Technical Report

UCAM-CL-TR-586
ISSN 1476-2986

Number 586

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Role-based access control policy administration

András Belokosztolszki

March 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted November 2003 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

The wide proliferation of the Internet has set new requirements for access control policy specification. Due to the demand for ad-hoc cooperation between organisations, applications are no longer isolated from each other; consequently, access control policies face a large, heterogeneous, and dynamic environment. Policies, while maintaining their main functionality, go through many minor adaptations, evolving as the environment changes.

In this thesis we investigate the long-term administration of role-based access control (RBAC) – in particular OASIS RBAC – policies.

With the aim of encapsulating persistent goals of policies we introduce extensions in the form of meta-policies. These meta-policies, whose expected lifetime is longer than the lifetime of individual policies, contain extra information and restrictions about policies. It is expected that successive policy versions are checked at policy specification time to ensure that they comply with the requirements and guidelines set by meta-policies.

In the first of the three classes of meta-policies we group together policy components by annotating them with context labels. Based on this grouping and an information flow relation on context labels, we limit the way in which policy components may be connected to other component groups. We use this to partition conceptually disparate portions of policies, and reference these coherent portions to specify policy restrictions and policy enforcement behaviour.

In our second class of meta-policies – compliance policies – we specify requirements on an abstract policy model. We then use this for static policy checking. As compliance tests are performed at policy specification time, compliance policies may include restrictions that either cannot be included in policies, or whose inclusion would result in degraded policy enforcement performance. We also indicate how to use compliance policies to provide information about organisational policies without disclosing sensitive information.

The final class of our meta-policies, called interface policies, is used to help set up and maintain cooperation among organisations by enabling them to use components from each other's policies. Being based on compliance policies, they use an abstract policy component model, and can also specify requirements for both component exporters and importers. Using such interface policies we can reconcile compatibility issues between cooperating parties automatically.

Finally, building on our meta-policies, we consider policy evolution and self-administration, according to which we treat RBAC policies as distributed resources to which access is specified with the help of RBAC itself. This enables environments where policies are maintained by many administrators who have varying levels of competence, trust, and jurisdiction.

We have tested all of these concepts in DESERT, our proof of concept implementation.

To my parents Екатерина and László.

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

The pronouns ‘we’ and ‘our’ in the text, which have been used for stylistic reasons, should be taken to refer to the singular author.

This dissertation is not substantially the same as any that I have submitted for a degree or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed 60,000 words including tables and footnotes, but excluding bibliography and diagrams.

Acknowledgements

This work would not have been possible without the constant guidance, constructive feedback, and help of my supervisor, Dr. Ken Moody, to whom I am deeply grateful. I also thank Dr. Jean Bacon, who provided me with advise and together with my supervisor led the OPERA Research Group.

I am grateful also to all the members of the OPERA Research Group who made my time enjoyable and productive. Special thanks to Dave Eyers, Peter Pietzuch, Alan Abrahams, and Walt Yao, with whom I had very helpful and inspiring discussions.

I express my thanks to King's College for providing me with a productive work environment and supporting my extra-curricular activities. Being a member I very much enjoyed living in its halls of residence, representing it in rowing races, and being part of its community. Many thanks to my friends at King's, especially to Marc Cardle and Leo Patanapongpibul.

My research was supported by the John Stanley Graduate Fund, my college, and the United Kingdom Overseas Research Students Awards Scheme. I would like to thank them for their help.

I want to thank my parents, Jekatyerina and László, for providing me with numerous opportunities to enable me to be where I am today. Finally, I would like to thank my wife, Magdalena, for constantly encouraging and supporting me.

List of Publications

[BE03] András Belokosztolszki and David M. Eyers. Shielding RBAC infrastructures from cyberterrorism. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security, July 28-31, 2002, Kings College, Cambridge, U.K.*, volume 256 of *IFIP Information Processing*, pages 3–14. Kluwer Academic Publishers, 2003.

[BME04] András Belokosztolszki, Ken Moody and David M. Eyers. A formal model for hierarchical policy contexts. In *Policy 2004: IEEE Fifth International Workshop on Policies for Distributed Systems and Networks*, 2004.

[BEM03] András Belokosztolszki, David M. Eyers, and Ken Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *Policy 2003: IEEE Fourth International Workshop on Policies for Distributed Systems and Networks*, pages 99–110, 2003.

[BEP$^+$03] András Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, U.S.A. 2003.

[BEWM03] András Belokosztolszki, David M. Eyers, Wei Wang, and Ken Moody. Policy storage for role-based access control systems. In *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, pages 196–201, 2003.

[BM02] András Belokosztolszki and Ken Moody. Meta-policies for distributed role-based access control systems. In *Policy 2002: IEEE Third International Workshop on Policies for Distributed Systems and Networks*, pages 106–115, 2002.

[DBE$^+$04] Nathan Dimmock, András Belokosztolszki, David Eyers, Jean Bacon and Ken Moody. Using Trust and Risk in Role-Based Access Control Policies. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, 2004.

[BS03] András Belokosztolszki and Erich Schikuta. An XML based framework for self-describing I/O data. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '03)*, pages 324–332, 2003.

Contents

List of Figures

1 Introduction

The objective of access control is to protect resources from unauthorised access, whilst ensuring authorised access.

Resources are generally accessed through some application, which enforces access control restrictions by allowing only authorised access. Rules, according to which access control decisions are made, are often programmed into the applications. This has the disadvantage that changes to access control requirements may require modification of the application. Furthermore, if resources are shared among many applications, the decisions to allow access to a specific resource may differ depending on the application used. To address this inconsistency *policies* are externalised. Such policies collect the rules that govern access control decisions. Since they are separated from applications, they can be shared, and thus reused. An additional advantage is that changes to policies can then be performed without the need to modify applications.

With the expansion of the Internet requirements for access control have changed. Instead of having individual computers protected by local access control policies, computers are interconnected, and access to resources is specified remotely as well as locally. As a result the number of resources and the number of potential users have increased by many orders of magnitude. This increase in complexity was reflected in policies, and has rendered them more difficult to administer.

Role-Based Access Control (RBAC) is a relatively recent access control model that can cope with the new, dynamically changing set of users and resources. While it simplifies certain aspects of policy administration, it provides new means to control access decisions. Access control decisions could thus be influenced by time, the result of database lookup, and the past behaviour of the principals which access resources.

As policy complexity increases, additional policy administrators are required. This increases the demand for tools to support cooperation in policy management. Furthermore, policy administrators face a new task, viz. monitoring and controlling the policy modifications made by their colleagues. Such requirements are common in environments where, in order to follow organisational structure, access control policies have been distributed over many computers.

1.1 Research motivation

There are two main application scenarios that motivated our research. The first one is the National Health Service (NHS) in the United Kingdom. The second application scenario is access control specification to event middleware.

1.1.1 National Health Service

In the United Kingdom, similarly to many other countries, patients are associated with a general practitioner (GP). The first contact between a patient and doctors usually starts through GPs, who, if needed, refer patients to specialist doctors. Patient data is stored in GP surgeries, and data about hospital treatments is located in the relevant hospitals. Through their lifetime patients may be treated in many different hospitals. Since data about such treatments might be vital for subsequent treatments, it is important that all medical information pertaining to a patient is accessible.

This need is realised by the British government, which set the goal of enabling doctors to access patients' Electronic Health Records (EHR) independently of the locations of doctor and patient.

Access to data is regulated by the NHS guidelines. These regulations aim to prevent fraudulent data access, as well as to provide patients with control over their data.

The NHS data access guidelines could be enforced at the location of the data, i.e. the hospitals and the GP surgeries. These enforcement points are autonomous, and have a large degree of freedom in managing their IT infrastructure. They do not run the same applications, and are further subdivided into semi-autonomous units, like wards (see Figure 1.1).

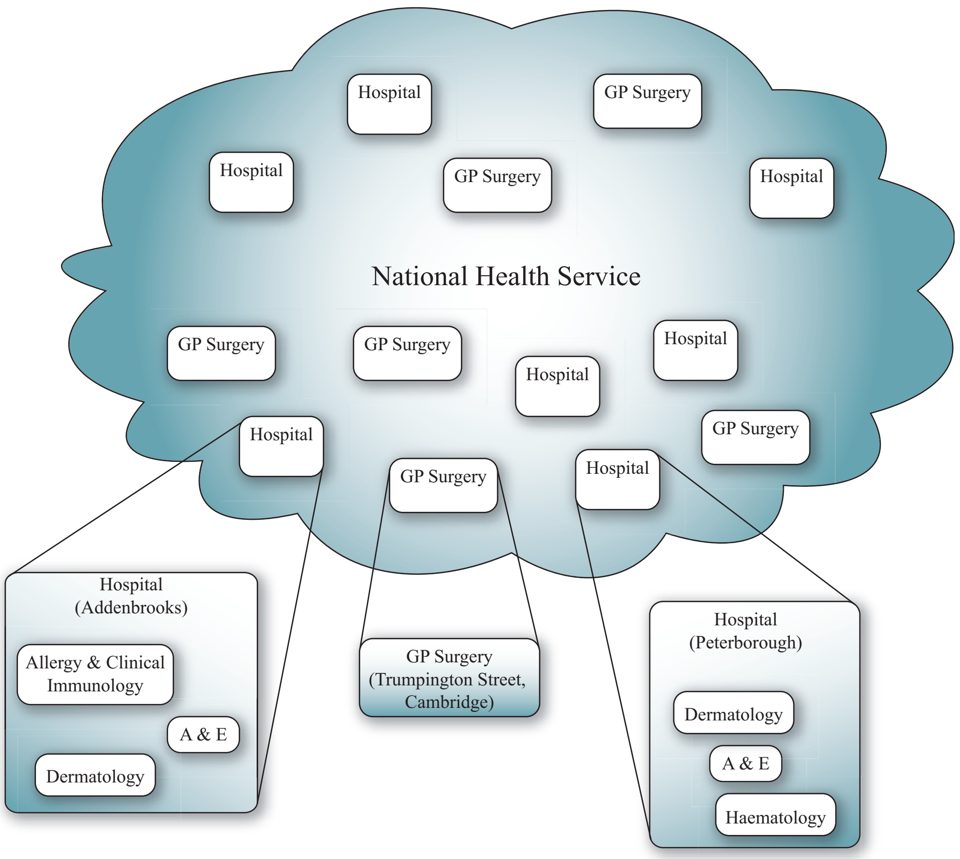


Figure 1.1: The National Health Service (NHS).

Therefore, the goal is to enable cooperation between hospitals and surgeries, in an environment where access control is specified locally, but according to some higher-level rules. We can find such hierarchical control over access control policy specification at multiple levels, for example between the NHS and hospitals, and between the hospitals and their wards.

One of the challenges is the heterogeneity of the policies involved, since, due to legacy applications and local autonomy, the deployment of applications managing patients' data has followed a bottom-up approach.

A further challenge is the required flexibility of the access control policies, since, for example, the NHS "Patient's Charter", and its successor, "Your Guide to the NHS" [Nat02], allow patients to restrict access to their own data. For example, a patient may prohibit access to her termination records by her aunt. Following such a restriction the aunt must be denied access to the relevant records, even if under normal circumstances she may be authorised to access them (e.g. she is working as a nurse).

Regulations, such as the above, and the scale of the health care environment provide many interesting research problems in access control management. A set of requirements, as well as the description of interesting problems in the health care area, can be found in [Bez98] and [WFSM02].

1.1.2 Security in event middleware

Another motivating application for our research comes from publish/subscribe event middleware. Events are asynchronously delivered, typed messages that originate from event publishers. Such publishers advertise the event types they intend to publish, and event subscribers may register their interest in such event types. Event middleware manages subscriptions and advertisements, and is responsible for delivering publications to the subscribers.

Such event architectures can span large networks. An example could be an active city, in which events within active houses (houses in which appliances are linked to a network) are communicated according to the above paradigm. For example, a fire sensor in a home may publish a fire event to which the owner of the house and the local fire brigade are subscribed.

When information contained in events is sensitive, access to events needs to be controlled. Only lately has this research area received attention [Mik02, WCEW02, BEP$^+$03].

From the policy perspective, event middleware is similar to the scenario of the NHS; there are many different cooperating domains, all of which may have autonomy over their administration. For example, access control policies within an active house must comply with the rules set by the utility providers of the active city, whose policies are also restricted at a higher-level.

Due to the communication requirement, restrictions that need to be enforced span many autonomous domains.

In [BEP$^+$03] we describe a way to control access to event advertisements, publishing, and subscriptions.

1.1.3 Large-scale organisations

The main reasons for difficulties in administering large-scale organisations are bottom-up deployment, legacy applications, badly designed organisational policies, heterogeneity, and local autonomy.

Unfortunately, due to the scale of such organisations, top-down access control deployment is an infeasible option.

1.2 Thesis contribution

Our aim in this thesis is to provide policy administrators with the means to manage RBAC policies, in large, distributed environments, where access control policies may be specified locally.

The main contributions of this thesis are:

Policy structure We define a policy structure that we use to express RBAC policies. This structure serves many purposes. First of all it defines components of policies. Second, it provides an abstract interface to policies. This interface is vital for specifying access control to policy components.

The concept of meta-policies It is difficult to express organisational policies with the help of available access control specification primitives, in our case the policy components. During the conversion from organisational policies certain information is lost. This loss is less critical for relatively static policies, but since in the long-term policies generally undergo changes, it is vital to preserve more information. In order to encapsulate such information we introduce meta-policies, that encapsulate some long-term properties of organisational policies, and decouple them from the policies as they are actually enforced.

Contexts We introduce the concept of contexts to group policy components according to various aspects, which themselves may change in time. With the help of an information flow relation we provide a means to restrict the use of policy components alongside components belonging to certain other groups. This helps to organise access control policies into a hierarchical, multidimensional structure.

Compliance policies Long term policy goals and requirements are usually part of organisational policies, but often they are only implicitly present in RBAC policies. We introduce compliance policies to fill this gap, and provide policy administrators with a way to store persistent, global, and higher-level access control requirements.

Since compliance policies encapsulate long-term goals and requirements for access control policies, they constitute a powerful means to control the evolution of access control policies at a lower abstraction level.

Compliance policies can be used to control existing policies, but they are also suitable as templates for new access control policies, thus helping conversion to a top-down policy design.

Interface policies Cooperation between autonomous policy management domains is a growing requirement in today's organisations. We define interface policies that help to set up cooperation between policy management domains. Furthermore, interface policies can set requirements for the cooperating parties.

These interface policies help to overcome certain problems associated with bottom-up policy design, since they help to convert between different policy representations.

Access control to policies We specify a set of privileges (or permissions) that can be used to specify fine-grained access control to our policy specifications. Together with our meta-policies, these privileges can be used to control access to policies from within policies. They can also be used to set up sandboxes or playgrounds for other policy administrators, allowing them to perform policy-limited modifications according to their trust, knowledge, and jurisdiction.

Policy evolution We investigate the evolutionary process of policies and our meta-policies, particularly from the aspects of life-time, self-administration, and hierarchical policy management.

Desert We present a proof of concept implementation that we have used to test the ideas described in this thesis.

1.3 Dissertation outline

This thesis is organised as follows:

Chapter 2 reviews research in the area of role-based access control (RBAC). It begins with a short introduction to earlier access control models. This is followed by a description of RBAC, in which we introduce the basic RBAC concepts. We then describe some popular RBAC extensions.

As part of this chapter we introduce OASIS, an RBAC model and implementation we use as the primary testbed for the concepts presented in this thesis.

Chapter 3 describes the structure of the policy components we use throughout this thesis. It starts with the most basic constituents of our policies, such as data type and role specifications. This is followed by the specification of rules, and a brief description of the three meta-policy types: contexts, compliance policies, and interface policies.

Chapter 4 introduces contexts, our first meta-policy type. We first define a simple context model, which we use to explain the basic concepts such as context elements and information flow restrictions. We later extend this context model to include hierarchical context elements. Finally we list potential uses for contexts.

Chapter 5 presents compliance policies, the second type of meta-policy. After describing our motivation for developing compliance policies we incrementally define their structure. First we introduce a simple model, which we then extend to support additional features, such as lossy functions, implicit rules, and negation. The chapter ends with a review of related work and a discussion of future directions.

Chapter 6 defines interface policies, our third type of meta-policy. This chapter starts with an overview of OASIS's support for cooperation, and explains the potential problems that can arise in the long-term. We present interface policies as a potential solution to these problems, and to provide a means to improve inter-policy cooperation. Finally, we describe related work and indicate some directions for further research.

Chapter 7 considers self-administration of our access control policies. It defines a set of privileges that could be used by various policy management APIs. These privileges are later extended to use contexts in order to support easier administration.

Chapter 8 shows how the three types of meta-policy and administration privileges fit into policy administration and policy evolution as a whole. The second part of this chapter considers policy version management. As the final part of this chapter we describe DESERT, our policy management framework.

Chapter 9 concludes this thesis by providing a summary of its main contributions.

2 Background

This chapter examines fundamental technologies and research related to this thesis. We begin with a brief review of early access control models. This is followed by an introduction to Role-Based Access Control (Section 2.2), in which, after describing its core concepts, we review the extensions that are available. Later, we describe the most popular RBAC models and implementations. In Section 2.3 we provide an overview of OASIS, the RBAC model this thesis primarily focuses on. Finally, we discuss some of the problems of RBAC systems.

2.1 Early access control models

Since the early 1960s access control has been a major issue, mainly in database management systems and in operating systems. Its objective was to protect system resources against undesired or inappropriate user access, whilst permitting authorised access.

Early research on access control was dominated by two approaches [NC00, SS94]: Mandatory Access Control (MAC) and Discretionary Access Control (DAC).

2.1.1 Mandatory Access Control

Mandatory Access Control, supported by military research and civilian government, enforces access control by means of security labels. This model, first formalised by Bell and LaPadula [BL75], attaches labels or security classifications to every object and user. As the many variations of the Bell-LaPadula model led to confusion, in [San93] Sandhu introduced a minimal model, named BLP, that encapsulates the essentials of the Bell-LaPadula model.

In both of these, access is granted based on the subject's and the accessed object's security label. We shall denote the security labels of these by $\lambda(s)$ and $\lambda(o)$ respectively. These security labels form a lattice with a partial order relation, $\leq$.

As MAC was developed with a military environment in mind, confidentiality was a major motivating issue, which was achieved by information flow restrictions among entities (users or objects) of different security groups. These restrictions are expressed by the following two properties:

- The simple security property, which is also known as "no read up", states that a subject can only read an object if $\lambda(o) \leq \lambda(s)$.
- The $\star$-property, or "no write down" property, allows a subject to write an object only if $\lambda(s) \leq \lambda(o)$. This property addresses information leakage by malicious programs. It does not allow programs to write information to objects that can be read by subjects with

a less privileged security clearance. Sometimes a different $\star$-property is used to address the problem arising from subjects writing contaminated data to objects of higher security class. The modified $\star$-property thus only allows the writing to objects that have the same security label as the user, formally $\lambda(s) = \lambda(o)$.

The system administrators are responsible for maintaining and setting security labels; accordingly, no other user may change these labels. Whenever an object is duplicated, the same security label is attached to the duplicate object.

A similar model to BLP was published by Biba [Bib75]. Unlike BLP, Biba's model aims to achieve data integrity as opposed to confidentiality. It allows data flow only from high to low integrity data, which is exactly the inverse of permitted information flow in the BLP model.

Unfortunately MAC models are rather rigid. They support a fixed set of security classes, a fixed set of supported operations on objects, and allow only administrators to modify the access control policy.

Nevertheless, the information flow restriction aspect of the MAC model greatly motivated parts of our work on contexts, described in Section 4.4.

2.1.2 Chinese Wall policy

To address confidentiality breaches through insider knowledge in [BN89] Brewer and Nash introduced the Chinese Wall policy. The motivating example for this policy is a financial institution that provides corporate business services. An analyst in this institution must abide by the rules of confidentiality, i.e. he cannot advise corporations where he holds insider knowledge about the plans of the competitor. However, analysts are free to advise non-competing organisations. This policy is required in the operation of many financial organisations, but it cannot be expressed in the Bell-LaPadula model.

The general idea behind the Chinese Wall policy is to form *company datasets* that contain the objects belonging to specific companies. Each of these company datasets refers to a *conflict of interest* class, for example 'Petrochemical' or 'Financial Services'. Based on this object classification, analysts are not allowed to access information which could conflict with any other information the analyst has already accessed.

The Chinese Wall policy indicates why MAC policies, although they suit military requirements well, cannot express important policies that are common in commercial organisations.

2.1.3 Discretionary Access Control

Discretionary Access Control originates from the academic area. It allows or denies access to an object based on the identity of the accessing user or group. Contrary to mandatory access control, users are allowed to control permissions to their objects. In this way, users can delegate their own rights to other users. A classical example of DAC is *Access Control Lists* (ACL), where objects are associated with a list of users or groups that are allowed access.

In a static world, where access does not depend on previous actions, a matrix can represent access to all objects by all users. This matrix, called the *access matrix* [Lam71, San92], has a row for every user and a column for every object. The elements of the matrix store the access rights of an individual user to a specific object. This matrix can be enormous in size, although it is usually sparsely populated. The task of access control includes developing a representation for the storage of this matrix. In practical systems typical approaches to implement this matrix are [SS94]:

- to store it by columns, regarding each column as a list. This leads to the access control list model (ACL).

- to store it by rows. These rows are known as capabilities. Such capabilities store a list of the allowed privileges for every subject.
- to store only the filled cells of the matrix together with their position in the matrix.

These approaches have both their advantages and disadvantages. Problems usually arise when new users are added, existing users are deleted, and when objects are created or deleted. For example, in the case of access control lists the removal of a user would imply a check of each access control list in the system. In organisations where users, as well as the objects to be protected, are changing frequently, DAC proved inadequate. But the flexibility DAC provides, namely that users can be in control of a portion of the access control policy, is an important aspect of this model that partly motivated our research on access control to policies, including self-administration of policies (see Chapter 7).

2.2 Role-Based Access Control

Because of the rigid nature of MAC, where users had little or no control over the access control policy, and the problems associated with policy changes in DAC, early access control models could not meet practical requirements of commercial organisations [CW87]. It was also realised that in large organisations data is not owned by individual users, but by the organisation itself, thus access to data should consider one's position in the organisational hierarchy. This inspired further work, a result of which was *Role-Based Access Control* (RBAC), the access control model that will be discussed in this section.

Early work on role-based access control goes back to 1988, when Lochovsky and Woo defined roles and organised them into a hierarchy [LW88]. The basic idea of role-based access control is to include another level of indirection between the user to permission (or privileges) mapping. Roles thus break this mapping into two, the first part maps users to roles, while the second maps roles to privileges. This is illustrated in Figure 2.1.

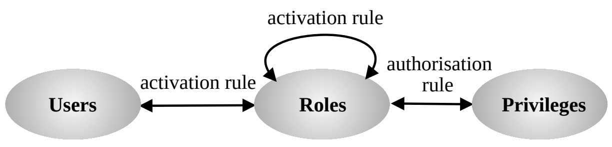


Figure 2.1: The basic RBAC model.

The indirection introduced by roles is very similar to one that can be expressed by groups, but while groups have only users as members, roles can form collections of users, permissions, and other roles [San95]. Also, unlike the case for groups, users can act in specific roles upon request, i.e. a user activates a role only when she needs the privileges associated with the role. Due to this dynamic behaviour RBAC can support the concept of least privilege [JHS75], which requires the users to hold a minimum set of privileges that is necessary only for their current task, thus avoiding unnecessary, accidental, or malicious resource access.

RBAC is a very general model that not only solves the described problems of MAC and DAC, but also has an expressiveness that enables it to express both MAC and DAC policies [NO95b, SM98].

During the last decade role-based access control has received considerable attention. Several models, like those published in [BMY02, NO99, SCFY96, FSG$^+$01, LS99], were developed independently. The differences between these models will be discussed later, but before that we present the basic concepts of RBAC.

