control*. Computer Security Series. Artech House, 2003.

[35] H. Foekema and C. Hendrix. Fouten worden duur betaald. TNS NIPO rapport B5561, 2004.

[36] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. In *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, 2002.

[37] D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. A linear logic of authorization and knowledge. In *Proc. of the European Symposium On Research In Computer Security (ESORICS)*. Springer Berlin, 2006.

[38] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Proc. of the Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 2006.

[39] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. In P. Syverson, editor, *Proc. of the Computer Security Foundations Workshop (CSFW)*, pages 111–122. IEEE Computer Society Press, 1999.

[40] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In R. Focardi, editor, *Proc. of the Computer Security Foundations Workshop (CSFW)*, pages 187–201. IEEE Computer Society Press, 2003.

[41] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *ACM*, 40(1):143–184, 1993.

[42] M. A Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Comunications of the ACM*, 19(5), 1976.

[43] H. Hochheiser. The platform for privacy preference as a social protocol: An examination within the U.S. policy context. *ACM Transactions on Internet Technology (TOIT)*, 2(4):276–306, 2002.

[44] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn. Assessment of access control systems - NIST interagency report. Technical report, National Institute of Standards and Technology, 2006.

[45] Stichting ICTU. Gemeenschappelijke machtigings- en vertegenwoordigingsvoorziening (GMV), 2008.

[46] Nationaal ICT Instituut in de Zorg (NICTIZ). Technische architectuur AORTA. www.aortarelease.nl, 2008.

[47] S. Jajodia, S. Gadia, and G. Bhargava. Logical design of audit information in relational databases. In D.Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security: An integrated Collection of Essays*, pages 585–595. IEEE Computer Society Press, 1995.

[48] S. Jajodia, M. Kudo, and S. Subrahmanian. Provisional authorization. In *Proc. of the International Workshop on Security and Privacy in E-Commerce (WSPEC)*, 2000.

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

[50] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In J. Peckham, editor, *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 474–485. ACM Press, 1997.

[51] T. Jim. SD3: A trust management system with certified evaluation. In R. Needham and M. Abadi, editors, *Proc. of the Symposium on Security and Privacy (S&P)*, pages 106–115. IEEE Computer Society Press, 2001.

[52] G. Karjoth, M. Schunter, and E. Van Herreweghen. Translating privacy practices into privacy promises -how to promise what you can keep. In *Proc. of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 135–146. IEEE Computer Society Press, 2003.

[53] G. Karjoth, M. Schunter, and M. Waidner. Platform for enterprise privacy practices: Privacy-enabled management of customer data. In R. Dingledine and P. F. Syverson, editors, *Proc. of the International Workshop on Privacy Enhancing Technologies (PET)*, Lectures in Computer Science, pages 69–84. Springer Berlin, 2002.

[54] A. Kern, A. Schaad, and J. Moffett. An administration concept for the enterprise role-based access control model. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–11, New York, NY, USA, 2003. ACM Press.

[55] R. Koot. ABAC, meer ideetjes. *het vakblad Informatiebeveiliging*, 2007.

[56] B. W. Lampson. Computer security in the real world. *Computer*, 37(6):37–46, 2004.

[57] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification - W3C Recommendation 22 February 1999, 2002.

[58] N. Li, J. Byun, and E. Bertino. A critique of the ANSI standard on role based access control. *IEEE Security and Privacy*, 5:41–49, 2007.

[59] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

[60] N. Li and Z. Mao. Administration in role-based access control. In *Proc. of the Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 127–138. ACM, 2007.

[61] N. Li and J. Mitchell. Datalog with constraints: A foundation for trust management languages. In V. Dahl and P. Wadler, editors, *Proc. of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2003.

[62] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In M. Abadi and S. M. Bellovin, editors, *Proc. of the Symposium on Research in Security and Privacy (S&P)*, pages 114–130. IEEE Computer Society Press, 2002.

[63] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In P. Samarati, editor, *Proc. of the Conference on Computer and Communications Security (CCS)*, pages 156–165. ACM Press, 2001.

[64] J. J. Longstaff, M. A. Lockyer, and M. G. Thick. A model of accountability, confidentiality and override for healthcare and other applications. In R.Sandhu, editor, *Proc. of the workshop on Role-based access control (RBAC)*, pages 71–76. ACM Press, 2000.