A widely cited document [SCFY96] in the world of role-based access control distinguishes four kinds of RBAC models. The first model – called RBAC$_0$ – is the simplest, it serves as a basis for the other three models. RBAC$_1$ extends the basic model with role hierarchies. RBAC$_2$ adds constraints to the basic model, while the consolidated RBAC$_3$ model combines both RBAC$_1$ and RBAC$_2$, supporting both role hierarchies and role activation constraints. The relation between these four models is shown in Figure 2.2.

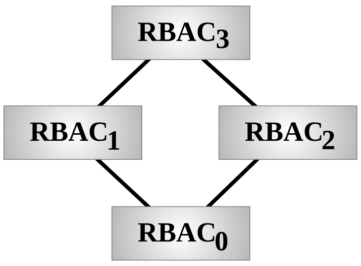


Figure 2.2: Relationship of the RBAC models.

Sandhu et al. refined these models and provided a more precise, but informal description in [SFK00]. These models use slightly different names, RBAC$_{\{0-3\}}$ are referred to as flat RBAC, hierarchical RBAC, constrained RBAC and symmetrical RBAC respectively.

The basic components of RBAC$_0$ are users, roles, permissions, and sessions. RBAC$_0$ includes a many-to-many permission to role assignment and a many-to-many user to role assignment. To

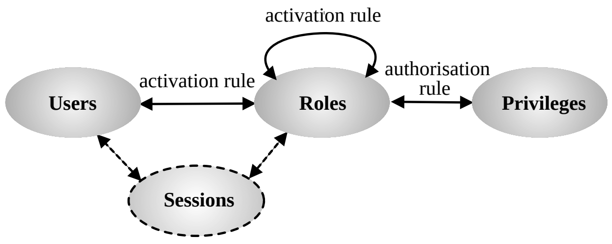


Figure 2.3: The basic RBAC model extended with sessions.

extend the support for least privilege *sessions* are introduced. They correspond to a particular occasion, when a user signs on the system to carry out some activity [SFK00]. In this way, sessions add a layer of indirection between users and roles (see Figure 2.3), so that users activate roles within the frame of a session. Users can have many sessions, and in each session they can have different sets of roles active. Whenever a session is terminated the roles activated for that session are revoked.

2.2.1 Role hierarchies

RBAC$_1$ extends the basic model with *role hierarchies*. Role hierarchies were thought to be a natural way to describe role relations reflecting organisational structure. The permissions assigned to a junior role are inherited transitively by the more powerful senior roles. In Figure 2.4 we show an example role hierarchy from [SFK00]. This example illustrates the hierarchy relation of ten roles, the most senior of which is the *Director (DIR)* role.

The argued advantage of role hierarchies was the easier management of roles, as they were thought to reflect better the structure of an organisation. Many models support role hierarchies, among these are [NO99, JSSS01, Lup98, AMN02].

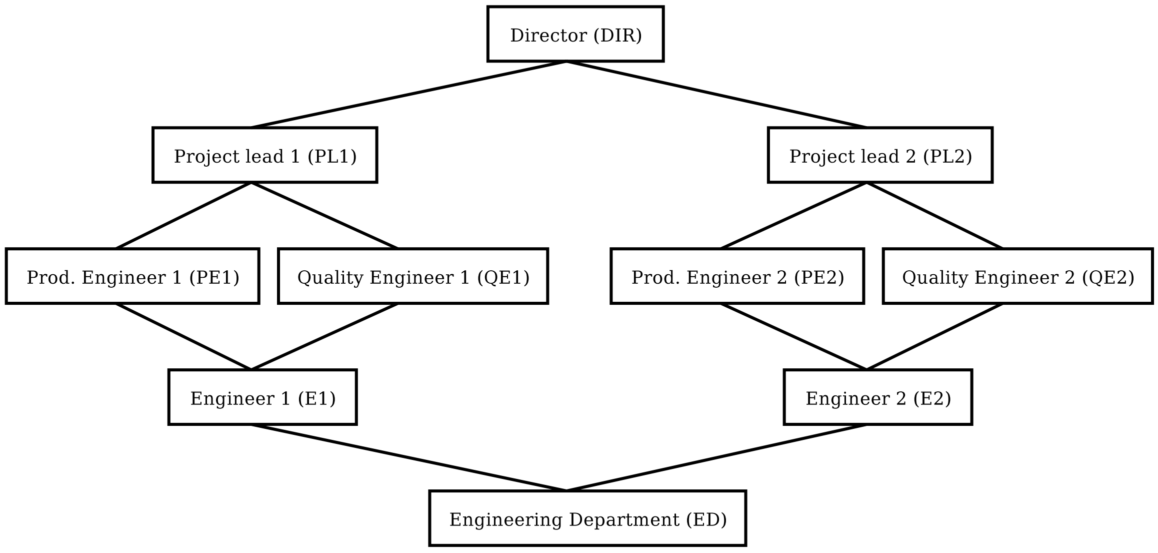


Figure 2.4: Example role hierarchy.

Role hierarchies have often been questioned [Awi97, San98, GB98, ML99, HV01]; it has been shown that permission inheritance is often absent in organisational structure where junior roles have more permissions than senior roles. Different kinds of hierarchies have been proposed [ML99] to solve the above problem; however, this also introduced additional complexity.

Many RBAC implementations consider only RBAC$_1$, as, especially without activity, this model is fairly simple. For example, the work presented in [BD99] demonstrates how RBAC$_1$ can be implemented using CORBA's Security Service. Similarly, Didriksen looks at how RBAC$_1$ could be implemented in active relational database management systems by translating policies into SQL triggers [Did97].

2.2.2 Constraints

The RBAC$_2$ model extends RBAC$_0$ by adding constraints, which help to specify preconditions to role entry. Chen and Sandhu distinguish two major kinds of constraints [CS95]. The first type specifies preconditions to role entry, while the second type specifies invariants that must be maintained by the policy enforcing system. The first constraint group can be further divided into two sub-categories, thus we consider next the following constraint categories: prerequisites, Separation of Duty (SoD) constraints, and cardinalities.

Prerequisites

The first type of constraint specifies a set of prerequisites a user must satisfy upon entering a role. One such prerequisite can be the requirement for a user to hold a particular role. An example would be to require someone to have the *university_student* role before allowing him or her to enter the *computer_science_student* role.

Other prerequisites can include more complex predicate evaluations, such as ones that consider the time of the evaluation. There have been only a few attempts to formalise languages to express particular types of constraints. One such research effort was carried out by Bertino, Bonatti, and Ferrari who define expressions to specify temporal constraints [BBF00].

Separation of Duties

The second type of constraint we discuss here is Separation of Duty (SoD). It is considered to be among the most important constraints and it is often mentioned as the driving motivation for role-based access control. SoD is a technique for reducing fraud by means of spreading responsibility and authority to perform a certain task over several users. A common example is the preparation and approval of a purchase order. In this example an order must be initiated and approved by different persons, therefore fraud requires the conspiracy of two people, which is less likely to happen, and is more likely to be disclosed or captured. Simon and Zurko [SZ97] informally identified a large number of SoD types:

Static separation of duties, also known as strong separation of duties, achieves separation of duties by specifying the role entry policies in such a way that it is impossible for any user to assume both of the conflicting roles.

Dynamic separation of duties, also known as weak exclusion, allows users to enter each of the conflicting roles, but entry is controlled at policy enforcement time. Based on the type of this control, weak separation of duties can be classified as:

- **Simple dynamic separation of duties** prohibits users from assuming conflicting roles at the same time.
- **Object-based separation of duties** allows users to enter the conflicting roles, but the conflicting privileges cannot be exercised on the same object.
- **Operational separation of duties** allows conflicting roles to be entered as long as the union of the privileges in the roles activated does not contain the privileges of a business task.
- **History-based separation of duties** prevents users from performing all the actions in a business task.

Gligor, Gavrila, and Ferraiolo formalised the above mentioned separation of duties in [GGF98]. Ahn and Sandhu extended them in [AS99] and provided the RSL99 language to describe separation of duty types. Kuhn provides a formal model to express separation of duties in single session environments [Kuh97]. Additional separation of duties, like the Chinese Wall (see Section 2.1.2), can be found in [JSS97].

In our work we shall primarily concentrate on static SoD, but our contexts defined in Chapter 4 can be also used to specify different varieties of dynamic SoD.

Enforcing dynamic separation of duty constraints at authorisation level is a difficult task. This problem is usually aggravated by sessions, as they allow the same principals to have many sets of active roles, as opposed to a single set. The common solutions to handle dynamic SoD constraints usually approach the problem from the protected object itself, and add some state information to these objects [San88] or use locks [Fad99].

Cardinalities

Cardinality constraints are the last type of constraints we consider. They limit the number of sessions a user may have or the number of users being active in a role at the same time. For example, cardinalities can express the policy that there could only be one *administrator* active at any particular time. Cardinalities are part of the *activation control* type constraints [SZ97] that were described in this section. These constraints must be evaluated at runtime.

2.2.3 Extensions

Apart from the above aspects to distinguish role-based access control models, there are other characteristics unique to certain models:

Delegation

Delegation allows a user or a role to empower other users or roles with some of the rights the user holds. The motivation for this comes from real-world scenarios, where users need to act on behalf of other users for accessing resources, for example, when they substitute for someone who is ill. There is a large amount of work concentrated on role delegation [BS00a, ZAC01, BS00b, NC00]. Some consider partial delegation, when only privileges are delegated, or when the delegated role is further constrained. Different delegation types are well described in [BS00a].

Delegation is strongly connected with the concept of *revocation*, which allows delegating users to revoke a specific role, or delegated roles to be revoked using time-restriction constraints. Architectures supporting delegation and revocation differ in the way that revocation is implemented. OASIS, the RBAC implementation we used as our primary testbed, stands out among these architectures with a fast, event based revocation system [Hay96]. OASIS also provides the notion of appointment, which is an abstraction of delegation; for more details see Section 2.3.

Parameters

To ease generalisation, many RBAC models include role and/or privilege parameters. Such parameterised roles can be refined during role activation time by setting the parameter values. Note that these parameter values are fixed, i.e. once they are set they are not modified. Parameterised roles serve only as a template. The parameters of privileges, similarly to the parameters of roles, are set when a privilege instance is created; consequently, the parameters of an issued privilege cannot be modified. Such privileges can contain information about the object accessed and thus reduce the number of privileges in the policy specification. For example, instead of specifying a large number of privileges of the form *NHS_read_haematology*, *NHS_read_biochemistry*, *NHS_read_microbiology*, and so forth, a single privilege with a NHS record type *(NHS_read_record(record_name))* could be used.

While parameters introduce a way to abstract away privileges and roles, they cause problems for role hierarchies. The parameters of the roles in a hierarchical relation must be set somehow, and this can become problematic if the parameters of the parent role are significantly different from those of the child role. Thus the role inheritance relation must include parameter setting information.

Parameters increase the demand for constraints, which, by supporting parameters, also become more complex [Jae99]. Accordingly, static policy analysis becomes harder. For example, parameters complicate cardinality checks and separation of duty constraints, as role names are no longer sufficient to distinguish between role instances, thus the policy enforcer needs to store more state information.

Context awareness

Requirements for access control specification languages continue to grow. This has made it necessary to consider the context in which the access control decision is made [CLS$^+$01]. Support for context awareness[1] can range from temporal predicate evaluations [BBF01] to complex

[1]Note that in this thesis we shall use the term context for concepts that are not related to context awareness.

predicates that provide information about the underlying system [GMS94] or allow almost arbitrary predicates to be evaluated [BMY02]. This extension to RBAC policies enables further restrictions to be added to role activation, restrictions that can be time-based or that can even perform complex database lookups.

Distribution

Scalability is an important factor in access control systems. A centralised RBAC system might not be able to follow requirements of large scale organisations that are normally spread out geographically; thus, it is essential to consider distribution.

Distribution introduces many challenges. To support users accessing resources from different locations, most implementations are extended with sessions. This, as well as the distribution itself, complicates constraint checking; for example, it is more difficult or expensive to gather data about currently active roles of a user. These systems must also be prepared to handle network failures. As a consequence, the policy language itself must be extended to allow the specification of policy enforcement behaviour under degraded network conditions. As different portions of organisations have some level of autonomy, there is a natural demand to modify policies locally.[2] Due to such requirements more and more RBAC models are being extended to support distribution [San00, HB99, YMB01].

Obligation policies

Most RBAC models are *permissive*, i.e. policies specify what a subject is allowed to do. In contrast *obligation* policies [LS97], which originate from requirements in system management, specify what subjects must or must not do to a set of target objects.

An example of an RBAC policy language that supports obligation is Ponder [DDLS01]. Obligations work well with other RBAC features, they can make use of hierarchies and contexts. An interesting work of Schaad et al. [SM02a] proposes to further integrate obligation with the popular RBAC extensions by providing support for delegating obligations.

Negative policies

Permissions that can be specified in most RBAC models are positive, i.e. they specify what one is allowed to do. In such environments the default policy is to deny access if it is not allowed explicitly.

On the other hand, negative permissions are exactly the opposite of positive ones; they specify what one cannot do. If only negative policies are present the default access control behaviour is to allow everything that is not explicitly prohibited.

Models can include both positive and negative policies, an example for such can be found in [Lup98]. The use of both positive and negative rules leads to conflicts as certain actions can be both permitted and prohibited at the same time. Such models must address conflict resolution. Generally this involves a priority ordering for rules, which specifies that rules with a higher priority always dominate over those with a lower priority.

Negative permissions can be simulated by positive permissions and through the appropriate use of policy constraints [AS99].

[2]We discuss policy maintenance and distributed administration later in this thesis.

Freedom of choice

RBACs differ in how much freedom they give to users, and this implementation detail is reflected at model level. Some RBAC implementations only allow roles to be activated automatically on demand of a privilege. Indeed, some RBACs (e.g. [CO02]) activate all the accessible roles, thus breaking the support for least privilege. Others permit users to activate the roles they need at the time when they need it. At model level, RBACs that support role revocation usually also allow users to deactivate roles. This further increases the user's control over the currently active role set.

Position-related vs. task-related roles

Roles group both privileges and users. Depending on the perspective this can be done in many ways. In [SBC$^+$97] Sandhu et al. distinguish three different types of roles: abilities, groups, and UP-roles. Abilities are roles that can only have privileges and other abilities as members, i.e. they do not explicitly group users. Groups, on the other hand, collect users and other groups, and do not explicitly consider privileges. UP-roles are roles that have both users and privileges associated with them.

Another classification of roles is based on what real-world entity the role corresponds to [NS02]. According to this classification a role can be *functional* or *organisational*. A functional role groups together privileges that are specific to a business function. For example, the role *database_user* is a functional role. An advantage of functional roles is that they are resilient against organisational restructuring. An organisational role, on the other hand, reflects the position of people in a hierarchical organisation in a company. An example for such a role is the *manager* role. RBACs that support organisational roles usually also support hierarchies.

When RBAC is not enough

All these extensions may give the impression that role-based access control is a very expressive access control model, but there are scenarios, and not only pathological ones, where it is very hard to express real-world policy with the help of roles and role activation rules. An example is a team that has a goal, and team members who are cooperating. Such cases give rise to other access control models like team-based access control [Tho97] or task-based access control [TS97].

Another scenario when RBAC is not sufficient in itself is the case of work-flows. To support work-flow the RBAC enforcement system needs to consider and rely on external information, e.g. the current stage of a business process. A good overview of how to extend RBAC to support work-flows can be found in [BFA97, HA99].

2.2.4 Review of available RBACs

The concepts we introduce in this thesis are applicable to many RBAC models, even though these models differ significantly. Next we introduce briefly the most popular RBAC models.

Sandhu and the NIST model

Ravi Sandhu is among the pioneers of RBAC research. He, his students and colleagues have developed many access control models and extensions. We have already mentioned their most cited work about the four basic RBAC classes [SCFY96] that we used to introduce the various features of RBAC. In [SFK00] Sandhu, Ferraiolo, and Kuhn revise these four models in an attempt to further formalise and standardise RBAC, but their description of the models still remains very general. In [FSG+01] Ferraiolo et al. describe a proposed RBAC standard and provides a formal specification using Z notation.

Almost all RBAC models Sandhu has worked on have support for role hierarchies. In their [SFK00] paper the authors go even further by requiring hierarchies to be prerequisites for separation of duty constraints.

The work described in [San00] is among the few that addresses certain distribution issues in RBAC. In this paper the authors provide a number of simulation models, autonomous entities that are administered separately, each recognising only a portion of the available roles.

Sandhu and his group are also among the few researchers who pay much attention to how RBAC policies are administered. In their paper Sandhu et al. [SBC+97] introduced the ARBAC model to manage RBAC using RBAC itself.

While many of these models are extensive and well specified, they provide no support for parameters. Parameters help to abstract roles and privileges, and therefore further simplify policy administration. Also parameters help to incorporate information into roles. The need for this was recognised in [AKS03].

Ponder

Ponder is an RBAC implementation developed at Imperial College, London. It is based on Sloman's and Lupu's previous experiences in policy-based management. Before considering RBAC, Sloman had done much research into policy-based distributed system management [Slo94].

Building on Sloman's research, Lupu considers in his work [Lup98, LS99] policies that can express permissions as well as obligations [LS97]. As both negative and positive policies are supported much attention has been paid to conflict resolution via rule prioritising.

In [DDLS01, Dam02] Damianou extended Lupu's work in a number of ways. These extensions included policy grouping for reusability, and delegation support [YLS96].

Sloman et al. have realised that specifying policies to individual objects is inefficient, and to address this problem they introduced domains. These domains group objects together, thus policies can be applied to domains as opposed to just individual objects. The membership of a domain is explicit, i.e. it is not defined in terms of predicates.

One of the strengths of Ponder comes from its strong typing and its support for object-orientation. The latter makes the policy specification language highly extensible and flexible. A self-explaining example policy rule is:

```
inst auth+ fileServerAccess {
   subject /Employees;
   target  Servers/PrinterServer;
   action  *;                      // wild-card that allows all actions
   when    Time.after(''1300'');   // constraint
}
```

Ponder supports three types of constraints. Subject/target state constraints consider the attributes of the subject or the target object. Action/event parameter constraints are expressions that use the parameters of events, which are fundamental for obligation policies. Finally, time constraints are supported via a specific time library.

Jajodia and Bertino

Realising the need for a flexible access control mechanism that is not bound to an application, in [BJS96] Bertino, Jajodia, and Samarati propose a centralised policy-based access control model that supports both positive and negative authorisations. Their model supports strong authorisations, that must be obeyed, as well as weak authorisations, that allow for exceptions. Building on this model, Jajodia et al. present FAM (Flexible Authorisation Manager) [JSSB97, JSS97], a formal model to enforce multiple access control policies within one system. This model makes use of predicates like *cando*, *do*, the derived *dercando* predicate, *done* and *error*. With these they express what a subject can or cannot do, while the *error* predicate helps to ensure consistency. As both positive and negative authorisations are considered their model must consider conflict resolution. These models were not role-based, but they supported groups and delegation, and they served as a good formal foundation for the following RBAC models.

[JSSS01] defines FAF (Flexible Authorisation Framework), which introduced role support while maintaining positive and negative authorisations. Partly due to their previous access control models Jajodia et al. kept the notion of groups. In their model, group membership is static, while roles are dynamic, as users may activate different roles at their will.

FAF can also consider access history in decision making, but real support for constraints was introduced in [BBF01]. This paper formally introduces temporal constraints and provides a proof of concept implementation using Oracle DBMS.

An application of their model is presented in [SRJ01], in which, motivated by the problems of slow revocation in PKI, Samarati et al. explore the use of their policies for key management.

Nyanchama and Osborn

Nyanchama and Osborn introduced their role graph model in [NO94], in which they propose a way to administer roles using graph theory. Their model defines roles as a named set of privileges. Because in their model neither roles nor privileges are parameterised, it is simple to find relations among roles by searching for shared privileges. This relation is used to organise roles into a graph. To make their role graph connected they define the *MaxRole*, the least common senior role (least upper bound of privileges), and *MinRole*, the greatest common junior role (greatest lower bound). Therefore the role graph, based on the junior (subset) relation, forms a lattice, where the unique least upper bound and greatest lower bound are *MinRole* and *MaxRole* respectively.

In [NO95a] Nyanchama and Osborn refine their privileges for the object-oriented world. They also look at method call implications, which consider dependencies among method calls.

In [NO99] Nyanchama and Osborn continue their work on role hierarchy management, but in this paper they investigate conflict of interest constraints. They categorise these into five types (user-group, role-role, privilege-privilege, user-role assignment, and role-privilege assignment conflicts). Unfortunately these restrictions can only express static separation of duty.

An interesting work is presented in [Osb97] that looks at MAC properties in the role graph model. This work considers objects and subjects with security labels assigned to them and how role hierarchies could be built by maintaining MAC's information flow restrictions. This work is extended in [Osb02], where the authors provide methods to produce information flow graphs from role hierarchy graphs.

Named set of protection domains

Giuri defines roles as a set of named protection domains [GI96, Giu95]. A protection domain is a privilege set, which identifies the set of all operations that could be preformed by a subject at a given time. Protection domains were introduced to support dynamic separation of duty like constraints at privilege level, i.e. they disallow subjects having conflicting privileges at the same time.

Roles are defined as a set of privileges and other roles. This implicitly supports role hierarchies as roles can be derived from other roles.

To support separation of duty, their role specification language allows *and-roles* and *or-roles*. And-roles specify a set of privileges and roles that can be enabled simultaneously, while or-roles define mutually exclusive roles.

Their basic model (Named Set of Protection Domains) does not support constraints, but an extension which does is provided in [GI96]. This adds a constraint element to protection domains that must be satisfied (together with all the constraints of the roles this named protection domain extends) before it is activated.

In [GI97] Giuri and Iglio further extend their RBAC model to support content based access control. They introduce restricted privileges that contain logical expressions which must be satisfied at the time the privilege is used. In this model they also introduce role templates that are basically parameterised roles.

In [Giu98] Giuri analyses the available sandbox policy support of the Java platform and how this could be extended to support RBAC. The implementation of such RBAC support in Java is described in [Giu99].

Schaad, Kern and Moffett

Schaad et al. have done many case studies that formed vital feedback for academic research. In [SMJ01] they show the results of such a case study that was performed at a major German bank with over 50,000 employees and around 1,500 branches. This work identified 65 official positions that are ordered into a hierarchy, and 368 different job functions. Because many job functions are related to only a few positions, the effective number of roles was about 1,300.

Kern et al. also look at case studies and share their practical experience about the use of roles in commercial organisations [KKSM02]. An important observation they make is that access control roles do change, often because it is difficult to transform natural roles into ones that can be used in an access control system.

PERMIS

Otenko and Chadwick have developed PERMIS (PrivilEge and Role Management Infrastructure Standards validation), an RBAC infrastructure that uses X.509 attribute certificates [CO02]. PERMIS has support for role hierarchies that help to reduce the size of policy specifications.

Their model supports delegation, but due to the underlying X.509 certificates and the problems associated with their revocation, role revocation is a problematic issue. Delegation in PERMIS can be restricted to a certain delegation depth, therefore it is possible, for example, to delegate a role that cannot be delegated further.

PERMIS only partially supports constraints. While some temporal constraints can be included in the X.509 certificates it is expected that constraints are enforced at the targets. Thus, when compared to RBAC models that support role revocation and dynamic role activation, the user-role associations of PERMIS policies are rather static [BMCO03].

While PERMIS has no direct support for role parameters it is able to evaluate context, thus enabling policies that consider identity information. For example, policies can express that all users in a *Generalised* role can read *their own* pending car parking fines.

X.509 in RBAC The idea to use RBAC and PKI (Public Key Infrastructure) can also be found in [HMM^{+}00], where Herzberg et al. provide a mechanism for distributed environments to map users to roles based on X.509 certificates. In his thesis [Yao02], Yao describes a trust management framework that can model and express complex RBAC models such as OASIS. Park and Sandhu also consider X.509 certificates to address scalability issues in RBAC [PS99, PSA01].

RBAC in UNIX-like systems

UNIX-like systems support discretionary access control in their file systems. At a basic level these systems identify three types of subjects (owner, group, and other) and have three access modes (read, write, and execute (or search)). Users can belong to groups, and every file is assigned a permission list that indicates what the three subject types can do with it. Note that certain changes to the access control policy, such as changes to a user's group membership, are not reflected immediately in the user's rights. Such changes take effect only with the start of a new session. Therefore, access control decisions are made based on the resource's permission settings and the user's identity and group membership at session-start time. While users had freedom to change access rights to their files there was a large demand for additional flexibility. Many custom extensions exist that provide support for more expressive access control lists, and recently we have seen many initiatives to add RBAC support to UNIX-like systems [LS00, Sma00, Ott02, Fad99].

Among the extensions to add RBAC to UNIX-like systems is the work of Faden [Fad99], which describes an implementation for Solaris. Faden's work has many interesting ideas that inspired parts of our research. In his system, objects have security labels attached to them, but these labels are visible only to specific roles: roles to which the labels are relevant. Another interesting idea is to support privilege hierarchies instead of role hierarchies.

Feature summary of available RBACs

Table 2.1 summarises the features of the most popular RBAC models that were described in this section.

The last entry in this table is OASIS, an RBAC implementation that will be described in Section 2.3.

2.2.5 Policy management

There are many ways to represent policies; furthermore, these representations can differ in the level of abstraction at which they consider policies. Consequently, policies that describe the same access control restrictions, but at different abstraction levels, can form policy hierarchies [Wie94, AMN02].

As they are present in our everyday communication, natural languages seem to be the most obvious choice for specifying policies. However, natural languages are ambiguous and policies specified using them can be easily misinterpreted. Initiatives to restrict natural languages to an unambiguous subset lead to languages like Attempto Controlled English (ACE) [FSS98, FS96]. Such controlled English can be used to specify access control policies [Llo00, BLM01] and later translated into lower-level, logic-based policies.

Alternatives to natural languages for specifying policies at the user-level include visual tools [HMT$^+$90, Lup98, TP01]. Human users can easily understand visual representations of policies, especially as connections among the policy components, such as role-privilege mappings or hierarchies are easily and clearly presented. Unfortunately graphical policy specifications cannot express everything in a straightforward, easy-to-read way. Tidswell and Jaeger show how to handle and visualise various separation of duty constraints in their dynamic type based access control model [TJ00, JT01]; however, the area of constraint and conflict visualisation still requires much research.

	Sessions	Hierarchies	Constraints	Parameters	Negation	Delegation	Context	Decentralised	Extras Features
NIST	✓	✓	✓	–	–	✓	–	✓	
Ponder	–	✓	✓	✓	✓	✓	✓[a]	✓	Obligation policies
Jajodia et al.	–	✓	✓	–	✓	✓	–[b]	–	
Nyanchama-Osborn	–	✓	–[c]	–	–	–	–	–	Role graphs
NSPD (Giuri)	✓	✓	✓[d]	✓	–	–	✓	–	
PERMIS	✓	✓	✓[e]	–	–	✓	✓	✓	X.509
OASIS[f]	✓	–	✓	✓	–[g]	✓[h]	✓	✓	Appointments Fast revocation

[a]Extensible via libraries

[b]TRBAC allows temporal predicate evaluations.

[c]Static SoD is supported

[d]Supported in their extended model

[e]Temporal and delegation restriction constraints, but other constraints are supported only at the targets.

[f]OASIS will be discussed in Section 2.3

[g]OASIS can emulate negation via environmental predicates.

[h]OASIS supports delegation through its appointment mechanism.

Table 2.1: Overview of popular RBAC models.

At the lowest abstraction level of policy languages are formal, mostly logic-based languages. While these are difficult to understand they can be translated into more user readable forms. Many RBAC models use these languages as they are well suited for formal reasoning.

In [AS99] Ahn et al. describe such a language to specify RBAC policies. Their set theory based language, which can be translated into first order predicate logic, supports both dynamic and static separation of duty constraints. Other popular, formal languages include the specification language of TRBAC by Bertino et al. [BBF00], and the policy specification language of OASIS [BMY02].

No matter how policies are represented, they must be created somehow, and once created they must be maintained.

Role engineering

Organisational policies are complex. Nothing better illustrates this than the large number of RBAC extensions described in Section 2.2.3. Although RBAC continues to be extended to support more and more real-world policies, there are other aspects of RBAC that require attention. One area that needs further research is the initial creation of policies.

In [FH97] Fernandez and Hawkins show a method that applies use cases to determine roles of an organisation. Epstein and Sandhu approach this problem using UML [ES95], while Roeckle et al. show a process-oriented way to determine roles [RSW00].

The above methods already assume an existing organisational policy; consequently, they try to convert that into access control policies. But organisational policies are often vague, sometimes based on or extended with unwritten rules. In such cases new methods are needed to determine the policy. A proposal to address such cases with the help of statistical analysis and data mining of accesses is described in [KSS03].

Policy administration

Once roles and the access control policy are determined they need to be maintained – policies generally evolve! They change as the organisational policy changes, but also since the mapping of organisational policies to access control is rarely perfect, policies change as discrepancies are found.

Distribution only complicates the problem of policy administration. Large heterogeneous environments include many administrators, and thus the goal is not only to have a consistent policy but also to ensure that the policy is only modified by the administrators who are allowed to do so. Sandhu et al. addressed the issue of distributed policy management in [San00] and in their ARBAC model [SBC^{+}97] with the help of *sysadmin* sets and privileges that treat policies as resources.

Mönkeberg and Rakete investigated policy changes in highly dynamic heterogeneous environments [MR00].

Lupu considers policy life-cycles with three stages *(dormant, disabled, enabled)* to help policy evolution.

We shall discuss policy administration in more detail in Chapter 8.

2.3 OASIS

The *Open Architecture for Secure Interworking Services (OASIS)* [BMY01, HYBM00] is an RBAC model developed at the University of Cambridge Computer Laboratory.

Ever since the earliest designs, OASIS has been built with the requirement of supporting scalability and interaction in an open and distributed environment.

The first version of OASIS was based on the capability system paradigm [Hay96, HBM98], which is reflected throughout its later versions.

Currently OASIS is a powerful model that extends basic RBAC in a number of novel ways, while at the same time it incorporates most of the features of other RBAC models. The implementation of OASIS is founded on a formal, logic-based model [BMY02].

The virtues of OASIS include session support, prerequisite parameters, context awareness, flexible delegation with a fast revocation mechanism, and distributed operation and management. Next we provide an overview of most of its key features and look at how some of them are reflected in OASIS policies.

Environmental predicates

OASIS is *context-aware*, a feature of RBAC implementations that has recently been receiving much attention. Context-awareness is achieved in two ways. OASIS roles can carry parameters that depend on the environment in which they were activated, and, in addition, OASIS allows the use of *environmental predicates*. These predicates, evaluated at policy enforcement time, allow access to both static and dynamic information outside the OASIS system, and enable this information to be incorporated into the access control decision process . In this way it is possible to set role parameters based on the result of a complex database query evaluation during policy enforcement time.

Appointments

The importance of delegation is well reflected in the number of RBAC models that support it. In commercial organisations it is vital that the access control policy can express the transfer of subjects' rights to other subjects. However, in the real world such delegation is not the only form of rights transfer. There is often a need to transfer, or indeed authorise, rights that are not held by the assigner. For example, in a hospital a nurse can assign a doctor as a treating doctor to a patient, even though the nurse does not have the rights of a doctor. OASIS provides such an assignment of rights in the form of *appointment* [BMY02].

Appointment is thus an abstraction of delegation. It can express traditional delegation, but at the same time it can also express the transfer of further rights, rights that the appointer does not hold.

Similarly to environmental predicates, appointments can make use of parameters that help to aggregate similar appointments and thus result in more concise policies. In policies we use only the abstract notion of appointments together with their syntactical description. However, during policy evaluation the instance of such appointments is considered. These instances, called *appointment certificates*, are the issued appointments that represent a valid action of an appointment. As the name suggests, appointment certificates are digitally signed certificates. Their lifetime can span over multiple sessions, thus they are well suited to represent long term user qualities like qualification.

Moreover, appointment certificates can have additional restrictions to their use. They may contain preconditions that must be satisfied before the certificate is permitted to be used as a precondition to a role activation. This mechanism further increases the expressive power of policy specification that is available to policy administrators. Indeed, it allows the issuer to restrict the use of the appointment certificate independently of policy.

Closely related to delegation is *revocation*, through which a currently active role can be invalidated. Revocation can be implemented by a variety of techniques, of which the use of timestamps and expiry is probably the most popular. OASIS supports fast revocation through an event-based middleware.

2.3.1 OASIS policies

Access control policy in OASIS is specified by a number of logic-based rules in Horn clause form. There are two kinds of rules in OASIS. The first rule type describes authorisation, i.e. it maps roles to privileges. The second one expresses role activation, this maps users and other roles to roles. Both of these rules consist of two major parts, the head or target that is either a role or a privilege, and the prerequisites.

Authorisation rules

Authorisation rules map roles to privileges. Their form is as follows:

$$r, e_1, ..., e_{n_e} \vdash p$$

In these rules there is one and only one prerequisite role (r). Every authorisation rule assigns a single privilege *(p)* to its role. The only other prerequisites permitted in these rules are environmental predicates, these are denoted $e_k(k \in [0..n_e])$ above. As mentioned earlier these form a link to the environment that can be external to rule evaluation.

The role, the privilege, as well as the environmental predicates, can all be parameterised. These parameter values within a rule can be marked as *in* or *out* to reflect the information flow as individual predicates are evaluated (we mark *out* parameters with a '?', e.g. $roleA(x, y?)$).

We next provide an example rule:

$$treating\_doctor(x?, y?),\ check\_field\_td(f)\ \vdash\ read\_EHR(y?, f?)$$

The above rule assigns the *read_EHR* privilege to the *treating_doctor* role. The parameters of the *treating_doctor* are the identifiers of the doctor and his patient respectively. *check_field_td* is an environmental predicate that checks whether treating doctors can have access to the field specified as the predicate's parameter. If the result of this predicate evaluation yields true the privilege with the appropriate parameters is assigned to the *treating_doctor* role.

This rule expresses the policy that a treating doctor may read a particular electronic health record of his patient if treating doctors are allowed to read that particular record type.

Activation rules

The assignment of users to roles in OASIS is handled by *role activation rules*. These rules allow users to enter roles based on the user's possession of valid prerequisites.

In case of activation rules such prerequisites can be either a role, an appointment, or an environmental predicate. Thus, the format of role activation rules is as follows:

$$r_1, r_2, ..., r_{n_r}, ac_1, ..., ac_{n_{ac}}, e_1, ..., e_{n_e} \vdash r$$

The r_i, ac_j and e_k terms represent the n_r prerequisite roles, n_{ac} appointment certificates, and n_e environmental constraint predicates in this rule respectively – note that it is acceptable for any of n_r, n_{ac} or n_e to be zero. Predicate expressions on the left hand side of the rule are called *prerequisites*, and must be valid for a given user to activate r, the *target role*. Roles and appointment certificates are valid if they have not been revoked, environmental predicates are valid if they evaluate to true. Environmental predicates are powerful, and constraints such as separation of duty or cardinalities can be expressed with their help. Just as in the case of authorisation rules all the prerequisites can be parameterised.

OASIS, with its role parameters, environmental predicates, and revocation, supports the concept of *active security*, according to which access control decisions depend on context, which is monitored. Every prerequisite can be tagged to make it a *membership condition*. If a membership condition should become false subsequent to role activation, the target role of the rule in question is revoked. Since the revoked role could itself have been used as a prerequisite to activate other roles, its revocation can trigger further revocation, resulting in cascade. Mechanisms to achieve finer control over revocation behaviour can be found in [BE03].

We next provide an example activation rule:

$$local\_user(h\_id?),\ employed\_medic(h\_id?),\ on\_duty(h\_id)^* \vdash doctor\_on\_duty(h\_id)$$

This rule has three prerequisites to its target role *doctor_on_duty(h_id)*. The first prerequisite is a role *(local_user(h_id?))* with a single out parameter. The following prerequisite is an appointment with the name *employed_medic*. The specification of this appointment has a single out parameter. In order to evaluate this rule an appropriate appointment certificate, whose parameter matches the role parameter, must be provided. Finally, the last prerequisite is an environmental predicate, once again with a single parameter. When the rule is evaluated the validity of this environmental predicate is checked, which in this case most likely includes a database lookup to find out whether the person identified by its *h_id* parameter is on duty or not.

Note that the last prerequisite (*on_duty(h_id)**) is tagged as being a membership condition. As a consequence, the validity of the target role depends on the validity of this predicate;

therefore, when the doctor finishes his duty and the *on_duty* predicate is invalidated, the *doctor_on_duty* role is also revoked.

This rule expresses the access control policy that a person can act in the ‘doctor on duty’ role as long as he is on duty, has activated the ‘local user’ role, and has been appointed as an employed medic.

Since most OASIS roles are expected to be parameterised, OASIS does not support role hierarchies in an explicit manner. It is our belief that privilege inheritance expressed through a role hierarchy often improperly reflects the privilege assignment in organisations, thus its use can be misleading (see Section 2.2.1). Nevertheless, the rule system of OASIS can express hierarchical role relations with the help of role activation rules. However, in order to support other RBAC models in this thesis we also consider role hierarchies.

2.3.2 Service Level Agreements

Unlike the majority of RBAC systems OASIS is inherently *distributed*, as it is based on cooperating services that themselves can be distributed. The coordination and cooperation of these OASIS services is managed by an asynchronous publish/subscribe event architecture [BMB^{+}00].

Cooperation between OASIS services is governed by prespecified bilateral agreements. These are referred to as Service Level Agreements (SLA).

An SLA is bilateral contract between two OASIS services. It contains the specification of roles or other policy components that are exported to the other OASIS service. Apart from these, it contains information about a communication channel, which needs to be set up for event based notifications to handle role revocation. In the first OASIS implementation an SLA would also set up a heartbeat between the cooperating services. This heartbeat protocol helps to prevent certain attacks that try to compromise the network in order to disable role revocation messages. Different revocation behaviours are discussed in [BE03].

A problem with SLAs is that they are set up between only two OASIS services, and whenever the policy of at least one of these services changes the SLA must be updated. When the number of services is large and there is much cooperation among the services, a change to a policy can imply a large number of SLA updates. Each of these updates can require the policy administrators of both services to exchange information. In environments such as the National Health Service, where individual hospitals are expected to run their own OASIS services, and where cooperation is expected between hospitals, this is clearly infeasible. We propose a solution to this problem in Chapter 6, in which we investigate automatic SLA generation.

2.4 Problems with RBAC

While role-based access control solves many problems associated with MAC and DAC policy administration and it also retains important qualities of earlier access control models, it still requires much research.

For the last decade role-based access control has been considered as a promising alternative to other access control models such as MAC or DAC. In fact, there are many workshops and conferences focusing primarily on RBAC, and there is a large research community working on formal RBAC models and extensions to it. Industry also has favourable opinions about RBAC; for example Gartner lists RBAC among the best practises for security [Gar02].

But where is RBAC today? Why is it not everywhere in industry? There were many initiatives to adopt RBAC in industrial applications, and naturally many of these have failed. The successful implementations of RBAC (like in COM+ and .NET [HL03], and the access control systems of Dresdner Bank [SMJ01]) use only the core concepts of it, such as the grouping aspect of roles without the support for activity and parameters.

There are many reasons for RBAC's lack of proliferation:

Most RBAC models are too expressive. Although RBAC extensions are justified by important real-life scenarios, they require expressive policy languages. These can express simple access control concepts in a number of alternative ways. Each alternative will have implications for the future, consequently the policy administrator must make a decision. For example, in the case of OASIS policies one must decide whether to use environmental predicates, an appointment, or a role parameter to store a piece of information that is needed for an access control decision.

Complex policies are difficult to administer, and due to the complex semantics it is difficult if not impossible to ensure that a policy does what it supposed to do. The consequences of policy modification are difficult to foresee. The latter is especially problematic when many administrators – who most likely do not have a degree in policy maintenance – manage a policy.

Vague organisational policy. If an organisation has a well defined, precise policy why is it a problem to translate it to a policy language that is sufficiently expressive?

Policies of commercial organisations are not perfect. Even if they are written down, English is an expressive policy language, and can easily be misinterpreted. Poor and incomplete specifications lead to unwritten rules that complement organisational policies.

There are techniques that help to capture organisational roles (we discussed them in 2.2.5), but these are still insufficient.

Policy administration. Organisational policies do evolve. Accordingly, policies for access control systems must be updated. Such updates must preserve access control decisions that were made earlier, and goals that were achieved in previous policies or should be ensured in subsequent policies.

Currently there are no tools available to assist policy evolution in distributed environments. Administrators get no information about the consequences of policy changes. On the other hand, administrators often have to create ad hoc policies to support inter-organisational collaborations that involve resource sharing.

In this thesis we propose some solutions that can help policy administration.

3 Policy structure

In OASIS RBAC, as in some other RBAC models, a policy is basically a set of rules. Although the structure of these rules could differ in the various models, they share elementary components. Such components are, among others, data types, privileges, and roles themselves. Some models may contain extensions, but most of the time these extensions make use of the basic policy components.

In this chapter we introduce such basic policy components. We also provide a possible representation of these policy components in an object-oriented implementation.

These components build on the components of previous OASIS policy specification languages, but we extend them in a number of ways to support access control restriction to policies, long-term policy management, and policy evolution.

Although the components are seemingly very simple, policies built up from them can rapidly become complex. To assist policy administrators to structure and manage such policies, in Section 3.2 we present meta-policies. These contain constraints for policies, together with information that is not necessarily required at policy enforcement time. The objective of meta-policies is to help to preserve long-term goals and security requirements throughout policy evolution.

3.1 Basic policy components

Generally policies of large organisations are administered by more than one person. The number of such administrators can depend on the number of users, resources, and on the complexity of the policy. Giga Information Group – a leading IT advisory firm and a subsidiary of Forrester Research, Inc. – estimates that there will be one security administrator for every one to six thousand employees, depending on the quality of the RBAC implementation. These administrators may have different qualifications, competence, and trust, therefore we must provide means to control access to the various policy components. In this way access control policies become resources themselves; we must specify exactly *what* a policy is and *how* it is accessed, i.e. what the actions are that can be performed on a policy. These actions must be sufficiently fine-grained to support an expressive language that can restrict access control to policy components.

In this section we present the structure of the basic policy components that we expect to be part of every OASIS RBAC policy specification. The majority of these components also constitute the building blocks of policies of other RBAC implementations.

Note that many RBACs support policies that may contain constructs, such as role hierarchies, that are not explicitly present in OASIS. While these can be expressed in OASIS RBAC in an alternative way, we include appropriate extensions in our policy components, so maintaining their applicability to other RBAC models.

Knowledge of policy structure is vital for programmers, policy administrators and users. It is important for programmers because the structure presented here can serve as a template or a reference implementation. Policy structure is also reflected by the serialised forms of OASIS policies (for example, an XML policy specification or a Service Level Agreement (SLA)). Knowledge of policy structure is also necessary to use the API of our implementation (see Section 8.3).

Policy administrators need a good knowledge of policy structure, since it helps them to understand the methods that are used to modify policies. This is especially important when such policy components are used to restrict access to policies themselves (see Chapter 7).

Users need a limited knowledge of the policy components to protect themselves and their information. Many RBAC implementations allow users to interact with the access control enforcer, thus permitting the users to select the roles they want to enter, to select the rules that should be used for role entry or authorisation, or to select the credentials or prerequisites that should be considered in an access control decision. If the user has an idea of "what is going on" he or she can assign a trust level to the policy enforcing application, and based on such a trust level the user might hide or provide credentials that are needed to enter a specific role.

Our policy structure introduces an indirection layer that can be mapped to low-level database components such as tables or other storage entities, but at the same time provides an interface on which we can build reflexive control.

At the most basic level policy specification can be divided into two major parts. The first part of the specification concerns the most elementary building blocks, such as data types, roles, functions, and so forth. Their declarative specification provides the syntax to be used for these components. Generally, this includes the signature (i.e. parameter name and data type) of all elementary components.

A second part of the specification uses these building blocks to define the access control policy itself with the help of both activation and authorisation rules. Some previous OASIS policy specifications (e.g. [Hay96]) contained only such rule definitions, and the specification of the elementary building blocks was provided in an implicit way. This was clearly insufficient for the newer policy models that support strong typing and are connected to external services through well defined interfaces.

3.1.1 Data types

As already discussed in the section about OASIS (Section 2.3), policy components such as roles, environmental predicates and privileges can all be parameterised; thus, parameter type information may be included in policies.

Data type specification binds a name to a real data type that is known by the policy enforcer. This name can then be used in the type specification of variables and parameters.

Note that the data type specification presented here is basically just a label with some information about sub-types. Without functions this can only be used for type checking. Some standard functions are described in the next section, Section 3.1.2.

Policy components themselves use certain common data types. These common data types are therefore predefined for every policy, and they can be used in each policy specification. Currently there are only a few built in data types; among these is *boolean*, which is used, for example, as the return type of environmental predicates or to indicate validity of prerequisites.

3.1.2 Functions

Functions, as we shall see in Chapter 6, are used for handling parameters, e.g., converting parameter values from one representation to another. In our implementations functions are internal to the policy enforcer, i.e. functions cannot reference external services like web services or other OASIS services.

The return type of a function is a data type, and the parameters of a function are terms. In an object-oriented implementation a function could be represented according to Figure 3.1.

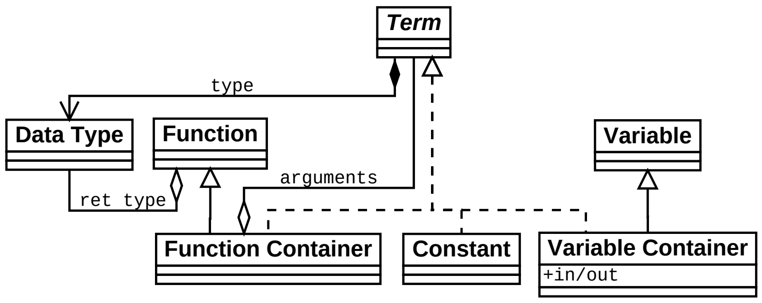


Figure 3.1: The structure of a term.

A *term* can be either a constant, the return value of another function or a variable (variables will be described with rules in Section 3.1.7). In our example implementation displayed in the figure the *term* class – as indicated by the italic typeset – is an abstract class.

As described in the previous section, data types, without supporting functions, can be used for static type checking only. There are a number of functions that are advisable to be specified for data types. Such functions might manage serialisation, convert values of the data type to strings, and vice versa. These functions can be used to embed constants as parameters in some policy components. They are also needed for transferring instances of policy components via text-based protocols, for example ones that use XML.

Comparison functions form another common set of functions. These compare values of certain data types, but most importantly, they can check for equality.

Whether a data type needs any of the above functions is primarily determined by the way it is used in the policy specification, but some functions could also be prescribed by global consistency rules (see Chapter 5).

3.1.3 Environmental predicates

Environmental predicates are similar to functions in the sense that they accept arguments and they return a value of a specified data type, in this case it is fixed to the built-in *boolean* data type. However, predicates may have side-effects, i.e. they can set parameter values – we call these parameters *out* parameters. Predicates can be viewed as a sequence of functions (defined over the parameters for which the predicate returns *true*) all with the same parameters (*in* parameters) and each returning an *out* value.

Unlike functions, predicates may also reference – and they often do – external services, thus they form a link to databases or to services provided by other applications.