[65] G. C. Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1998.

[66] OASIS Access Control TC. eXtensible Access Control Markup Language (XACML) Version 2.0 - Oasis Standard, 1 Feb 2005, 2005.

[67] J. Park and R. Sandhu. Originator control in usage control. In *Proc. of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, page 60, Washington, DC, USA, 2002. IEEE Computer Society.

[68] J. Park and R. Sandhu. Towards usage control models: Beyond traditional access control. In E. Bertino, editor, *Proc. of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–64. ACM Press, 2002.

[69] Paulson. Designing a theorem prover. In Abramsky S, Dov. M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon, 1992.

[70] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[71] F. Pfenning. Automated theorem proving course handouts, 1999.

[72] F. Pfenning. Linear logic course handouts, 2002.

[73] F. Pfenning and C. Schürmann. System description: Twelf – A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of the International Conference on Automated Deduction (CADE)*, pages 202–206. Springer Berlin, 1999.

[74] E. Rissanen, B. Sadighi Firozabadi, and M. J. Sergot. Discretionary overriding of access control in the privilege calculus. In T. Dimitrakos and F. Martinelli, editors, *Proc. of the IFIP Workshop on Formal Aspects in Security and Trust (FAST)*, pages 219–232. Springer Berlin, 2004.

[75] R. L. Rivest and B. Lampson. SDSI - A simple distributed security infrastructure. Presented at CRYPTO'96 Rump session, 1996.

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

[77] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Survey*, 28(1):241–243, 1996.

[78] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[79] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[80] M. Schunter, E. Van Herreweghen, and M. Waidner. Expressive Privacy promises - how to improve P3P. Position paper for W3C Workshop on the Future of P3P, 2002.

[81] V. Shmatikov and C. L. Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005.

[82] W. H. Stufflebeam, A. I. Antón, Q. He, and N. Jain. Specifying privacy policies with P3P and EPAL: lessons learned. In *Proc. of the Workshop on Privacy in the Electronic Society (WPES)*, page 35. ACM Press, 2004.

[83] The European Parliament and the Council of the European Union. The data protection directive (95/46/EC). Official Journal of the European Union, 1995.

[84] The European Parliament and the Council of the European Union. Directive on privacy and electronic communications (2002/58/EC). Official Journal of the European Union, 2002.

[85] The US Dpt. of Health and Human Services. Summary of the HIPAA Privacy Rule., 2002.

[86] U.S. Securities and Exchange Commission. The Sarbanes-Oxley Act, 2002.

[87] J. Wainer and A. Kumar. A fine-grained, controllable, user-to-user delegation method in RBAC. In E. Ferrari and G. Ahn, editors, *Proc. of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 59–66. ACM Press, 2005.

[88] H. Wang and S. L. Osborn. An administrative model for role graphs. In *Proc. of the IFIP TC-11 WG 11.3 Annual Working Conference on Data and Application Security (DBSec)*, pages 302–315. Kluwer, 2003.

[89] X. Wang, G. Lao, T. De Martini, H. Reddy, M. Nguyen, and E. Valenzuela. XrML: eXtensible rights markup language. In M. Kudo, editor, *Proc. of the Workshop on XML Security (XMLSEC)*, pages 71–79. ACM Press, 2002.

[90] N. Whitehead, M. Abadi, and G. C. Necula. By reason and authority: A system for authorization of proof-carrying code. In R.Focardi, editor, *Proc. of the Computer Security Foundations Workshop (CSFW)*, pages 236–250. IEEE Computer Society Press, 2004.

[91] T. Yu, N. Li, and A. I. Antón. A formal semantics for P3P. In *Proc. of the Workshop On Secure Web Service (SWS)*, pages 1–8. ACM Press, 2004.

[92] L. Zhang, G. Ahn, and B. Chu. A rule-based framework for role-based delegation and revocation. *Transactions on Information and System Security (TISSEC)*, 6(3):404–441, 2003.

[93] X. Zhang, S. Oh, and R. S. Sandhu. PBDM: a flexible delegation model in RBAC. In D. F. Ferraiolo, editor, *Proc. of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 149–157. ACM Press, 2003.

Titles in the IPA Dissertation Series since 2005

Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development*

Processes. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and*