The identification of environmental predicates within a policy is done the same way as in the case of functions, i.e. the name and parameter signature uniquely identifies an environmental predicate. Note that although the signature now also contains value binding information (*in/out*), we do not use this extra information to differentiate environmental predicates. Thus we do not differentiate between the following two environmental predicates:

$$treating\_doctor(doctor\_id, patient\_id?)$$
$$treating\_doctor(doctor\_id?, patient\_id)$$

These environmental predicates all concern information about patients treated by doctors. The first predicate returns the treated patients for a particular doctor, while the second returns the treating doctors for a particular patient. Since parameter binding is not considered to differentiate environmental predicates, we need to assign the above two predicates different names.

3.1.4 Privileges

Privileges describe the available resource accesses and the parameters for these accesses. Privileges are not method calls! They are tokens that are preconditions to certain protected method, procedure, or function calls. The parameters of privileges are needed for access control decisions only, therefore they should not contain information that is irrelevant for that purpose.

An example privilege could be $read\_EHR\_record(x : patient\_id, y : record\_type)$, where the two parameters are the identifier of the patient and the type of the electronic health record (e.g. 'haematology'). An example for an actual method that requires this privilege for its execution can be a method of a Java class $NHS\_patient$ of the form:

```
public String read_Field(Object fieldID, String returnFormat);
```

This method has different parameters from the privilege. For example, its `String returnFormat` parameter, which specifies formatting information for the return value, is not included among the privilege parameters as it is irrelevant from the perspective of access control. On the other hand, the patient identifier, which is implicitly part of the method via the object on which the method is called, must be included in the privilege.

Privilege components concern only the specification of privileges; they have information about names and parameter signatures only, and contain no role assignment.

Privilege inheritance

A possible extension to the above privileges is support for privilege inheritance. In this case we must augment the privilege specification with hierarchy information. This hierarchy can be easily simulated in RBAC, but having explicit support for privilege hierarchies allows for more compact policy specifications. If a parent and a child privilege are in a hierarchical relation, then the parent privilege can be used as a prerequisite in all the method calls where the child privilege is accepted as a prerequisite.

By using hierarchical relation between privileges we can express privilege dependencies, for example, a *write* privilege can only be assigned to a role with a *read* privilege. An algorithm to check non-parameterised privilege dependencies is described in [IO03] by Ionita and Osborn.

One difficulty, however, arises from parameters. It must be possible to convert the parent privilege with its parameters into the child privilege, which can have a different set of parameters. This mapping looks almost like a policy rule that treats a senior privilege as a prerequisite, and authorises inherited privileges automatically (i.e. a user has no control over this change).

By introducing rules that allow privileges as prerequisites the difference between roles and privileges starts to disappear. Indeed, privileges could even be considered as roles that cannot be prerequisites; the separation of the notion of privileges and roles is merely a convenience for administrators. There are models, for example PERMIS [CO02], that do not differentiate privileges from roles; indeed, there are access control models – like the capability-based access control – that handle roles and privileges as if they were one concept. Hierarchical privileges are basically functional roles with a restriction that they should not be used as prerequisites to organisational roles. They could prove to be a useful extension to RBACs that use organisational roles only; however, all RBACs could benefit from a clean separation of functional and organisational roles, even in the form of privilege dependencies.

3.1.5 Appointments

Appointments are unique to OASIS, however this concept replaces many "hacks" in other RBAC implementations that realise the need for more general delegation support. The semantics of appointments was described in Section 2.3, here we discuss their syntax only.

The appointments section of a policy specification declares the structure of the certificates used for appointment. This declaration specifies the name of the appointment, lists the parameters and their types, and provides additional information to bind this appointment to appointment certificates. This includes information about the consistency of an appointment certificate, and instructions about how to check this consistency (e.g. a list of accepted authoritative servers).

Note that appointment certificates are instances of appointments. They can contain additional prerequisites, but these are appointment instance specific, and thus are not part of a policy specification.

3.1.6 Roles

Roles are specified through a signature, which consists of the name of the role and the names and types of its parameters. In a policy description this information could be implicitly encoded in the rules, and indeed this has been done in the past. However, having a separate role definition gives information about unused roles as well; this information is needed for policy evolution and for meta-policies. In addition, to support strong typing it is vital to have a separate role declaration.

Role hierarchy

Role hierarchies are part of many RBAC models; in fact, there is more than one type of role hierarchy that can be supported by an RBAC model [Mof98]. It has often been suggested that role hierarchies are unnecessary and can cause problems [HV01, GB98, ML99].

OASIS resolves the above debate in a simple way. It does not support role hierarchy in an explicit form. However, it is possible to simulate role hierarchies by automatic role entry rules, i.e. with rules that have a single prerequisite role, the 'senior role', with no additional preconditions, and the 'child role' as target role. The addition and deletion of such rules is equivalent to the addition and deletion of role hierarchy relations, under the condition that there is no hierarchy management that maintains hierarchy relationships even if a role between two roles in the hierarchy is removed.

Such handling of role inheritance, similarly to the case of privilege inheritance, has the advantage of handling role parameters properly, i.e. propagating the role parameters between senior and junior roles.

3.1.7 Authorisation rules

Authorisation rules, together with activation rules, are the most important, but also the most complex, constituents of a policy. As described in the section about OASIS RBAC, these rules map roles to privileges. In the simplest cases, when no parameters are involved, the complexity of these rules arises from the variable number of prerequisites, i.e. a rule can have any number of prerequisite environmental predicates. This extra complexity requires the careful design of the data structures these rules are stored in, as well as the API to access these data structures (generally a single API call is insufficient to create a rule with arbitrary numbers of prerequisites, see Section 7.1.1).

Thus, the two main components of these rules are the prerequisites (a prerequisite can either be an environmental predicate or a role) and the target privilege.

An important requirement is the identification of rules. This was not part of the previous role definition languages of OASIS, but is very much needed for administrative, storage, and long-term evolutionary reasons. We therefore extend rules with an identifier that is unique within a policy.

Variables and containers

In general, authorisation rules are even more complex, and this complexity arises from their having variables. Apart from the simplest cases when all the prerequisites and the rule target lack parameters, rules contain variables. These variables are typed, and they can be used as parameters in the prerequisites. As they are rule specific, their scope is restricted to one particular rule.

Rules of a policy must satisfy certain constraints. One such constraint concerns free variables [BMY02]. As mentioned in Section 2.3 the occurrences of rule variables can be either *in* or *out*. A rule variable is bound in a rule if at least one of its occurrence is out parameter. A rule variable is free if it is not bound.

The constraint for an authorisation rule is that it contains no free variables. We assume that the rules used in the remaining part of this thesis are well-formed, i.e. they satisfy the above constraint.

In our implementation the main components of the rule are thus extended with variables. The structure of these rules is shown in Figure 3.2. Every parameter of the prerequisites and

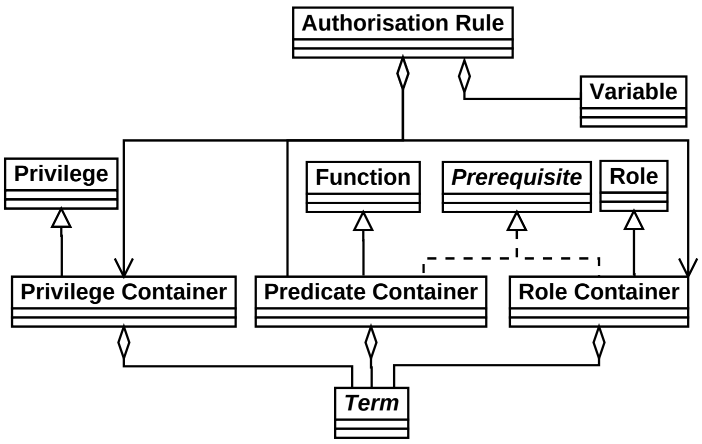


Figure 3.2: The structure of an authorisation rule.

the target privilege must be assigned to a term. Currently in OASIS this term is either a rule variable or a constant value. This assignment information is stored with the help of *Containers*. Note however, that containers can support the binding of parameters to function results, where the function is itself embedded into a container to fill its parameters. This extension will be important for the future extensions of policies (see Chapters 5 and 6), but it is also needed to support other RBAC models.

We next provide a short example for containers and variables. First we specify a role, an environmental predicate, and a privilege:

$$\begin{gathered} treating\_doctor(doctor : doctor\_id,\ patient : patient\_id) \\ check\_field\_td(fieldname : string) \\ read\_EHR(patient : patient\_id,\ fieldname : string) \end{gathered}$$

The rule $NHS1$ will have three variables $x : doctor\_id$, $y : patient\_id$, and $f : string$.

The rule has the following form:

$$NHS1 : treating\_doctor(x?, y?),\ check\_field\_td(f)\ \vdash\ read\_EHR(y?, f?)$$

In this rule there are two prerequisites and a target privilege. The role, environmental predicate, and the privilege are embedded into containers in this rule. The role container, for example, binds the rule variable x to the first parameter of the role (*doctor*), and the rule variable y to the second role parameter (*patient*). The container also specifies that these parameters are *out* parameters.

When policies are stored in plain text files there is an implicit ordering of rules. While in many models this has little significance, some models use such rule relations for conflict resolution. To support this we have added support for partial ordering of rules.

3.1.8 Activation rules

Activation rules control the activation of roles based on the prerequisites presented by the principal requesting the role – see Section 2.3. From the point of view of policy specification or policy structure, activation rules are much like the authorisation rules that were presented in the previous section.

Unlike authorisation rules, activation rules may have appointment certificates as prerequisites.

Similarly to authorisation rules activation rules may have no free variables.

The main components of activation rules are the target privilege, the prerequisites, and as in the case of authorisation rules, rule variables.

Each prerequisite, as described earlier, can be tagged to indicate whether it is a membership condition.

The structure of a possible object-oriented implementation of an activation rule is visualised in Figure 3.3.

In this representation membership information is stored in the containers that embed the prerequisites. Apart from this addition, the prerequisite containers work in the same way as for authorisation rules, i.e. they bind the prerequisite parameters to either variables or constant values.

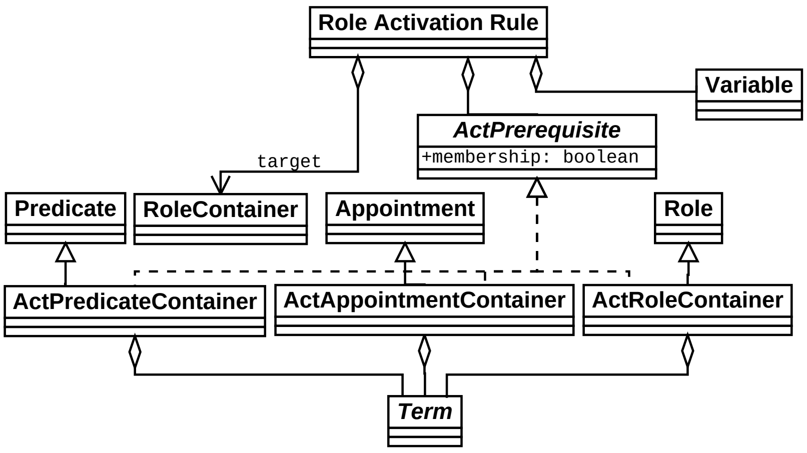


Figure 3.3: The structure of a role activation rule.

The following example activation rule contains three prerequisites (a role, an appointment, and an environmental predicate) and a target role. The specification of these is as follows:

$$local\_user(user : user\_id)$$
$$employed\_medic(medic : user\_id)$$
$$on\_duty(user : user\_id)$$
$$doctor\_on\_duty(doctor : user\_id)$$

The activation rule looks as follows:

$$NHS2 : local\_user(h\_id?),\ employed\_medic(h\_id?),\ on\_duty(h\_id)^{*} \vdash doctor\_on\_duty(h\_id)$$

In this rule the rule containers, similarly to those of authorisation rules, bind rule variables (in our case *h_id* is the only rule variable) to the rule component parameters, and indicate binding information, i.e. whether a parameter is an *in* or *out* parameter. In addition, activation containers hold information about membership conditions; in our case the container for $on\_duty(h\_id)$ records that the environmental predicate it contains is a membership condition.

More general membership conditions

In the simplest case a membership condition is just a tag that specifies whether a specific prerequisite is a membership condition or not, but the API produced as part of this research also supports possible extensions to membership conditions that can control revocation behaviour of prerequisites. We shall briefly consider such extensions in Section 4.6.3, but [BE03] contains a more detailed description. Basically, this extension gives finer control over OASIS server behaviour in unusual circumstances like network failure or a denial of service attack. Some OASIS implementations use a heartbeat protocol to keep informed about the state of a remote OASIS service. If such a heartbeat is lost, the validity state of the relevant membership conditions becomes unknown. The extra piece of information that is stored with the membership condition can control, according to the “importance” of the prerequisite, whether the prerequisite should be revoked immediately or in some delayed manner.

3.2 Extended policy components

In this thesis we extend the basic policy specifications to support other aspects of policy administration. These extensions we call *meta-policies*. They contain extra information and restrictions for a policy, but they do not necessarily form part of the policy that is expressed through activation and authorisation rules. Through meta-policies we divide organisational policies into access control policies at different abstraction levels. By structuring organisational policies in this way we support policy refinement and evolution in a way that is easy to administer, since policy changes, which occur frequently at the lower abstraction levels, can be performed subject to the control of parts of the policy at a higher abstraction level. We assume that the higher the abstraction level of a meta-policy is, the longer its expected lifetime will be.

As a naming convention in this thesis we refer to the policies of an organisation as *organisational policies*. These are most likely expressed in a textual form and govern various aspects of organisations. By the term *policies* we mean a set of RBAC or OASIS RBAC rules. The *meta-policies* we introduce specify some restrictions and a set of requirements for policies. Policies and meta-policies together are formed to represent organisational policies from the access control point of view.

The three kinds of meta-policies we shall discuss are *contexts*, *compliance policies*, and *interface policies*. In Figure 3.4 we illustrate how these meta-policies depend on each other.

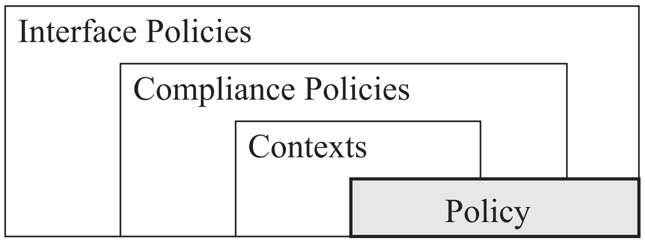


Figure 3.4: The relation among meta-policy types.

Each of these meta-policies contains a policy-independent and a policy-dependent portion. The independent part contains general definitions and the main concepts of the meta-policies, while the policy-dependent part makes use of the underlying policy model and contains binding information to policy and meta-policy instances. The higher the abstraction level of a meta-policy is, the more independent from a policy it is.

Although these meta-policies will be discussed individually in this thesis, a short summary is provided next:

3.2.1 Introduction to contexts

Policies can became large and complex, thus it is desirable to group policy components together and to be able to address such groups. This grouping is analogous to the concept of roles themselves, since at a basic level roles group permissions, and users. We introduce *contexts*, which can be viewed as a set of labels that are assigned to the policy components, including contexts themselves. Policy components that share a particular context label form groups. Every policy component can belong to a number of such groups, and this group membership can be taken into consideration in order to specify access control to a policy component, or to specify higher-level constraints over the policy in question. For a more detailed description of contexts see Chapter 4.

In terms of policy structure, contexts behave like very basic policy components. It is necessary to store the "labels", the basic constituents, and a dependency relation among these labels. We must also store the annotation of policy components, either with the policy specification by extending the policy components, or separately, in which case we must refer to the identifiers of the policy components. There must also be a few predicates to support contexts; these predicates are very much like the predicates associated with set management, they allow us for example to check whether a context label is included in a context.

The addition of contexts is really just an extension of the basic policy components. Contexts are not used at the time that access control decisions are made. The only time contexts are considered is when a policy is modified in some way, hence contexts form a meta-policy.

3.2.2 Introduction to compliance policies

Another extension to the basic policy components is a set of integrity rules, which we call compliance policies. These rules are checked for either the whole or a part of a policy specification. A policy that has been examined is said to comply with these rules if this compliance check evaluates to true.

The constraints and integrity checks expressed by compliance policy rules are specified over an abstract policy model, which is independent of the policies that will be checked against the compliance policy. For compliance checks policy administrators must provide a mapping of the abstract policy components to the components of the policy to be checked. The indirection introduced by this abstract policy model of compliance policies has many purposes:

First of all *re-usability*; the abstract model can be used for more than one policy. This is a desirable property, as many policies or policy parts are conceptually similar (component names and signatures may differ) and compliance policies can concentrate on a certain property, like, for example, separation of duty.

An example scenario where there are many conceptually similar policies is a set of hospitals, each with their own policies, but under the partial control of a higher-level organising group, like the National Health Service (NHS). The NHS may have a number of general rules, and local hospital policies, which are likely to differ from one another, can be required to comply with such general "guidelines" or compliance policies of the NHS.

This re-usability property does not concern only policies that exist in parallel, but also policy versions that follow each other in time. Policies evolve. Such evolutionary changes may include modifications to a role name and its parameters, new policy components, such as rules, and so forth. A compliance policy can capture important concepts of policies, concepts that should survive different policy versions, concepts that could be described as long-term goals of a policy. Then, via a compliance check, it can be ensured that these concepts are kept in the consecutive versions of policies.

Mappings between compliance policies and policies support much more than simple bijection, and thus they add to the expressiveness of the compliance policies, widening their applicability. This enables us to reconcile certain incompatibilities between high-level security requirements and the requirements set by legacy applications.

The mapping between the abstract policy model and the policy, as well as the successful compliance check, can form a part of a policy. This compliance can be used as a prerequisite for other compliances, but it can also provide information about the policy to both the users of the policy and its administrators, without their needing to know all the complex details.

Compliance policies will be described in detail in Chapter 5.

3.2.3 Introduction to interface policies

With the proliferation and rapid growth of the Internet, the need for inter-organisational cooperation has increased considerably. Our compliance policies capture certain aspects of policies that exist in parallel and possibly are distributed over a WAN. Such compliance policies thus describe something that is common across a set of policies. With our interface policies we extend compliance policies to facilitate cooperation between the management domains of different policies.

The RBAC model we use as our base model and as a testbed for our extensions – OASIS – supports the cooperation of policy domains that are spread over a network. Cooperation is managed by Service Level Agreements, which are bilateral point-to-point agreements. In the case of a large number of differing policy domains between which communication is common, the number of SLAs can become unmanageably large. An example scenario is, once again, a set of hospitals that store patient data in a distributed manner; access to such data from other hospitals is very desirable. In this environment, policy domains are heavily interconnected, and a change of a single policy can result in the update of many SLAs. This is expensive both in terms of human and material resources.

The rules of our compliance policies are specified on an abstract model that can be used as a mediator between two cooperating domains. If two domains with their relevant policies, comply with some abstract rules, the fact of this compliance gives useful information to both of these domains. This pair of compliances can be used to establish a connection between the policy components of the two domains.

Interface policies implement this concept. With the help of mappings – similar to those of compliance policies – certain roles of a domain can be converted into roles of another domain. In this way roles can be requested from a remote policy enforcer and be used to access remote resources based on roles in a local policy domain. The way the remote role is requested, together with the information that needs to be provided for this, is all specified in an interface policy.

We shall show how to generate SLAs automatically with the help of interface policies and corresponding policy mappings.

Our interface policies thus introduce an indirection between cooperating policy domains. An advantage of this indirection is the reduced administrative cost of SLA generation, as whenever a single policy is modified only the mapping between this policy and the abstract model needs to be updated, and the SLAs are regenerated automatically on request.

Interface policies are described in detail in Chapter 6.

3.3 Discussion

Every RBAC model has a set of components that are used to specify policies. Some of these building blocks are predicate based, some use objects; however, their general structure is similar. Unfortunately, most RBACs, including OASIS, were designed without consideration for policy maintenance; they had little support for strong typing, component declaration, and naming.

In this chapter we have introduced the policy components that we shall use in the remainder of this thesis.

We described the structure of our basic policy components, and we indicated how these extend the previous component models of OASIS policies. We shall use these extensions, which include naming, to add support for policy evolution and to simplify long-term policy administration.

Our components are OASIS specific, but they include extensions that enable them to express the policies of other RBACs. For example, they can express hierarchical relations, which are not part of OASIS policies.

Apart from policy specification there are two primary uses for our policy components. First, they are used to specify constraints over policy specifications. These constraints, which we call meta-policies, help long-term policy administration and policy evolution. Second, our component model will be used to specify a set of privileges in order to cater for access control restrictions to policies themselves.

The three kinds of meta-policies (contexts, compliance policies, and interface policies) and the policy administration privileges for self-administration will be described in the following chapters.

4 Contexts

Role-based access control organises and groups the users and privileges of organisations. This simplifies access control policies, since through roles policies can refer to aggregated component sets, as opposed to individual users and privileges. However, in the case of large organisations, where the number of users is measured in hundreds of thousands, the number of such aggregating roles could itself be very large.

Case studies like the one described in [SMJ01], that examine large commercial organisations, estimate the number of roles required to express the organisational policy to be well over one thousand. Even in the case of simpler policies, like the one described in [KKSM02], the number of roles is several hundred. All of these roles are mapped to users, privileges, or maybe other roles, and this mapping, usually in the forms of rules, also forms a part of the RBAC policy. In addition, roles and rules, as described in the previous chapter, rely on policy components such as data types, environmental predicates, variables, and so forth, thus further adding to the total number of components in a policy.

There are not many humans who could actually keep in mind and understand such large numbers of components. Naturally, such policies are managed by many administrators, and it is expected that they understand the changes made by their colleagues.

Real-world organisational policies are not specified in the form of activation and authorisation rules. When these policies are expressed, using a policy specification language, as a set of policy components, the logical binding – i.e. that these components belong to one “real-world” policy – is lost. As a consequence, policy specifications become flat, which makes them difficult to read for human users.

In this dissertation we propose many methods to address the problem of maintenance of large policies in the form of policy constraints or meta-policies. These meta-policies are not used directly for access control decisions but describe and restrict policies at policy specification time.

In an effort to preserve policy structure the meta-policy described in this chapter groups policy components together according to some logical coherence. We refer to such groups as *contexts*[1], and we annotate the policy components with a label that refers to such a context.

Contexts form a new policy component. While it might seem at first glance that contexts add to the complexity of a policy specification, they actually end up simplifying it, as they organise a policy into smaller coherent fragments that can be administered individually. In this way administrators with different trust, competence, or rights can manage portions of a policy without breaking other policies that are out of the scope of their responsibilities. For those who

[1]Note that this use of the term *context* is different from the use in *context awareness* that we talked about in Section 2.

administer a larger number of policy component groups, contexts help by providing views to a scope of policy components.

Policy component groups organised with the help of contexts are not isolated, as their corresponding real-world policy component groups not only overlap but relate to each other in some way. We express such relations with the help of information flow restrictions between contexts. This will enable us to specify restrictions over rules and thus enforce segregation of unrelated policy components.

Parts of the work described in this chapter were published in [BEM03]. This paper was written together with David Eyers, whose primary focus on contexts is motivated by how contexts can be used to specify work-flow management and separation of duty like constraints.

The author's motivation to explore contexts was primarily to group components for administrative purposes and to impose restrictions on rules. The joint work did not contain a formal specification of contexts, and the semantics of the information flow restrictions are completely revised in this thesis. These extensions are described in [BME04].

In this chapter we first define contexts in Section 4.1. We describe in detail the components that build up contexts, define the relations between contexts, and give examples of how contexts relate to each other. Next, in Section 4.2, we consider the context assignment to OASIS policy components, as in later sections we shall use OASIS RBAC policies to demonstrate the use of contexts. In the following two sections we explore the two primary uses for contexts. In Section 4.3 we investigate the grouping aspect of contexts, while in Section 4.4 we look at how we can use contexts to impose information flow restrictions on policy rules and what further constraints we can express with their help. We extend our context model to support hierarchical context elements in Section 4.5. In the subsequent section (Section 4.6) we show other uses for contexts. Finally we look at how contexts could be applied to other RBAC models, how our context model extends possible grouping aspects of those RBACs, and how others' work relates to ours.

4.1 The definition of contexts

We define a *context* as a finite set of labels. These labels are referred to as *context elements* and appear as $\mathcal{C}'$ in our formal descriptions[2].

In order to use contexts in a policy specification we first must define the permitted context elements, i.e. specify the context constituent labels. The only restriction for context elements is that they an equality predicate defined over them. As a convention we shall use strings using the Latin alphabet for context elements.

In the following we shall use lower case characters from the beginning of the Latin alphabet *(a,b,c, ...)* to denote context element variables, i.e. variables that can have a label value. For context variables, i.e. variables whose values are sets of context elements, we shall use lower case letters from the beginning of the Greek alphabet (e.g. α, β, γ).

Part of the context element specification is an information flow relation. This describes the direction in which information is permitted to flow between context elements. There are many ways to specify such a relationship between context elements. A trivial choice could be to treat the context elements as the nodes of a directed graph, and then list the edges of this graph. The final information flow relation between the nodes will be the transitive closure of the graph edges. Although this is seemingly the simplest way of specifying an information flow relation, in

[2]We shall use primes to differentiate the context notation in the model presented in the first part of this chapter. In Section 4.5 we shall extend and finalise this model, and use the symbols introduced here without primes.

this approach it is more difficult to add new, general context labels that allow information flow from or into any other node, since the relation with every other disconnected node group must be specified explicitly. It is also more difficult to specify fine-grained access control restriction to the graph edges.

We use a different approach here, in which we specify for every context element a set of context elements that it accepts information from, and a set of context elements it is willing to provide information to.

Formally this can be expressed with the following two functions:

$$\begin{aligned} \mathit{context\_in} &: \mathcal{C}' \rightarrow \mathcal{P}(\mathcal{C}') \cup \{\star\} \cup \{\varepsilon\} \\ \mathit{context\_out} &: \mathcal{C}' \rightarrow \mathcal{P}(\mathcal{C}') \cup \{\star\} \end{aligned}$$

The function *context_in* specifies for every context element the set of context elements it accepts information flow from, and *context_out* specifies in the same way for a context element the set of context elements it allows information to flow to. As loop information flow from context elements to themselves will be ensured by our information flow definition (Definition 4.1), information flow specification through *context_in* and *context_out* does not need to take care of the reflexivity of the information flow relation.

The range of both functions is extended by an additional element ($\star$) to support wild-cards, i.e. to specify that information from or to any context element is allowed. Using an extra symbol to refer to all the available context elements instead of listing all the context elements allows us to evaluate the set of available context elements dynamically. This is especially important for successive context element specifications.

The range of *context_in* is further extended by a special value ε. The value ε indicates that a context element serves as an initial context element (see Section 4.5.4).

An explicit edge between two context elements in the information flow graph will exist if the set of context elements to which the source node permits information flow includes the target node, and the set of context elements from which the target node accepts information flow includes the source node.

The above is expressed formally with the help of the $\hookrightarrow$ relation:

Definition 4.1 (Direct information flow between non-hierarchical context elements) $\hookrightarrow \subseteq \mathcal{C}' \times \mathcal{C}'$

$$\begin{aligned} a \hookrightarrow b \quad \Leftrightarrow \quad & (a = b) \vee \\ & \Big(\big(b \in \mathit{context\_out}(a) \vee \star \in \mathit{context\_out}(a)\big) \wedge \\ & \big(a \in \mathit{context\_in}(b) \vee \star \in \mathit{context\_in}(b)\big) \Big) \end{aligned}$$

The final information flow graph is produced by the transitive closure of the direct edges. We shall use the symbol $\hookrightarrow^*$ to denote the relation that represents information flow between context elements.

An advantage of handling context information in the way presented above, as opposed to storing just the edges, is that it allows a more flexible administration of context elements. Context elements that use wild-cards do not need to be updated when new context elements are added or removed, thus from the access control point of view no access to these elements is necessary. Also, principals responsible for certain context elements can specify information flow without access to other context elements.

4.1.1 Example specification of context element

We next give an example in which we illustrate how to specify information flow using the *context_in* and *context_out* functions. The context elements are shown in Figure 4.1:

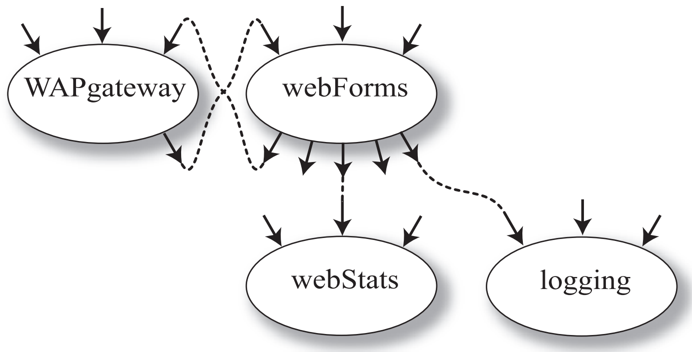


Figure 4.1: Example information flow among context elements.

In this example there are four context elements: `WAPgateway`, `webForms`, `logging` and `webStats`. The context element `webForms` specifies that any information flow will be accepted either in or out of this context element. With the help of the *context_in* and *context_out* functions this is expressed as:

$$\begin{aligned} \textit{context\_in}(\texttt{webForms}) &= \star \\ \textit{context\_out}(\texttt{webForms}) &= \star \end{aligned}$$

The context element `logging` can be considered as "mode in". It specifies an empty set as possible context elements it permits information to flow to, thus information may flow only toward the `logging` context element.

$$\begin{aligned} \textit{context\_in}(\texttt{logging}) &= \{\} \\ \textit{context\_out}(\texttt{logging}) &= \star \end{aligned}$$

The context element `webStats` allows only `webForms` as information source, and similarly to `logging`, it does not allow any information flow out of itself.

$$\begin{aligned} \textit{context\_in}(\texttt{webStats}) &= \{\texttt{webForms}\} \\ \textit{context\_out}(\texttt{webStats}) &= \{\} \end{aligned}$$

`WAPgateway` allows information flow from any source into only `webForms`.

$$\begin{aligned} \textit{context\_in}(\texttt{WAPgateway}) &= \star \\ \textit{context\_out}(\texttt{WAPgateway}) &= \{\texttt{webForms}\} \end{aligned}$$

Because the effective information flow graph is the transitive closure of the specified graph, information from the `WAPgateway` may flow into the `webStats` context element, even though these were not directly connected.

4.1.2 Cycles in the information flow graph

The information flow graph, defined by the information flow relation for context elements, may contain directed cycles. As the effective information flow is the transitive closure of the information flow relation, information may flow from any context element to any other context element on the same directed cycle. Thus in the example presented in Section 4.1.1 the `WAPgateway` and `webForms` context elements will behave in the same way.

Context elements on a directed cycle are equivalent from the information flow perspective, but from a policy structuring perspective these context elements differ. For example, access control to the policy can be specified according to the context elements, so one could permit a role to modify policy components tagged with the `webForms` only.

Also, because of policy evolution, apparently equivalent context components could become significantly different in the future.

For such reasons we allow cycles in the information flow specification; however, to avoid accidental cycles we suggest that context elements that are open on both sides – i.e. that allow information to flow freely both into them ($context\_in(...) = \star$), and out of them (($context\_out(...) = \star$) – are avoided.

A possible alternative to specifying context elements that accept information from any other domain is to mark them as initial (ε), but this we shall discuss later.

4.1.3 Information flow relation between contexts

As mentioned previously, policy components can be associated with contexts, i.e. with more than one context element. The assignment of a set of labels is very similar to the assignment of a single label. If only one context element were allowed per policy component then these could be created from the powerset of the context labels, and the information flow relation in the new scheme could also be deduced.

The advantage of having a *set of labels* assigned to a component is that it allows policy components to be segmented along multiple dimensions. For example, one dimension distinguishes components based on their belonging to either procurement or marketing components, and another dimension can distinguish components on their security classification like 'web' or 'secure web'.

Based on information flow restrictions between context elements we can specify information flow between contexts.

If α and β are two contexts, then information flow between them is permitted only if:

1. for every context element in α there is a context element in the target context β into which information may flow, and
2. for every non-initial context element in β there is a context element a in the source context (α) from which information flow is permitted to the target context element.

Formally this is defined by the following relation:

Definition 4.2 (Information flow between non-hierarchical contexts)
$\hookrightarrow \subseteq \mathcal{P}(\mathcal{C}') \times \mathcal{P}(\mathcal{C}')$

$$\begin{aligned} \alpha \hookrightarrow \beta \quad \Leftrightarrow \quad & (\forall a \in \alpha : \exists b \in \beta : a \hookrightarrow^* b) \wedge \\ & (\forall b \in \beta : ((\exists a \in \alpha : a \hookrightarrow^* b) \vee (\varepsilon \in context\_in(b)))) \end{aligned}$$

Note that this definition allows information flow between two empty contexts. This special case we shall discuss in more detail in Section 4.2.1.

Thus, based on the example in Section 4.1.1 information flow from the context `[WAPgateway, webForms]` to `[webForms, webStats]` is permitted, as information from both `WAPgateway` and `webForms` may flow to both `webForms` and `webStats`.

A counterexample is the contexts `[webForms, logging]` and `[webStats]`; information flow from the first context to the latter one is not permitted because, although information flow from `webForms` to `webStats` is permitted, information cannot flow from `logging` to `webStats`.

An example that uses ε in *context_in* is provided in Section 4.5.5.

The above definition has two consequences that are identified in the following two lemmas:

Lemma 4.1 *If α and β are two contexts between which information flow is permitted ($\alpha \hookrightarrow \beta$), and α has a context element ($a : a \in \alpha$) from which information may not flow anywhere else ($context\_out(a) = \{\}$), then this context element must be in β.*

Lemma 4.2 *If α and β are two contexts between which information flow is permitted ($\alpha \hookrightarrow \beta$), and β has a context element ($b : b \in \beta$) into which information may not flow from anywhere and it is not initial ($context\_in(b) = \{\}$), then this context element must be in α.*

Parallel context flow structures

These lemmas enable parallel information flows for two context element sets that have no information flow permitted between each other. For example, in Figure 4.2 there are two context element groups that can describe two different information flows. One group has a `web` $\hookrightarrow$ `secureWeb` information flow restriction. This context element group takes care of information flow from the web security aspect. The second group, consisting of the `CompLab`, `OPERA`, and `Security` context elements, describes information flow from an organisational perspective. The two context element groups thus describe two separate, orthogonal information flows.

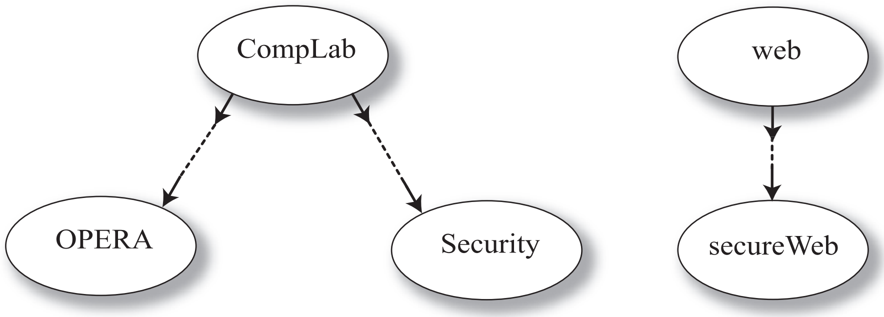


Figure 4.2: Parallel information flow example.

Components of both classifications will have to propagate one context element from each group. Thus, for example a role that is marked with the {`web,OPERA`} context can only be used as prerequisite to roles that have at least one context element from the {`web, secureWeb`} and one element from the {`CompLab, OPERA, Security`} context element sets.

4.2 Context assignments

We introduced contexts in order to annotate policy components with them. In this section we examine at syntactic level which OASIS policy components can have context information associated with them. We shall briefly consider other policies in Section 4.7.

We allow the annotation of data types with contexts. Similarly, other basic components of a policy, such as function, role, appointment, and privilege specifications, can all be assigned contexts, but as these components use data types in their specification, they have two groups of context associated with them. The first is the context that is assigned directly to the component, the second refers to the contexts of the data types used.

The most complex policy components are activation and authorisation rules. These rules can have contexts associated with many of their constituent parts. In the case of activation rules these parts are depicted in Figure 4.3. Note that rules themselves can be associated with a context.

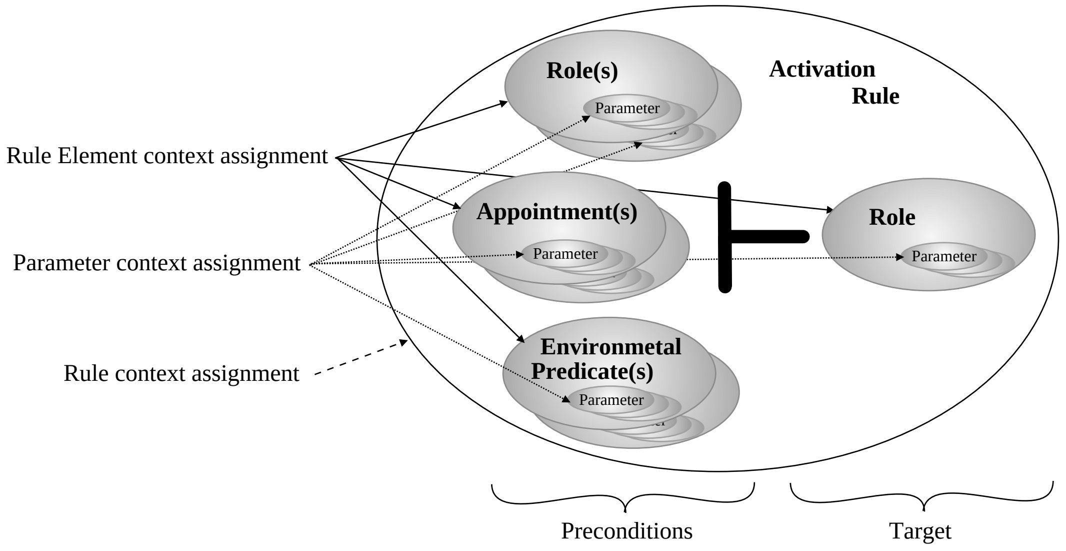


Figure 4.3: Role activation rule components that can be assigned a context.

To enable fine-grained information flow restrictions we allow context specification at sub-component level, e.g. we allow context specification for prerequisites that are parts of a rule (which itself is a policy component).

Rule components such as roles, appointments, environmental predicates, and privileges can be assigned contexts at the time they are defined. In rules, these rule components are encapsulated into containers (see Section 3.1.7) that are not permitted to have contexts assigned to them directly. The containers have effectively the same contexts associated with them as the role, appointment, environmental predicate, or privilege that they encapsulate.

Parameters of rule components can also have contexts assigned to them. This allows the differentiation of parameters within prerequisites. For example, in the case of the privilege *read_EHR_record(patient_id, record)* the two parameters can be differentiated from an information flow perspective. Assuming that the *patient_id* parameter, which uniquely identifies a patient, is more sensitive than the *record* parameter, which can have values like 'haematology', the propagation of *patient_id* should be more restricted.

Rule variables, in the same way as containers, cannot be assigned contexts. The contexts associated with them will be the contexts of the role component parameters that are bound to these variables.

To show what context a policy component belongs to we use a superscript notation within square brackets. For example, to indicate that *RoleA* has the context elements `web` and `secureWeb` assigned to it, we write $RoleA^{[\texttt{web},\texttt{secureWeb}]}$. Thus an activation rule can have the following form:

$$rule\_name^{[rulec]} : r_1^{[rc_1]}, r_2^{[rc_2]}, ..., r_{n_r}^{[rc_{n_r}]}, ac_1^{[acc_1]}, ..., ac_{n_{ac}}^{[acc_{n_{ac}}]}, e_1^{[ec_1]}, ..., e_{n_e}^{[ec_{n_e}]} \vdash r^{[rc]}$$

where *rule_name* is the name of the rule, r_i, ac_j and e_k are the n_r, n_{ac} and n_e prerequisite roles, appointments, and environmental certificates respectively, and r is the target role. The contexts are in superscript in square brackets, *rulec* is the rule context, while rc_i, acc_j, ec_k are the prerequisite contexts; finally, rc is the target role context. The parameters of each rule component can also be assigned a context, thus any of the above rule components can look like:

$$rule\_component^{[component\_context]}(parameter_1^{[context_1]}, ..., parameter_n^{[context_n]})$$

Note that while we shall use the above form of a rule in our examples, it is not intended for policy administrators. Note also, that rules are only associated with rule contexts. Since containers have no contexts, the rule component context and parameter context in the above forms of rules are specified with the rule component specifications, i.e. outside the rule specification.

As the activation and authorisation rules illustrate, rules as well as other policy components, can have many parts associated with contexts. Which of these contexts must be considered when a decision is made, or whether the union of the contexts must be considered, depends on the purpose the contexts are used. These purposes, as well as how contexts are used to support them, will be explained in the next sections.

4.2.1 If there is no context association

Contexts form a simple meta-policy, thus they are not required in a policy specification. Policies may exist with no or only partial context specification.

Definition 4.2 allows information flow between two empty contexts, therefore information may flow between policy components that have no contexts associated with them.

Information is permitted to flow from empty contexts to non-empty contexts only if all the context elements are initial ones, i.e. they have ε in their *context_in*.

The reason for segregating components with and without contexts is to avoid the use of 'unchecked' policy components (ones that have no contexts associated with them) together with 'checked' ones (ones with context). Initial context elements are oblivious to the information source; they indicate a new piece of information.

The second part of this segregation, which says that information may not flow from non-empty contexts into empty ones, is expressed by the following lemma:

Lemma 4.3 (Information flow from non-empty contexts)

$$\alpha, \beta \in \mathcal{P}(\mathcal{C}') : \alpha \neq \{\} \wedge \alpha \hookrightarrow \beta \;\Rightarrow\; \beta \neq \{\}$$

This is an immediate consequence of the information flow definition.

4.3 Component grouping

By annotating policy components with contexts we form policy component groups. These groups provide a view to the policy, which can be used for several purposes, for example structural component grouping can be used both to define coherent portions of a policy and for access control.

4.3.1 Structural groups

First of all, structuring introduced by contexts allows administrators to work with a smaller number of policy components.

For example, policy components that are used for roles of a specific ward in a hospital, say roles for haematology, can be grouped with the help of a context element called '`haematology`'. Policy components that are annotated with this context element, i.e. whose contexts include the context element '`haematology`', will belong to one specific group.

Such groups can be freely formed. One of their primary purposes is to structure a policy for readability reasons. This is especially useful if policies are stored in a policy store as opposed to plain text files. These groups can later be visualised by appropriate graphical user interfaces.

Consistency requirement

Most policy components are built up from other components, and many of these can be labelled with contexts. For example, roles can have contexts associated with them but the parameters of a role can also be associated with contexts. As a role encapsulates its parameters it cannot be separated from them.

Therefore, to maintain consistency, we impose a restriction on the context specifications that are permitted, and *require that subcomponents of a policy component belong to the context of their parents*. For example, the context of a role parameter must be a subset of the parent role's context.

Data types are not considered to be child components of policy components, thus the above requirement does not apply to them.

4.3.2 Grouping for access control

In Chapter 7 we introduce a mechanism for fine-grained access control to policies and to their constituent components. This mechanism is based on privileges that are associated with every basic operation on policy components. In the case of small policies, such access privileges could be used efficiently to specify access control to policies, but when a large number of policy components is involved this approach becomes impractical.

Privileges concerning access control to policy components can take advantage of context-based component groups. Thus, via such policy component groups, specifying access control to policies can be expressed in a more concise form.

As mentioned in the case of structural grouping, the contexts of policy components that are constituent parts of another component must include the context labels of their parents. This property ensures that in the case of context-based access control, a principal that wishes to modify a part of a policy component must also have modification rights for the parent component. For example, if a rule has the context {`web, secureWeb`} the constituent elements must also have these context elements associated with them. Thus, the modification of a rule component – say a prerequisite role – will also require privileges to modify elements in the {`web, secureWeb`} contexts. Note that this restriction does not extend to data types.

However, child components can belong to contexts that the parent component does not have. Reasons for this will be discussed in the section about information flow restrictions (Section 4.4). From the access control perspective this means that if a principal has rights to modify a policy component, he still might not be able to modify specific subcomponents of it. This can be useful, for example, in the case of an activation rule, where a principal may be authorised to change only specific prerequisites of the rule.

4.3.3 Self-administration of contexts

Since in our model contexts form a meta-policy, they are intended to be used only at policy specification time. Thus contexts need not be part of a policy at policy enforcement time. They only help to administer a policy. But it is natural to store contextual information with the policy, forming in this way a new policy component.

One motivation for contexts was the specification of evolutionary access control restrictions to policy components. It is expected that the lifetime of contexts and meta-policies will be much longer than that of individual policies; thus the maintenance of meta-policies, including context element specification, should occur less frequently than the modification of the policy itself. Still, policy contexts can be modified, and access control restriction to such modification must be specified. The privileges associated with access to contexts can make use of contexts themselves, making contexts self-administering. It is important to note that self-administration as described in this thesis allows only the creation of a new version of a policy. Thus the access control rights of a principal that is modifying a policy are not affected after each modification, but only after a consistent set of modifications, when a new policy version is committed.

Predefined context elements

There is a set of predefined context elements available by default to policy administrators. These context elements have a special form, in that they start with the keyword 'desert'[3]. For example, the '`desert.administration`'[4] context is used to mark policy components (mainly privileges) that concern policy modification.

4.4 Information flow restrictions

In this section we look at how to specify information flow among policy components, or in other words, how to restrict the use of certain policy components. First, in Section 4.4.1, we show our motivation for such use of contexts. Later we discuss the information flow restrictions at three different granularity levels. These three levels correspond to the three categories of rule constituents that can be assigned contexts. As depicted in Figure 4.3 on page 63, rule components such as prerequisite roles, appointments, and environmental predicates can have contexts associated with them. Information flow restrictions for these policy components will be described in Section 4.4.2. Later, in Section 4.4.3, we narrow our focus and look at the information flow between subcomponents of rule components, i.e. we look at individual rule component parameters. Finally, in Section 4.4.4, we look at how information flow constraints work at rule level.

4.4.1 Motivation

Grouping certain policy components and specifying information flow restrictions for such groups enable us to treat such component groups differently. Some rule targets may require stricter preconditions in order to be acquired or entered. For example, in a policy with two context elements `web` and `secureWeb`, it might be required that roles in the `secureWeb` contexts cannot be used as prerequisites to roles that belong to the `web` context. With the help of contexts and the information flow relation between context elements, we can express such policy specification time restrictions and decouple this restriction from the policy itself. In this way we can control

[3]DESERT is our meta-policy management tool.

[4]Hierarchical context elements, such as `desert.administration` will be explained in Section 4.5.

which policy components – belonging to a specific context – can be used as prerequisites to policy components of another group.

Apart from the simplest policy specifications, most policy rule components, such as roles and appointments, contain parameters. While parameters form a useful extension they also increase the sophistication of policies. Parameters hold information. In rules the parameter values can propagate to other policy components, and finally to privileges. For example in the rule

$$local\_user(h\_id?),\ employed\_medic(h\_id?),\ on\_duty(h\_id) \vdash doctor\_on\_duty(h\_id)$$

the *h_id* parameter of the *local_user* role and the *employed_medic* appointments can propagate into the target role *doctor_on_duty*.

Whether parameter information is sensitive or not depends on the environment in which it is used. For example, the patient identifier parameter that can be part of most roles in the National Health Service policy should not be included in a role like *HIV_test*, that authorises its holders to have them anonymously checked for HIV infection. Note that a role can have both sensitive and insensitive parameters.

The information does not only propagate to other policy components but, in OASIS via environmental predicates, also to the external world. Environmental predicates form a link to applications external to the policy enforcement engine, and information flow into such external entities must also be controlled. An additional complexity is that this external link, due to the fact that environmental predicates can set parameters, also forms an information source.

We expect valid policy rules to satisfy information flow restrictions. This will be discussed in more detail in Chapter 8.

4.4.2 Context specification for rule elements

Rule elements, roles, appointments, environmental predicates, and privileges, can all be given contexts. Information flow restrictions among these contexts control what can be used as a prerequisite for the target role or target privilege in a rule.

The information flow restriction thus specifies that in rules – both activation and authorisation rules – a rule component can be used as a prerequisite to the target component only if information flow is permitted from the prerequisite's context to the context of the target component.

Example

This next example shows how to create two or more contexts that cannot be used as prerequisites of each other.

The context elements are `CompLab`, `OPERA`, and `Security`. Roles are *local_user(uid)*, *OPERA_-webadmin(uid)*, *OPERA_meetingadmin(uid)*, and *Security_webadmin(uid)*. The `CompLab` is used to mark policy components that are under the direct control of the Computer Laboratory. `OPERA` and `Security` are used to mark components that are managed by the two research groups, the OPERA and the Security group. There is also an environmental predicate – *group_member* – available that checks whether a particular user is in a specific research group.

The information flow restriction among the context elements is given and visualised in Figure 4.4.

In this example context element specification, information is allowed to flow from the `CompLab` context element to any other context element – in our case these are the `OPERA` and `Security` context elements; and no other information flow is permitted.

$$\begin{aligned}
context\_in(\texttt{CompLab}) &= \{\} \\
context\_out(\texttt{CompLab}) &= \star \\
context\_in(\texttt{OPERA}) &= \{\texttt{CompLab}\} \\
context\_out(\texttt{OPERA}) &= \{\} \\
context\_in(\texttt{Security}) &= \{\texttt{CompLab}\} \\
context\_out(\texttt{Security}) &= \{\}
\end{aligned}$$

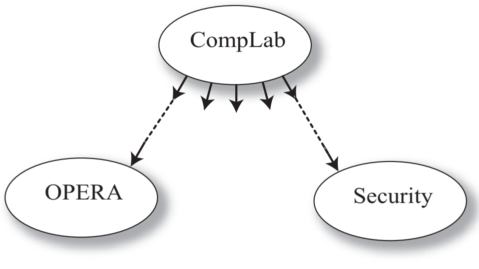


Figure 4.4: Information flow example for rule components.

The assignment of contexts to roles is as follows:

$$\begin{aligned}
&local\_user^{[\texttt{CompLab}]}(uid) \\
&OPERA\_webadmin^{[\texttt{CompLab,OPERA}]}(uid) \\
&OPERA\_meetingadmin^{[\texttt{OPERA}]}(uid) \\
&Security\_webadmin^{[\texttt{Security}]}(uid) \\
&group\_member^{[\texttt{CompLab}]}(uid, group)
\end{aligned}$$

We next present some example rules that satisfy the above information flow requirements. These rules follow the guideline expressed by the meta-policy, according to which a policy may contain rules that use Computer Laboratory roles as prerequisites to the research groups' roles or privileges.

$$\begin{gathered}
local\_user^{[\texttt{CompLab}]}(uid?), member\_of^{[\texttt{CompLab}]}(uid,\ 'OPERA') \vdash \\
OPERA\_webadmin^{[\texttt{CompLab,OPERA}]} \\
local\_user^{[\texttt{CompLab}]}(uid?), member\_of^{[\texttt{CompLab}]}(uid,\ 'Security') \vdash \\
Security\_webadmin^{[\texttt{Security}]}(uid)
\end{gathered}$$

The information flow graph restricts the use of roles in the `OPERA` context so that these roles can be used as prerequisites only to roles in the `OPERA` context. Thus the rule

$$OPERA\_webadmin^{[\texttt{CompLab,OPERA}]} \vdash OPERA\_meetingadmin^{[\texttt{OPERA}]}(uid)$$

is permitted, but the rule

$$OPERA\_meetingadmin^{[\texttt{OPERA}]}(uid) \vdash Security\_webadmin^{[\texttt{Security}]}(uid)$$

cannot be part of the policy specification, because information is not allowed to flow from the `OPERA` context element to the `Security` context element. In other words, the roles of the OPERA group cannot be used as prerequisites to roles of the Security group. Note however, that a role from the OPERA research group can still be used for shared roles, i.e. for roles that are marked at specification time as belonging both to `OPERA` and `Security`.

4.4.3 Context specification for parameters

Parameters of rule components can have different levels of sensitivity themselves. For example, in a role *patient('Alice', 'burns')* with two parameters, one of which is a patient identifier, the second just a ward identifier, the first parameter is likely to be more sensitive than the second one. While it is acceptable to use the second parameter as an information source for specific roles, it might be required that the information held by the first parameter is not propagated.

We therefore allow parameters to belong to contexts that their parent components do not share. Information from such parameters with extra context elements may propagate only into parameters whose context includes context elements into which information may flow from the extra context element. Note that to check information flow restrictions only the parameter contexts need to be considered, since the context associated with the parameter's parent component must be a subset of the parameter's context.

Example parameter information flow

In the next example there are three context elements involved: `personal`, `hospital`, and `HIV_test`. Components marked with `personal` indicate that these components handle or contain information that is personal to a patient. The `hospital` context element is used to mark hospital specific policy components. The `HIV_test` context is used to mark policy components that are used for HIV screening. To ensure anonymity it is expected that components marked with `HIV_test` contain no personal information.

Information is permitted to flow from `hospital` to `personal` and `HIV_test`, and from `HIV_test` to `personal`. This is visualised in Figure 4.5.

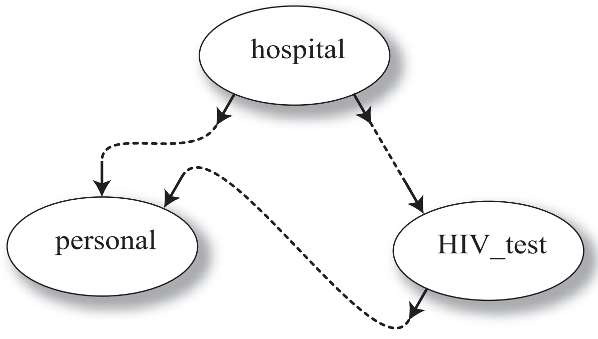


Figure 4.5: Example information flow for parameters.

The specification of the role involved is:

$$patient^{[\texttt{hospital}]}(patient\_id^{[\texttt{hospital,personal}]}, ward\_name^{[\texttt{hospital}]})$$

It can be seen here that the role has the `hospital` context element associated with it, but one of its parameters (*patient_id*) is – in addition to the role context – also associated with the `personal` context element to indicate that it represents personal information.

There is also a privilege in this example; its specification is:

$$HIV\_testable^{[\texttt{hospital,HIV\_test}]}(some\_parameter^{[\texttt{hospital,HIV\_test}]})$$

This privilege can be used for example to issue a magnetic card to a patient so that he may prove that he may be tested for HIV free of charge. Here the privilege has two context elements (`hospital, HIV_test`) in its context. Its parameter, which for the sake of this example is called just *'some_parameter'*, has the same contexts as its parent component, the privilege.

For simplicity, all the parameters involved in this example have "*character string*" data type (which, we assume, is defined in the policy specification).

Let us consider the following authorisation rule:

$$patient(x,\ y) \vdash HIV\_testable(y)$$

Or with the contexts shown:

$$patient^{[\texttt{hospital}]}(x^{[\texttt{hospital,personal}]},\ y^{[\texttt{hospital}]}) \vdash HIV\_testable^{[\texttt{hospital,HIV\_test}]}(y^{[\texttt{hospital,HIV\_test}]})$$

To test whether this rule complies with the information flow restrictions the following tests are performed:

✓ *Rule component level tests: checks whether the prerequisites can be used for the target component.*

In this case it is checked that information may flow from `[hospital]` – the context of the prerequisite role *patient* – to `[hospital]`, the context of the target privilege *HIV_testable*.

✓ *Component parameter level tests: checks whether the prerequisites can be used for the target component.*

In this case it is checked whether information is allowed to flow from the second parameter (y) of the *patient* role to the first parameter (y) of the target privilege. In this case the test passes, as information is permitted to flow from the relevant parameter contexts, i.e. from `[hospital]` to `[hospital,HIV_test]`.

The following rule is an example when the parameter level information flow restriction is not satisfied.

$$patient(x,\ y) \vdash HIV\_testable(x)$$

Or with the contexts shown:

$$patient^{[\texttt{hospital}]}(x^{[\texttt{hospital,personal}]},\ y^{[\texttt{hospital}]}) \vdash HIV\_testable^{[\texttt{hospital,HIV\_test}]}(x^{[\texttt{hospital,HIV\_test}]})$$

This rule is not accepted, as according to the rule the value of the first parameter (x) of the prerequisite role propagates to the first parameter of the target privilege, but the information flow between the relevant contexts – `[hospital,personal]` to `[hospital,HIV_test]` – is not permitted (information may not flow from the `personal` context element to either `HIV_test` or `hospital`). This result is not unexpected, as there is no information flow from the `personal` context element, and according to Lemma 4.1, it is required that this context element is present in the target component's context.

Taxonomy of environmental predicates

Environmental predicates, in addition to being able to read and to set parameter values, can have side-effects. They form a link to the application outside the policy decision making engine, therefore it is prudent to provide a means to specify their information flow characteristics.

Extra care must be taken with environmental predicates and with the information that flows into them. When other prerequisites (roles and appointments) are evaluated, the information stored in their parameters does not leave the policy enforcement engine. In the case of environmental predicates the *in* parameter values are sent to the service that evaluates these parameters and sets any *out* parameters. Therefore it is imperative to know and to be able to specify what information may flow to environmental predicates.

Also, unlike the other rule prerequisites, environmental predicates can set parameter values – *out* parameters – based on other parameter values – *in* parameters – in the same environmental predicate. This introduces an information flow channel within the environmental predicate itself. This channel conveys information from the *in* parameters to *out* parameters. Consider for example the environmental predicate $eq(a, b?)$ that sets the value of b to be equal to the value of a. This information flow must be considered when a rule is checked. For example, the rule that was rejected earlier:

$$\begin{gathered} patient^{[\texttt{hospital}]}(x^{[\texttt{hospital,personal}]},\ y^{[\texttt{hospital}]}) \vdash \\ HIV\_testable^{[\texttt{hospital,HIV\_test}]}(x^{[\texttt{hospital,HIV\_test}]}) \end{gathered}$$

could be extended as follows:

$$\begin{gathered} patient^{[\texttt{hospital}]}(x^{[\texttt{hospital,personal}]},\ y^{[\texttt{hospital}]}), eq(x, z?) \vdash \\ HIV\_testable^{[\texttt{hospital,HIV\_test}]}(z^{[\texttt{hospital,HIV\_test}]}) \end{gathered}$$

Without the consideration of information flow within environmental predicates the above rule would be accepted.

We show two ways to handle the information flow within environmental predicates.

The first approach follows a paranoid principle. Environmental predicates can be potentially complex, and have many *in* as well as *out* parameters. It is often difficult to learn how information is flowing in such a predicate, therefore it is assumed that all *out* parameters have been able to benefit from the information given in the *in* parameters. This approach thus considers information flow almost at environmental predicate level. All out parameters are 'tainted', and thus must be treated that way. In the case of such environmental predicates the predicate is valid only if information flow is permitted from every *in* parameter context to every *out* parameter context. Note that this restriction might be insufficient in certain cases. For example, an environmental predicate *equals(x,y)* may check whether its two *in* parameters are the same. Since none of these parameters are *out* parameters, it is possible to create rules that exploit the backtracking capability of policy enforcers, and can thus use this *equals* predicate to propagate the information from x to y.

However, the semantics of environmental predicates, including the actual context information flows from *in* to *out* parameters, might be known to the policy administrator. Therefore, if it is exactly known which *in* parameters are used for determining the value of the *out* parameters, administrators should be allowed to specify information flow between parameters explicitly.

4.4.4 Context specification for rules

The final granularity of context specification is done at the overall rule level. Based on contexts, policy rules can be organised into groups.

Administrators with different levels of trust and jurisdiction, and thus different privileges, can create rules that belong to specific contexts. Similarly to the context restriction for rule components, it is required that all the child components of such rules belong to the contexts of the rule. That is, the rule context is a subset of each child component's context.

This constraint can be used to restrict the policy components that are allowed in a rule. Consider for example the context elements specified in Figure 4.4 on page 68. If a rule is marked with the `OPERA` context element, it will not be able to use components that are tagged with a context that does not contain a context element into which information may flow from `OPERA`. For example, this rule may not contain a component tagged with the `[Security]` context.

This is useful for restricting the domain within which a policy administrator may work, as we can grant him privileges to add and modify rules that have a specific context associated with them. In this way we can limit the rule components that the policy administrator may use for a given rule. For example, we can require that an internal policy administrator for the OASIS research group use only group specific roles and privileges in the group's policy.

4.4.5 Context-based restrictions for data types

In the same manner as for other policy components we have permitted context assignment to data types in order to restrict their use. But, unlike these other policy components, we do not consider data types to be child components within policy components, thus we define here separately the parent-child context relation for data types:

If a data type has no context associated with it we handle it as a general data type, and allow its use in any component specification.

On the other hand, if a data type is assigned a context, then we require that it is used in components that have contexts associated with them into which information may flow from the context of the data type.

Thus, in the case of role, privilege, appointment, function, and environmental predicate specification, information must be allowed to flow from the data type's context to the parameter's context.

The only other place where data types are used in policy specifications is the declaration of rule variables. As rule variables simply provide information conduits, we allow rules to use any data type, since information flow restrictions will be checked at rule component parameter level.

4.5 Hierarchical contexts

We next present an extension to our previous context element specification, in which we add hierarchical information for context elements. The idea was first mentioned in our paper [BEM03] that introduced contexts, but here we elaborate on this concept and provide a more formal description and include examples.

As context elements that are found on directed cycles (see Section 4.1.2) behave in the same way from an information flow perspective, it is vital to avoid accidental creation of such cycles. Unfortunately, it is easy to introduce cycles! One must take special care of the $\star$ sources and destinations, i.e. one must be careful when specifying context elements that accept information from everyone else or permit information flow to potentially every other context element.

It would also be nice to group relevant context elements together. In the example presented on page 62 we talk about two groups of context elements. The first group restricts information flow from the `CompLab` context element to the `OPERA` and `Security` context elements. The other group, which is independent of the first one, contains two context elements, `web` and `secureWeb`, and permits information flow from the `web` to the `secureWeb` context element.

These context element groups were separated in Figure 4.2 on page 62, but in their specification, which puts all the context elements into a set, these two groups would not be distinguished.

In the above example, the specification of wild-cards is also problematic, as it can introduce undesired information flow that could break the parallel context flow specification by joining the context component groups. With the hierarchies introduced in this section it is possible to specify free information flow within a group, i.e. we can specify that from the `CompLab` context element information may flow freely into any other context element in that group, which in our case is {`CompLab, OPERA, Security`}. This context element group can be extended later without the need to modify the information flow restriction for the parent – `CompLab` – context element.

A further problem, which we shall address with hierarchical context elements, is the distributed administration of contexts. In the above example the Computer Laboratory must take care of its `CompLab` context element, but its subgroups are responsible for their own context elements. If additional context elements are introduced they should not affect the context elements and information flow indirectly.

In hierarchical contexts, participating context elements can be grouped together into a new context element, and can be referred to collectively via this new context element. For example, all the Computer Laboratory context elements can be grouped together under the `CompLab` context element. At the university level this `CompLab` context element can be used to specify information flow to the Computer Laboratory, while within the Computer Laboratory the information flow can be further refined.

Therefore context element hierarchies introduce a layering in context specification. This is visualised in Figure 4.6.

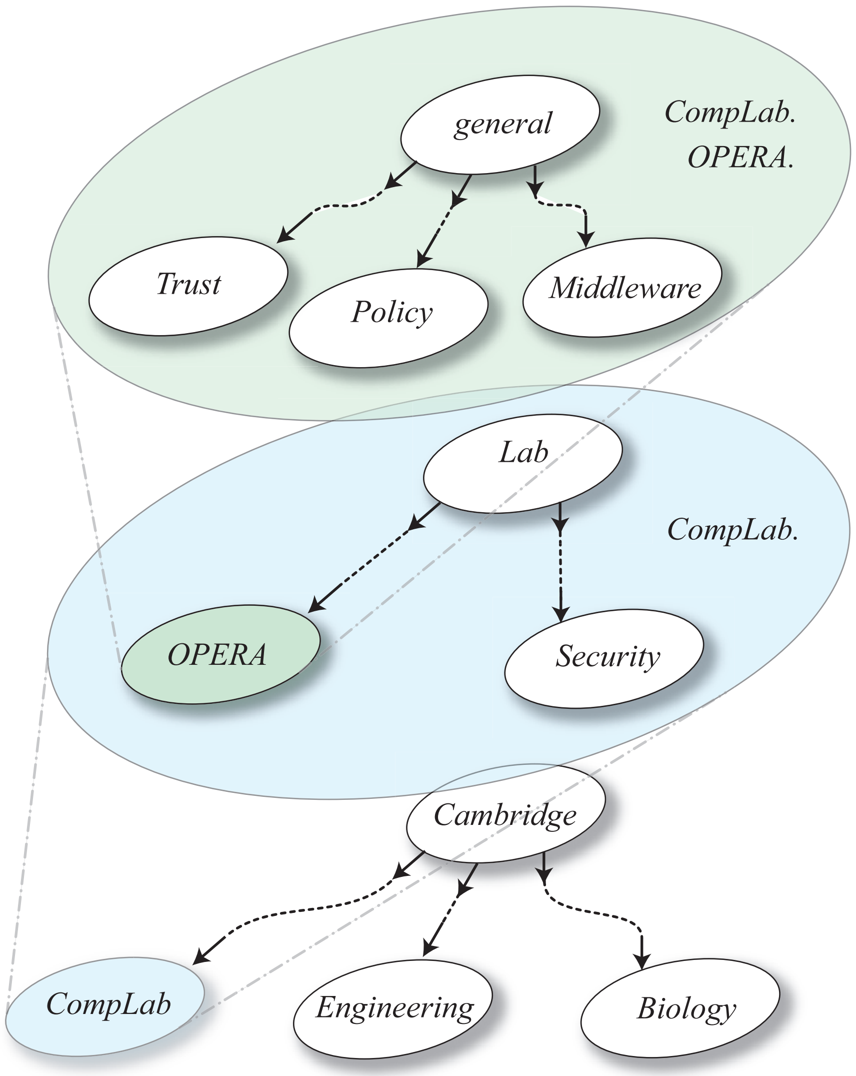


Figure 4.6: Some example information flow layers.

4.5.1 Definition

Building on the definition of non-hierarchical context elements (denoted by $\mathcal{C}'$) we next present a more formal definition of hierarchical context elements.

In the non-hierarchical case we used context elements as unique identifiers, but since we cannot require all context labels to be globally unique (for example a research group at another institution might also be called OPERA), we decouple labels from context elements. We shall use $\mathcal{C}$ to refer to all the hierarchical context elements.

For variables that are from $\mathcal{C}$ we shall use capital letters, e.g. *A, B, C,* or *X, Y*.

Definition 4.3 (Hierarchy relation of context elements) *The hierarchy of the context elements is specified by the $Cparent \subset \mathcal{C} \times \mathcal{C}$ relation. This relation pairs together parent and child context elements.*

We do not allow multiple parents in our context element hierarchy, therefore the Cparent relation must satisfy the following restriction:

$$\forall C, P_1, P_2 \in \mathcal{C}: \ (P_1, C) \in Cparent \wedge (P_2, C) \in Cparent \ \Rightarrow \ P_1 = P_2$$

This restriction says that a context element may have at most one parent.

Context elements cannot be their own parents; this is achieved by the following restriction:

$$Cparent = Cparent \setminus \triangle_{\mathcal{C}}$$

Furthermore, we require the reflexive-transitive closure of Cparent, for which we shall use $\leq$ symbol, to be antisymmetric.

$\leq$ is a partial order – reflexivity and transitivity are consequences of the closure, while antisymmetry was required by the definition of *Cparent*. Thus the context elements, together with the *Cparent* hierarchy relation, form a forest of rooted trees $(\mathcal{C}, Cparent)$.

We also introduce the predicate *Croot* to denote root context elements, i.e. ones that have no parents:

$$Croot : \mathcal{C} \to bool$$

$$\forall A \in \mathcal{C}: \ Croot(A) \ = \ (\neg\exists B \in \mathcal{C}: \ (B, A) \in Cparent)$$

Context elements that satisfy this predicate are the minimal context elements.

4.5.2 Labels

As we decoupled context element labels from context elements, we specify a function (*Clabel*), that returns the label associated with such a hierarchical context element. The range of *Clabel* is $\mathcal{C}'$.

Just as in the case of non-hierarchical context elements, for which we required label uniqueness, we require that labels assigned to root nodes are unique. Similarly, siblings must use different labels. This is expressed by the following constraint:

$$\forall A, B \in \mathcal{C}: \begin{array}{c} A \neq B \wedge ((Croot(A) \wedge Croot(B)) \vee (\exists C \in \mathcal{C} : Cparent(C, A) \wedge Cparent(C, B))) \\ \Downarrow \\ Clabel(A) \neq Clabel(B) \end{array}$$

In the examples it is easier to refer to context elements if we use a path expression. To support this we introduce the *path* function, which returns the labels (separated by '.') on a path from the root of a context element to the context element. Using + as a concatenation the definition is as follows:

$$path(X) = \begin{cases} path(Y) + \text{ `.' } + Clabel(X) & \text{if } \exists Y \in \mathcal{C} : Cparent(Y, X) \\ Clabel(X) & \text{if } Croot(X) \end{cases}$$

Note that as a path from a context element's root to a context element is unambiguous, and because of our labeling restrictions, there is a bijection between the above path strings and the context elements ($\mathcal{C}$). Consequently, the *path* function is invertible, with inverse $path^{-1}$.

This bijection enables policy administrators to refer more easily to hierarchical context elements through a path expression. Accordingly, in our examples we shall use labels (whenever this does not lead to unambiguity) and path expressions.

4.5.3 Context administration

The introduction of hierarchies requires only minor modification to context administration. The main difference is that hierarchical context labels provide additional flexibility for access control management, as they provide a means to associate groups created by contexts. Thus, giving a role rights to modify elements associated with a context will imply the right to modify all the elements in the sub-contexts.

For example, giving someone rights to modify policy components in the `Cambridge.CompLab` context element (or actually to the $path^{-1}$(‘`Cambridge.CompLab`’) context element) will imply the modification rights for all the child components of this context element, thus for the `Cambridge.CompLab.OPERA` and the `Cambridge.CompLab.Security` context elements.

4.5.4 Information flow restrictions

In the case where no wild-cards are used, the information flow specification for our new context elements is the same as for our non-hierarchical ones (except for the decoupling of context elements and labels). But, when wild-cards (e.g. $\star$) are allowed, the hierarchical model differs. The meaning of the phrase *‘information is allowed from everywhere’* can be interpreted now at each context element. Therefore we shall introduce wild-cards on a context element basis. These wild-card symbols will have the form of $subtree(X)$, denoting information source or target for the X context element and its entire subtree.

We shall refer to the set of these parameterised symbols as $\mathcal{C}_\star$.

To get the corresponding context element subtree from a wild-card symbol, we introduce the following function: $expand : \mathcal{C}_\star \to \mathcal{P}(\mathcal{C})$.

$$expand(\text{‘}subtree(X)\text{’}) = \{Y | Y \leq X\}$$

Using the new wild-card context elements we can define information flow restrictions for hierarchical context elements, using functions similar to those introduced for non-hierarchical context elements.

$$\begin{aligned} context\_in &: \mathcal{C} \to \mathcal{P}(\mathcal{C} \cup \mathcal{C}_\star \cup \{\star, \varepsilon\}) \\ context\_out &: \mathcal{C} \to \mathcal{P}(\mathcal{C} \cup \mathcal{C}_\star \cup \{\star\}) \end{aligned}$$

The *context_in* and *context_out* sets must be evaluated depending on the current context elements. We introduce the *Ceval* function to convert sets of context elements, wild-card expressions, and ε to a set of context elements. First we specify *Ceval* for elements of the above set:

$$Ceval : \mathcal{C} \cup \mathcal{C}_\star \cup \{\star, \varepsilon\} \to \mathcal{P}(\mathcal{C})$$

$$Ceval(\omega) = \begin{cases} \omega & \text{if } \omega \in \mathcal{C} \\ expand(\omega) & \text{if } \omega \in \mathcal{C}_\star \\ \mathcal{C} & \text{if } \omega = \star \\ \emptyset & \text{if } \omega = \varepsilon \end{cases}$$

Extending the *Ceval* function to sets of context elements, wild-card expressions, and ε:

$$Ceval : \mathcal{P}(\mathcal{C} \cup \mathcal{C}_\star \cup \{\star, \varepsilon\}) \to \mathcal{P}(\mathcal{C})$$

$$Ceval(\psi) = \bigcup_{\omega \in \psi} Ceval(\omega)$$

The following example shows how the *Ceval* function expands wild cards (for this we use the informal context specification in Figure 4.6) :

$$Ceval\left(\left\{\begin{array}{l}\texttt{CompLab.Security},\\ subtree(\texttt{CompLab.OPERA})\end{array}\right\}\right) = \left\{\begin{array}{l}\texttt{CompLab.Security},\\ \texttt{CompLab.OPERA},\\ \texttt{CompLab.OPERA.general},\\ \texttt{CompLab.OPERA.Trust},\\ \texttt{CompLab.OPERA.Policy},\\ \texttt{CompLab.OPERA.Middleware}\end{array}\right\}$$

The information flow between hierarchical context elements is specified by the $\hookrightarrow$ relation.

Definition 4.4 (Direct information flow between hierarchical context elements)
$\hookrightarrow \subseteq \mathcal{C} \times \mathcal{C}$

$$\begin{aligned} A \hookrightarrow B \quad \Leftrightarrow \quad & (A = B) \vee \\ & \Big(\big(B \in Ceval(context\_out(A))\big) \wedge \big(A \in Ceval(context\_in(B))\big) \Big) \end{aligned}$$

According to the first part of this definition, information flow from a context element to itself is permitted. Also, information is permitted to flow from context element A to context element B if B is included in A's expanded (i.e. wild-card expressions are evaluated and replaced by context elements of a subtree) *context_out* set, and A is included in B's expanded *context_in* set.

Instead of considering the information flow relation as it is, we shall consider its transitive closure. Consequently, we get the following behaviour for parent and child context elements.

Information may flow from a child to the parent if this is either stated explicitly, or the child allows information to flow via a wild-card to its parent and the entire subtree of the parent.

Similarly, information may flow from a parent to a child, if the information flow is either explicitly specified, or information flow is permitted via a wild-card from the parent to itself and all elements of its subtree. These two cases are shown in Figure 4.7.

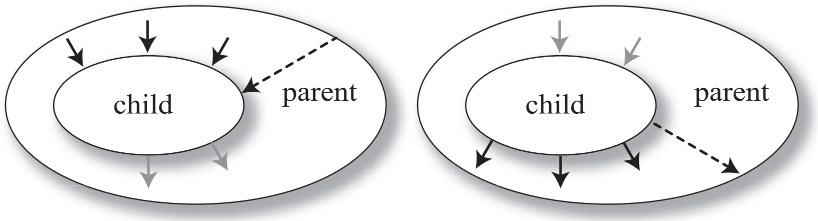


Figure 4.7: Parent-child information flow via wild-cards.

The transitive closure of $\hookrightarrow$, which describes the information flow among context elements, is once again denoted by $\hookrightarrow^*$.

Information flow between contexts is exactly the same as in the non-hierarchical case, only the respective $\hookrightarrow$ functions are used.

Definition 4.5 (Information flow between hierarchical contexts) $\hookrightarrow \subseteq \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C})$

$$\begin{aligned} \alpha \hookrightarrow \beta \quad \Leftrightarrow \quad & (\forall A \in \alpha : \exists B \in \beta : A \hookrightarrow^* B) \wedge \\ & (\forall B \in \beta : (\exists A \in \alpha : A \hookrightarrow^* B) \vee (\varepsilon \in \mathit{context\_in}(B))) \end{aligned}$$

The above definition preserves our lemmas (Lemma 4.1, Lemma 4.2). Lemma 4.3 is preserved as well, the only minor modification that's needed is the replacement of $\mathcal{P}(\mathcal{C}')$ with $\mathcal{P}(\mathcal{C})$ so that it refers to hierarchical contexts.

4.5.5 An example hierarchical context specification

We next present an example, which offers an alternative to part of our previous example on page 62, this time using hierarchical contexts.

Instead of having two top level context elements `web` and `secureWeb` we only have one context element, `WSec`. This has two child context elements with `normal` and `secure` labels. The information flow between these is shown in Figure 4.8. `WSec` groups together the two context elements that we use to annotate policy components that have something to do with web security. The two child context elements (`normal` and `secure`) represent the different classifications of policy components from the web security perspective.

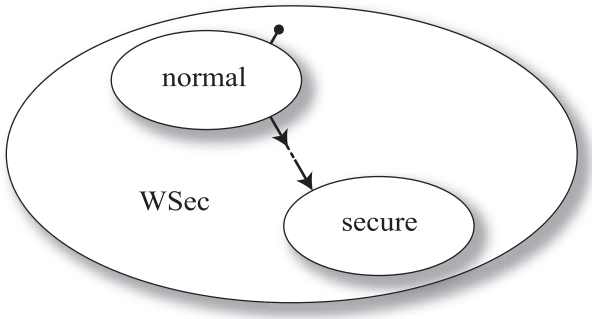


Figure 4.8: Example hierarchical information flow.

The same information flow of our old example is described next using the *context_in* and *context_out* functions:

$$\begin{aligned} \mathit{context\_in}(\texttt{WSec}) &= \{\} \\ \mathit{context\_out}(\texttt{WSec}) &= \{\} \\ \mathit{context\_in}(\texttt{WSec.normal}) &= \{\varepsilon\} \\ \mathit{context\_out}(\texttt{WSec.normal}) &= \{\texttt{WSec.secure}\} \\ \mathit{context\_in}(\texttt{WSec.secure}) &= \{\texttt{WSec.normal}\} \\ \mathit{context\_out}(\texttt{WSec.secure}) &= \{\} \end{aligned}$$

This example shows an advantage of hierarchical context specifications, as unlike in the example on page 62, we keep the web security related context elements together, and separate from other context elements.

Note the new symbol in the figure which indicates that the `WSec.normal` context element is an initial context element.

The parent component does not specify any permitted information flow in or out and it is not even initial! This makes this context element very limited, as information may only flow to itself, i.e. from `WSec` to `WSec`. If a role is marked with this context it must be an initial role, or in activation rules all its prerequisites must have `WSec` in their context.

But, as `WSec.normal` is an initial context it can be included in any target role's context. Every target role that uses this role as a prerequisite will have to have in its context the `WSec.normal` context component, or a context element to which information may flow from `WSec.normal`, which in our case is only `WSec.secure`.

We introduce an additional context element `OPERA` that is defined as follows:

$$\begin{aligned} \mathit{context\_in}(\texttt{OPERA}) &= \{\varepsilon\} \\ \mathit{context\_out}(\texttt{OPERA}) &= \{\} \end{aligned}$$

With this context element we can express the following context information flows:

$$\begin{aligned} [\texttt{OPERA}] &\hookrightarrow [\texttt{OPERA}] \\ [\texttt{WSec}] &\hookrightarrow [\texttt{WSec}] \\ [\texttt{WSec.normal}] &\hookrightarrow [\texttt{WSec.normal}] \\ [\texttt{WSec.normal}] &\hookrightarrow [\texttt{WSec.normal}, \texttt{WSec.secure}] \\ [\texttt{OPERA}, \texttt{WSec.normal}] &\hookrightarrow [\texttt{OPERA}, \texttt{WSec.secure}] \end{aligned}$$

Note that since initial contexts are considered only in the information flow for contexts, as opposed to the information flow for context elements, the transitive closure of the information flow graph does not take account of them. As a consequence, information flow from `[OPERA]` to `[OPERA, WSec.secure]` is not permitted.

This example illustrates well the expressive power of both normal and hierarchical context elements and contexts.

4.6 Further uses for contexts

Until now we talked about two main uses for contexts. The first one was grouping policy components for administrative and access control reasons. We shall discuss this use in more detail in Chapter 7. The second use was to restrict information flow in role activation and authorisation rules. This can be used to differentiate between organisational and functional roles.

4.6.1 Organisational vs. functional roles

As discussed in Section 2.2.3, roles can be classified as organisational or functional roles. In RBAC policies that support such differentiation between roles it is expected that users first enter organisational roles, and then, according to the task they are performing, they enter functional roles. The restriction on such role entry order can easily be specified with the help of contexts. For this, two context labels are required (e.g. `org` and `func`). These labels shall be used to differentiate organisational and functional roles. By specifying information flow only from the `org` context element to `func`, we allow rules that use organisational roles as prerequisites to functional or organisational roles, and we disallow rules that would use functional roles as prerequisites to organisational roles.

Until now we were considering contexts as meta-policies only, i.e. they formed a part of a policy specification that was not considered during policy enforcement time. We next describe three uses of contexts in which they can be used to control or influence the policy enforcer's behaviour.

4.6.2 Auditing

Proper auditing is crucial in systems that support access control. Although in role-based access control roles help to support the principle of least privilege, a user may activate more powerful roles than required, and it must be ensured that these more powerful roles are not abused. Logging the actions a user performs, and letting the user know that such an auditing system is in place, helps to discourage improper access.

In an access control enforcer many decisions are made. Such decisions concern policy components of different sensitivity, and depending on such sensitivity information logging requirements might vary.

There could also be different external factors that must be considered when information is logged. For example, an organisation may have a system of security levels. When the alert level is high, perhaps some hacker activity was detected, and the organisational policy may require every action to be logged. In normal situations, when the alert level is low, the policy may state that only entry to powerful roles must be logged.

Contexts annotate roles, and this annotation is conveyed through the role activation tree via the information flow relation, thus facilitating meaningful annotation of log entries. Information that must be logged thus can depend on the contexts of the roles activated or privileges requested.

The log itself is a resource. Either there are many log files to which information of different sensitivity is written, or some sensitivity annotation is included in the main log. Context elements of the requested privilege or role can be used to segregate information or to mark log entries. This extra information stored in the logs can be used to specify access control information to the logs.

Contexts introduce an abstraction for policy components. Because the expected lifetime of context elements (and other meta-policies) is intended to be greater than that of policy components, contexts are useful for log classification.

4.6.3 Failure behaviour

Another use of contexts is to control the failure behaviour of access control systems. In Section 2.3.2 we described the way one of our OASIS implementations supports certain failure detection by means of heartbeats.

In distributed environments where access control is managed by a service that is itself distributed over the network, mechanisms must be in place to handle network failure. In [BE03] we proposed extensions to OASIS policies to control heartbeat loss.

Heartbeat loss mainly concerns membership conditions. If a membership condition becomes false, the roles activated using such a condition must be revoked. Such revocation can lead to a cascade as a role that is to be revoked itself could have been used as a precondition and marked as a membership condition for other roles, which now must also be revoked. We specify the following revocation behaviours that an OASIS service can choose from in the case of a heartbeat loss.

Time-delayed revocation tagged precondition can hold up to t (t is a parameter) milliseconds after the deadline for the missing heartbeat. If no heartbeat is received during this period the role in question is revoked.

Count-delayed revocation tagged preconditions may hold for up to c (c is a parameter) heartbeat periods after detecting heartbeat loss.

Lazy-revocation will not revoke the role based on loss of heartbeat.

Quick-revocation revokes the role if heartbeat loss is detected (based on the expected heartbeat period for each particular service). This is the existing OASIS strategy.

While the above revocation behaviour can be expressed in policies by using different membership condition symbols (see [BE03]), here we propose to associate such revocation behaviour with the context of roles. This has the advantage that revocation behaviour can be changed without policy modification, and it can also depend on external conditions, such as the organisational security level.

4.6.4 Separation of duty with contexts

Separation of duty constraints (see Section 2.2.2) can be expressed in many ways. Usually conflicting roles are specified with the help of a set. Such a set lists the roles that cannot be activated by the same user. Contexts could also be used for defining conflict groups, i.e. we can indicate conflicting roles by tagging them with a particular context element. Moreover, we can specify conflicting context elements. This could allow separation of duty constraints like: *"A user is not allowed to be active in roles that belong to two conflicting role sets at any time."* We have not explored such use of contexts, but other researchers are currently extending the work presented in this thesis to provide support for context-based dynamic separation of duties and work-flow management.

4.7 Discussion

Until now we have focused primarily on how contexts can be used to supplement OASIS policies. However, contexts are general enough to be used with other role-based access control models and policy specification languages, like the ones we described in Section 2.2.4.

To extend the models of Sandhu and NIST (page 33) is relatively simple, since these models are very similar to OASIS. As in the case of other models that have no support for parameters (e.g. Nyanchama et al. and Giuri et al., see page 35) information flow restrictions can be used primarily for limiting prerequisites, i.e. what roles can be used as preconditions to other roles, and what roles can be assigned to what privileges.

Unlike OASIS, which expresses inheritance via its rules, many systems support role hierarchies. Contexts and information flow between them can limit this inheritance relation by restricting for a role the set of roles it may inherit from. This is achieved by analysing the inheritance graph, and ensuring that information is permitted to flow from less powerful roles to more powerful ones. Such restriction on role hierarchies is very useful, as it can restrict the use of a powerful privilege to a specific class of roles.

Ponder (page 34) can use contexts to mark both authorisation and obligation rules, libraries and objects. For example, Ponder authorisation rules follow a structure that assigns a privilege (target and action) to a subject, under certain conditions. For example:

```
inst auth+ fileServerAccess {
   subject /Employees;
   target  Servers/PrinterServer;
   action   *;                      // wild-card that allows all actions
   when     Time.after(''1300'');   // constraint
}
```

By annotating directory entries such as `/Employees` and `Servers/PrinterServer`, and libraries (`Time`), the use of different policy components can be restricted via the information flow relation. We may require, for example, that information flow must be permitted from the contexts of the *when* and *subject* clauses to the *target*'s context.

Contexts will also be able to restrict the use of certain libraries, for example ones that provide time information, for rule constraints. Thus we can restrict the use of temporal constraints for a certain category of rules. Contexts can also be assigned to different directories in Ponder's directory service, and in this way we can control what targets can be used together with what subjects.

Ponder supports component grouping with the help of hierarchical management domains. Contexts can express these domains, but they also extend management domains with information flow restrictions.

It is relatively hard to apply contexts to the models of Jajodia and Bertino et al. (see page 34). Their models are very rich and their logic-based specification language contains many predicates. The integrity rules of the Flexible Authorisation Framework (FAF) can express the same information flow restrictions for FAF rules as our contexts. But contexts, being meta-policies, can potentially be used for grouping subjects and objects outside a FAF policy specification, and check information flow restrictions of FAF policy rules, in this way restricting the literals and their parameters that can be used for decision rules.

Contexts can also be applied to models that use certificates for roles. In addition to the above we can associate certificate validity times with particular context elements. Thus it could be required that certificates for roles that are marked with a specific context element are issued for at most two hours.

4.7.1 Related work

Information flow control is a well researched area. Denning provides a unifying view of all systems that restrict information flow [Den76]. While in our work we consider only explicit and static information flow, she considers both static and dynamic information flow as well as implicit information flow.

Data and information flow analysis is still a hot topic in security research. Its process is vital in many security assessments [HL03].

Information flow restrictions can be found in many access controls, for example in Mandatory Access Control (see Section 2.1.1). In fact, the security labels in MAC inspired much of our work on contexts.

Bidan and Issarny model information flow as an extension to access control [BI98]. In their access control model, policies specify what a subject is authorised ($s \sim> o$) or prohibited to access ($s \not\sim> o$). This can also be interpreted as allowing information to flow from the subject to the object. Following this reasoning the authors extended their model with inverse information flow ($s <\sim o$). These two information flow directions, specified from the subject's perspective, correspond to the in and out information flows of our context elements.

Nevertheless, there has not been much work in the area of information flow analysis of RBAC access control policies. Many researchers have concentrated on core RBAC models, and focused less on how policies are created and modified.

Information flow in the role graph model

In [Osb02], Osborn analyses her role graph model from an information flow perspective.

As in the role graph model roles are unparameterised and privileges have a form (o, x), where o is an object, and x is an operation (basically read or write operation), it is possible to look at possible information flow among the objects accessed. If, for example, a role may read object a and write object b, then information may possibly flow from a to b.

Osborn's work is basically the reverse of ours, as it produces information flow graphs from role graphs. Such an approach is infeasible in the case of OASIS policies, as roles and privileges are both parameterised, and the assignment of roles to privileges may include environmental predicate evaluation that can set parameters in an unpredictable way.

On the other hand we can assign context to privileges, and thus impose classification of privileges. We can later ensure that such privileges are assigned only to roles that are permitted to access objects of a specific classification. Unfortunately contexts can only express information flow restrictions by static policy analysis, thus parameter values cannot be considered.

PCASSO

The PCASSO (Patient Centered Access to Secure Systems Online) project aims to enable health care providers and patients to retrieve health information over the Internet [Bak00, MBBC02]. To make this information access secure they use security classifications of objects together with RBAC in an access control system for health care.

In a trial application PCASSO supported five security classifications: *low–information* that cannot be identified with a patient; *standard–information* that can be identified with a patient; *public–deniable-information* about conditions, such as HIV/AIDS, abortion, adoption, mental health, genetics, substance abuse, and sexually transmitted diseases, that by law requires special protection; *guardian deniable–information*, such as that about teenage abortion, that can be denied to a guardian; and *patient deniable–information* that, if disclosed to the patient, might harm his or her well-being.

Roles are associated with capabilities (in OASIS these are called privileges) that specify what data the role may access. PCASSO works with an unusually small set of roles (in the above example there are only four roles: primary care provider, secondary care provider, emergency care provider, and patient).

The information flow restrictions that PCASSO enforces can be expressed with the help of our contexts. We can annotate roles and privileges with the above security classifications, and ensure that only privileges that comply with the security policy are associated with roles. In fact, our approach can specify classes of roles and privileges, thus we can have a number of roles belonging to a context or security classification. The information flow restrictions defined for the contexts will be maintained in the authorisation rules. In this way the above information flow restriction can be extracted from the policy. This helps to apply this information flow restriction in environments that have existing, legacy policies.

4.7.2 Summary

In this chapter we presented a kind of meta-policy we call contexts. We did this by first introducing a basic context model, and then extended this to support hierarchical relations between context elements. Later we showed how these contexts can be used to group policy components and also to specify restrictions on OASIS RBAC policies. We used a relation on contexts, called the information flow relation, which can check information flow between parameter values within a rule, and between OASIS and the external world. With this relation we can also express constraints over policies, and thus assist policy evolution.

We provided examples to show how expressive these restrictions are and to indicate the motivation for our research.

We also showed additional uses for contexts when context information can be accessed during policy enforcement time. These uses included supplementing and refining audit information and failure behaviour for distributed policy enforcers.

Finally, we briefly described the applicability of our context model to other RBAC architectures and implementations.

5 Compliance policy

In the previous chapter we looked at the two main uses of contexts; namely, to group policy components and to impose restrictions on rules. This we have done by associating policy components with context labels.

In this chapter we explore a type of meta-policy that is more expressive than contexts, as in addition to further restrictions it will allow us to specify requirements. These meta-policies we call compliance policies. They contain an abstract policy component model, which extends the policy components of policies.

Compliance policies specify requirements on this abstract policy model, thus making themselves independent from policies. A complete compliance policy includes the specification of its abstract components and a list of constraints over these components. In order to use this meta-policy a connection between a policy and this meta-policy must be provided; we shall refer to this connection as a *mapping*. Based on this mapping we can evaluate policies against the requirements specified on the compliance policy's abstract model.

We expect policy administrators to use compliance policies to extract specific aspects of policies and separate them from the policy itself. For example, a simple compliance policy could encompass a static separation of duty constraint or require that a particular role should exist. These qualities can be checked at policy specification time, so there is no need to include them in policies that should govern only policy enforcement decisions. These design goals are encapsulated in compliance policies, and when a policy changes, compliance policies can be used to check the new policy version, and in this way ensure that the original design goals are still satisfied by the new policy.

As in the case of every meta-policy, we expect that the lifetime of compliance policies is longer than the lifetime of individual policies. However, meta-policies also evolve. We support the maintenance of compliance policies in the following two ways.

First of all, we allow the coexistence of many compliance policies. All these compliance policies define an abstract model through which they capture different aspects of a security policy. A policy can then be checked against all of these abstract policies. Consequently, if only a specific aspect of a long-term policy goal changes, only the appropriate compliance policies need to be updated.

Second, as compliance policies are themselves similar to policies, we allow hierarchical refinement of compliance policies. Thus we can distinguish goal specification at different levels of abstraction.

Compliance policies can also form the means to provide a user with information about a policy without disclosing the low-level policy specification. As the lifetime of a compliance policy potentially spans the lifetime of many policies, providing information through these meta policies not only reduces the complexity of the information given to users, but also the frequency with which users are given substantially new information.

For example, a compliance policy may specify that a student role is authorised to use a printer via a virtual *can_print* privilege. If policy administrators check the effective policy against this meta-policy, it is sufficient to show the users only this meta-policy together with its compliance. The way this role-privilege binding is accomplished in the effective policy, whether there are some intermediary roles activated, or some environmental predicates evaluated, is completely irrelevant to the users, as long as they know that they can access a printer.

Similarly, compliance policies can help to give information to other policy administrators, and this quality of compliance policies is vital in environments with many policy administrators. Furthermore, compliance policies can be used to restrict the freedom of policy administrators, and provide them with a sandbox environment where they may do everything as long as the resulting policy complies with the relevant meta-policies.

Another justification for the use of a separate component model, as opposed to the component model of a specific policy, is the differences among policies. Policies are often designed in a bottom-up way, by first specifying the policy and then thinking about its properties. A reason for such policy design comes from the need to support legacy applications that might require specific privilege formats. By using a separate component model we facilitate the reuse of compliance policies. This will later lead to further benefits, such as inter-policy cooperation, which we shall describe in a later chapter (Chapter 6).

We next build up the structure of compliance policies; starting from simple compliance policies, we gradually extend their abstract model to include more complex features, such as partial mappings, implicit rules, and negation.

5.1 The simplest compliance policy

The simple compliance policy structure we start with is almost the same as a policy that uses only unparameterised components. We use this to show how compliance policies are expected to work. This is also illustrated in Figure 5.1, which illustrates the three steps for compliance policy checking: (1) specification of a compliance policy, (2) mapping this compliance policy to a policy, and (3) performing the compliance check.

The components with which we shall gradually extend our basic model are also shown in this figure, but these are grayed out. As they are introduced we shall also refine the above three steps.

5.1.1 Structure

Our simple compliance policy model consists only of the specification of roles, predicates, appointments (note that from the point of view of prerequisites, environmental predicates and appointments are very similar to roles), privileges, and both authorisation and activation rules. We do not yet allow these components to contain parameters. We also do not associate meaning with the components of the compliance policy; they behave simply like a set of labels.

Consider the following example compliance policy, which specifies a role, a privilege and an authorisation rule:

$$\begin{array}{l} general\_student\_role \\ general\_db\_access \\ general\_rule1 : general\_student\_role \vdash general\_db\_access \end{array}$$

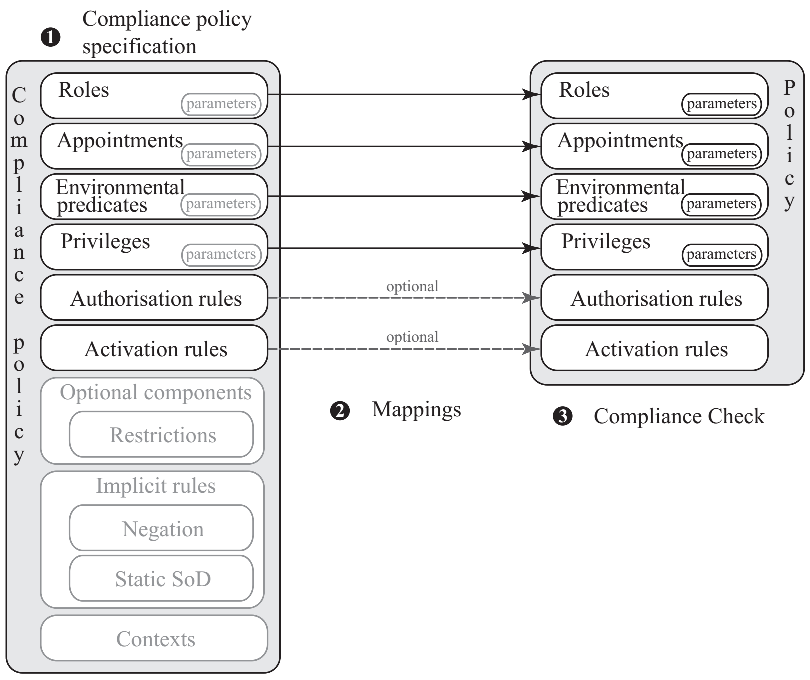


Figure 5.1: The structure and usage of simple compliance policies.

This compliance policy expresses that there shall be an unparameterised role *general_student_role*, an unparameterised privilege *general_db_access*, and an authorisation rule that unconditionally assigns this privilege to the role.

This compliance policy is independent from any policies. It only specifies an RBAC policy on an abstract policy model.

Since a compliance policy very much resembles a policy, we require that compliance policy components satisfy the same constraints as policies, i.e. they are well-formed. One such constraint is that rules may not contain free variables. We assume that the following compliance policy rules are well-formed.

5.1.2 Mapping simple compliance policies to policies

Once a compliance policy is specified it can be used for checking real policies. As it is independent from policies it can be used to check any number of policies, but also a policy may be checked against any number of compliance policies.

To continue with our previous example we first provide a policy specification:

$$\begin{array}{l} student\_role \\ lecturer\_role \\ db\_access \\ rule1 : student\_role \vdash db\_access \end{array}$$

The next step is to specify the connection between the policy components and the compliance policy components. This is done with the help of mappings, which specify the corresponding components of the compliance policy and the checked policy.

In our example this mapping is as follows:

compliance policy	maps to **(map())**	**policy**
general_student_role	$\sim$	*student_role*
general_db_access	$\sim$	*db_access*
general_rule1	$\sim$	*rule1*

In our simple model we use $\sim$ mapping, with which we indicate equivalence. Later in this chapter we shall see more complicated mapping types. Note that the rule mapping is optional; if it is provided it can simplify the compliance check, but if it is not provided the compliance check can still be performed. In general we shall use the function name *map* to refer to the mapping between the compliance policy and the policy components. We shall use this function to get the components corresponding to an abstract component, for example, according to the above mapping, $map(general\_student\_role) = student\_role$.

5.1.3 Compliance

Once a mapping between the components of a policy and a compliance policy is provided we can check whether the requirements and restrictions specified in the compliance policy are satisfied by the policy. This consists of the following steps:

1. The first step checks whether the mapping is complete, i.e., whether compliance policy components are mapped onto an appropriate policy component. We require the mapping of roles, appointments, environmental predicates and privileges, but we also accept optional mappings for rules.

2. The second check concerns the rules of the compliance policy. It checks whether the policy contains rules that are equivalent to the translation of compliance policy rules. We say that two rules are equivalent if they contain the same prerequisites and targets, and bind the variables in the same way, i.e. the only difference is in the order of prerequisites, the names of the variables and the names of the rules.

   The rule translation is done with the help of the mappings.

In our example, the first step checks for the existence and correctness of the three compliance policy component mappings. The second step translates the compliance policy rule into the world of the policy:

$$translated\_general\_rule1 : map(general\_student\_role) \vdash map(general\_db\_access)$$

After applying the map functions (when it is possible) we get the following rule:

$$translated\_general\_rule1 : student\_role \vdash db\_access$$

As the policy contains a rule that is equivalent to this translated rule (we even had a hint in form of a rule mapping), and there are no other rules to check, the compliance check passes.

A successful compliance check will result in a certificate that contains the version of the policy that is checked, the version of the compliance policy, the mapping between the two, and the time of the compliance check.

It is the responsibility of the policy administrators to provide a binding between the components of a compliance policy and the components of a policy. But is it possible to ensure the correctness of this mapping? We expect compliance policies to help administrators to maintain subsequent policy versions. In such cases compliance policies are used to describe specific aspects of a policy and can even be used as templates to specify policies. The correctness of the mappings is up to the policy administrators. But, if this compliance is used as a source of information to the external world, let us say to the users, then a trusted third party is required to check the correctness of the mappings between a policy and a compliance policy. This third party – trusted by both the administrators and the users – can digitally sign the mapping and compliance certificate. Users must be able to check whether the policy is the same as the one that is used (in our implementation, DESERT, we provide a function that returns a hash value of a serialised policy).

5.2 Parameters

In our simple model we considered unparameterised rule components only. While this compliance policy can express requirements for RBAC models that support no parameters, in OASIS policies, apart from the most trivial ones, rule components do have parameters. Unfortunately parameters complicate compliance policies, as, even if both component models (those of the policy and the compliance policy) use the same data types, the values of these could be interpreted differently. For example, a privilege that contains a temperature parameter could express this parameter value in either Celsius or Fahrenheit. These type of problems, referred to as *semantic heterogeneity*, are well researched in other research fields, such as federated databases. We next consider the analogy between our compliance policies and federated databases.

5.2.1 Federated databases

A federated database [HM85, HM93] is a collection of cooperating database systems. Each of the constituent databases is autonomous, i.e. it has its own set of schemata that are maintained locally. The relation between compliance policies and policies is very similar to the relation between two cooperating databases; we can associate the data type model of a compliance policy and the data type model of a policy with two component databases in a federation. Later we shall see that this similarity holds even from the perspectives of autonomy and administration.

From a user's viewpoint, the entire federated database should look like a single, consistent system. Consequently, users should be able to query and modify data, which is most likely stored across a number of databases, parts of which are possibly stored using different data representations.

As data is stored in different forms, some databases contain more detailed information. Generally, federated databases distinguish local and global accesses. Local accesses allow the use of local functions, and give access to all the details, while global ones support only a subset of functionalities shared in the federation.

In every local relational database data is organised with the help of schemata, which are named sequences of field declarations (a name and data type pair). Thus, depending on the federated database architecture, the global functionalities can be expressed either with the help of a global schema or through pairwise contracts between cooperating databases.

In [SL90], Seth and Larson describe a reference architecture for a federated database. Among the six major components in this architecture (data, database, command, processors, schemata, and mappings) are mappings, which are functions that correlate schema objects of two different schemata.

Our parameterised policy rule components are similar to schemata. Both associate names with components, and both contain a list of typed parameters or fields. For example, the role specification

$$treating\_doctor(doctor : doctor\_id,\ patient : patient\_id)$$

could as well be interpreted as a relational schema to store the relation between doctors and their patients.

Similarly, the mappings between global and local schemata, or between two local schemata, closely resemble our mappings between compliance policy and policy components. Naturally, we face similar problems. We shall discuss next one of the biggest of these problems, semantic heterogeneity, which is associated with disagreements about the meaning of data.

5.2.2 Semantic heterogeneity

Semantic heterogeneity is used to describe the phenomenon of disagreement about the meaning or interpretation of the same or related data. For example, as we have already mentioned, a temperature value could be given either in Fahrenheit or Celsius.

Naturally such disagreement can occur at many levels. In certain cases it is difficult, if not impossible, to provide an algorithmic resolution. An example is presented in [LSPR93] by Lim et al., according to which two database entities, stored in different databases, could refer to the same real-world entity, but, this remains undetected due to the structural differences in their representations.

Semantic heterogeneity that cannot be resolved is generally referred to as *fundamental semantic heterogeneity* [Col97]. In the case of compliance policies such fundamental structural heterogeneity can arise from the rich set of policy specification tools.

There could be many sources for semantic heterogeneity, such as differences in data definition constructs, differences in object representations, and system-level differences in the way atomic data is stored.

At the level of data semantics, Ceri and Widom describe four kinds of semantic heterogeneity [CW93]:

naming conflicts occur when different databases use different names to identify the same real-world entities. For example, in one database schema the identifier of a doctor could be stored in a field named *doctor_id*, and in an other database schema the same field could be referred to as *doctor*. We face the same problem in the case of rule component parameters, for example, parameter names of an abstract doctor role could be different from the parameter names of a policy role, even though they represent the same entity. Similarly the name of an abstract role can be different from the one used in a policy. We do not seek automatic resolution of such problems. Access control policies are a critical application domain, therefore computers should not be allowed to infer mappings automatically.

domain conflicts occur when different databases use different values to represent the same concepts. For example, a doctor might be identified in a central NHS database as 'ab374', but in a hospital he might be known as 'András'. In the case of compliance policies the abstract model's parameters and the parameters of the policy being checked could have similar differences. Our example about the temperature representation, in which either Fahrenheit or Celsius is used, falls into this heterogeneity category.

meta-data conflicts occur when the same concepts are represented at different levels, in one database at the schema, in the second database at the instance-level. For example, historical information about a person can be stored either as part of a table that stores people's information in a tuple, or as a separate table for every person.

structural conflicts occur when different data organisations are used, for example a name is represented as a single character string, or represented with two strings, one for the surname, and one for all the other names. Our compliance policies face this problem when they are applied to existing policies; however, our compliance policies can be used as templates for newly created policies, thus helping to synchronise the policy structures of independent policy domains.

This kind of semantic heterogeneity also includes *type conflicts*, when the same concept is represented by different data types. For example, a temperature can be represented by either a float point or an integer value.

One way of handling semantic heterogeneity is to use semantic values [SSR94]. In this approach, which is based on strong typing, the semantics of the data is determined only by the type it is an instance of. But even if such systems exist, we must provide conversion functions or mappings, that change data from one representation to another.

Automatic mapping of schemata is an ambitious goal (for such projects see [Hul97]), and there are not many implementations that support it. As in the case of most federated databases, we require policy administrators to provide the mappings between the components of the compliance policy and those of the policy. We require this manual control partly because of the higher sensitivity of our application domain.

5.2.3 Mapping types (partial mapping)

We next consider mappings of data types that can be specified in the abstract model of compliance policies. There are two problems we face with these mappings [Ken91]. The first problem arises when the structure of the mapped components is different, for example, a single data type value is mapped to a pair of data type values. For instance, a name parameter could be used in an abstract role, which can be mapped to two parameters of a policy role (first names and surname). The second kind of problem we can face is when the interpretation of the values is different in the compliance policy and the policy, and a lossless mapping is not available. For example, it is not possible to convert a value that contains an address (e.g. 459 King's College, Cambridge, CB2 1ST) to a city (Cambridge) without some information loss, thus we shall not be able to convert this value back to its original form.

As a consequence we must consider conversion functions that might take more than one argument.

We require that functions be total functions, i.e. they must be defined for every element in the function's domain.

In ideal cases we can provide invertible one-to-one mappings, i.e. ones that form a bijection between their domains and ranges. Unfortunately this is not always the case, and some mapping functions will have no inverses.

Thus, for our compliance mappings, we distinguish bijections and lossy mapping functions. An example for the first kind of function is one that converts from metres to feet. An example for the second is the one that converts addresses to cities.

Mapping of parameterised components

With the help of these functions we can map compliance policy components and their parameters to policy components. The compliance policy must specify whether it accepts lossy mappings, or only bijections, for a specific abstract component parameter.

Restrictions Such augmentation of policy component is done with the help of *restrictions*, which are annotations added to compliance policy components. We shall list restrictions for abstract components in braces ('{}'). If a component has more than one restriction, then these will be separated with a comma. For example, *roleA{restriction1, restriction2}()*. The list of all restrictions is presented in Appendix A.

Because rule component parameters can share data types, i.e. general policy specifications do not make use of semantic values, we require parameter mappings on a rule component basis, i.e. for every abstract rule component such as a role, the mapping of parameters must be specified together with the mapping of abstract roles.

For example, consider the following simple compliance policy that contains parameterised components. For readability we prefix the compliance policy components with 'c_'.

$$\begin{aligned}&c\_student\_role(c\_id : c\_idtype)\\&c\_db\_access(c\_id : c\_idtype)\\&c\_rule1 : c\_student\_role(x) \vdash c\_db\_access(x)\end{aligned}$$

This compliance policy specifies an abstract student role with an identifier parameter, and a privilege with a student identifier parameter. This policy expresses that a student role must be able to acquire the *c_db_access* privilege, which we assume will give him access to his own data.

In our example policy, students are identified by two parameters: their department and name.

Our policy thus looks as follows:

$$\begin{aligned}&student\_role(dep : string, name : string)\\&db\_access(dep : string, name : string)\\&rule1 : student\_role(x, y) \vdash db\_access(x, y)\end{aligned}$$

The mappings, which in our example we require to be bijections, are as follows:

compliance policy	maps to	**policy**
c_student_role(x)	$\sim$	*student_role*$(f_1(x),\ f_2(x))$
c_db_access(x)	$\sim$	*db_access*$(f_1(x),\ f_2(x))$
c_rule1	$\sim$	*rule1*

This mapping requires two function symbols (f_1 and f_2) that map the compliance policy component parameters. Note that in our mappings we require semantics to be attached to these functions only if there are constants present in the compliance policy, otherwise functions can behave like labels. The same applies to data types, which in the absence of constants behave only like labels. In translated rules we can basically replace the $f_1(x)$ expression with new variable names. Consequently, the compliance test translates the abstract rule ($c\_rule1 : c\_student\_role(x) \vdash c\_db\_access(x)$) to $student\_role(f_1(x), f_2(x)) \vdash db\_access(f_1(x), f_2(x))$, it replaces these function labels with new variable labels ($student\_role(z, w) \vdash db\_access(z, w)$), and checks whether this rule is the same as the *rule1* rule.

In this section we still require that every abstract parameter is mapped onto a policy parameter, and every policy parameter is also mapped onto. Our current mappings thus support only splitting and merging abstract parameters, which also includes changes to parameter order. Since this is often insufficient, we shall provide more general mappings.

Mapping functions may use databases to convert values. The reason for this is that value sets are often not available at the time of policy specification. In such cases the domain and range of the mapping functions are determined by populations. For example, the identifiers of doctors in a hospital are usually provided in a database.

Mapping direction If we used only bijections for parameter mappings, then we could convert parameter values both from and to the compliance policy and policy value domains. If, in addition to this, every abstract rule component is mapped to a different rule component in the policy, then the compliance policy and the policy behave isomorphically. That is, were the inverses of the mappings known, we could swap at any time between one policy and the other one by translating the components to the other's representation.

For an OASIS role activation rule this is illustrated next:

$$\begin{array}{ccccccccccccc} c\_r_1, & c\_r_2, & \dots & c\_r_{n_{c\_r}}, & c\_ac_1, & \dots & c\_ac_{n_{c\_ac}}, & c\_e_1, & \dots, & c\_e_{n_{c\_e}}, & \vdash & c\_r \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & & \updownarrow \\ r_1, & r_2, & \dots, & r_{n_r}, & ac_1, & \dots, & ac_{n_{ac}}, & e_1, & \dots, & e_{n_e} & \vdash & r \end{array}$$

Such situations could occur if the compliance policy has been used as a template to create a policy.

If the compliance policy were specified on a real policy, jumping from one representation to the other would be especially meaningful for environmental predicates, as it would be enough to evaluate one of the environmental predicates (either the one of the policy, or the corresponding one of the compliance policy). Similarly, appointment certificates can be converted this way from one representation to another. We shall exploit such mappings in Chapter 6, in which we shall use mappings to reconcile differences between different policies.

Uniqueness of mappings However, we cannot expect that every abstract policy component can be mapped onto a distinct policy component. For example, two abstract roles might be mapped onto a single role. We must specify for each compliance rule component whether we require it to be mapped to a policy component that is not yet mapped onto. We do this with the *unique* restriction keyword.

Component consumption With unique mappings the components of a policy are consumed if they are mapped onto. As a consequence, the order in which they are mapped on does matter, as different orders could lead to different consumptions. To avoid ambiguity we allow ordering of the compliance policy components (in our implementation they are represented in an XML file, which is ordered). Rule mapping hints also help to resolve such ambiguity. Such hints are particularly important since we do not support backtracking.

Split mappings There might be two policy roles that correspond together to an abstract role. To allow such mappings of rule components we use the *split* restriction keyword.

If such an abstract role occurs as the target of an abstract rule, then this rule is translated into two (or more) policy rules, each of which must be satisfied.

If such an abstract role occurs as a prerequisite, then any non-empty subset of it can be present in the translated rule, i.e. the abstract rule is translated into a set of rules, out of which at least one must be equivalent to a policy rule.

5.3 Minimum and maximum specifications

Until now we have been considering compliance policies and mappings that prescribe exactly what a policy must contain. For example, we specified what roles we want, and exactly what prerequisites a rule must have. On the other hand, we have seen that in certain cases we allow minor differences between the compliance policy and the policy. In the case of rule components we permitted different parameter signatures (different parameter numbers and parameter orders) while still satisfying the requirement that we map every abstract component parameter to a policy parameter, and every policy component parameter is mapped onto.

We extend this flexibility by allowing compliance components to specify both a minimum and a maximum set of components that are expected to be in a policy. Our current compliance policies and the restrictions on mappings can be considered as a minimum set of requirements, as every compliance policy component must map onto a policy component, thus they must be part of a policy.

5.3.1 Optional policy components

By introducing optional elements in the compliance policy we make a first step towards specifying the maximum requirements for a policy. We can specify a set of elements a policy may have, and if a policy has these elements, then we can prescribe how these components should behave.

We next consider the different components of a compliance policy where we allow optionality.

Optional parameters

We have seen that rule component parameters of a compliance policy and of a policy can differ in terms of the order of the parameters, and even in the number of parameters. But because of the restriction that every parameter of an abstract rule component must be mapped onto a parameter of a policy component, and that every parameter of policy component must be mapped onto, we could not specify optional parameters. Note that there is an indirect way to introduce an optional parameter (by adding the optional parameter to any of the mapping function arguments), but this should be avoided as it only obscures the mapping specification.

We therefore introduce syntax to specify that a rule component is optional, in which case the corresponding component in the policy is not required to have it either directly or indirectly (by merging it with other parameters).

Consider, for example, the next compliance policy specification:

$$\begin{aligned}
&c\_student\_role(c\_id : c\_idtype, [subject : string])\\
&c\_db\_access(c\_id : c\_idtype)\\
&c\_rule1 : c\_student\_role(x, y) \vdash c\_db\_access(x)
\end{aligned}$$

In this example the *c_student* has two parameters, but one of these parameters (*subject*) is an optional one. Thus, this role can be mapped onto a policy role such as *student(id:idtype)*.

Indirectly these optional parameters are used in rules, thus their values can be propagated to other rule component parameters. Clearly, if a parameter does not exist in a policy it cannot be used, therefore we must impose the following constraint on compliance rules:

If a rule variable is assigned to an optional parameter we distinguish two cases:

1. There is a non-optional prerequisite parameter that is assigned to the same variable, i.e. during policy evaluation the value of this variable can be determined. In such cases there is no need for further restrictions.

2. If there is no non-optional prerequisite parameter to which this variable is assigned, then every parameter that serves as an input parameter to the rule, and to which the variable is assigned, must also be optional. Furthermore, it must be ensured in the compliance mappings that if an optional parameter is absent from a policy specification, then the parameters that were dependent on it (via any compliance policy rule) are not present in the policy in other rule components.

For example, if the *c_db_access* privilege had a string parameter, and the compliance rule had the form

$$c\_rule1 : c\_student\_role(x, y) \vdash c\_db\_access(x, y),$$

then the specification of the *db_access* privilege must mark its second parameter as optional. Also, when the *c_student_role* is mapped to a policy role that does not contain the *subject* parameter, then the equivalent of the *c_db_access* privilege must not have a second parameter either.

In what cases can a policy component have different parameters from the compliance policy parameters? First of all, the compliance policy could be authored by a different administrator or set of administrators than the administrator of the policy. This could lead to differences, especially as a local policy might have to include some extra parameters to support legacy applications. Such bottom-up policy designs are very common. Using optional components, compliance policy authors can give more freedom and flexibility to policy administrators.

In Appendix A we provide a table that shows the restrictions which can be used with various compliance policy components.

Optional rule component

Like optional parameters, we allow the optionality of rule components. We thus allow the specification of abstract rules that can include prerequisites that may be absent from policies. Rule heads, such as target role or privilege, could also be made optional, but as every rule must have one head, these cases are equivalent to rule optionality, which is discussed after this section.

For example, in the following compliance policy we introduce a rule that can have an optional prerequisite role (*c_OPERA*):

$$\begin{array}{l} c\_student\_role(c\_id : c\_idtype) \\ c\_OPERA(c\_id : c\_idtype) \\ c\_db\_access(c\_id : c\_idtype) \\ c\_rule1 : c\_student\_role(x, y), \boldsymbol{[c\_OPERA(x)]} \vdash c\_db\_access(x) \end{array}$$

Similarly to the case of optional parameters, we must be careful about the parameters of optional prerequisites. We can thus ensure that every parameter can be assigned a value during rule evaluation.

Optional component specifications

The specification of abstract data types, roles, appointments, environmental predicates, privileges, and even rules can be marked optional, in which case a policy may not provide mappings for these components. Obviously, in such cases the compliance policy must handle components that rely on optional components and whose equivalents are not present in a policy. If a data type is optional then it can only be used in abstract policy components that are themselves optional (or require the existence of this data type as a precondition, see following sections). Similarly if a rule component specification is optional, the rules using this component must be themselves optional, or the prerequisite containers that hold this component must be optional within the rules.

It is relatively simple to specify optional rules, as there are no basic policy components that depend on a rule. By marking a rule optional we indicate that there need not be a policy rule to which the role is mapped.

5.3.2 Restrictions

In the example presented in Section 5.3.1 we have seen that if an abstract rule component is marked as optional, it must also be specified as an optional prerequisite in every rule that uses this component. But, in this case there is no way to enforce the use of this prerequisite in a rule if the component has a corresponding policy component (i.e. it is mapped onto a policy component). Similarly, while in certain scenarios, like environments where a web service that provides an environmental predicate is unavailable, it is acceptable to have optional prerequisites, it can be expected that in such cases alternatives are used. To support this, we introduce restrictions to abstract rule components.

Restrictions can contain conditions that are evaluated during compliance checking. These conditions can check whether specific mappings have been provided, i.e. whether some optional abstract policy components have a corresponding policy component. Abstract policy components whose restriction is not satisfied will not be considered in the compliance check.

The restrictions we allow now are:

isdefined() is a restriction predicate; it checks whether a specific optional compliance policy component (specified as the predicate's parameter) is mapped onto a policy component or not, i.e. whether this component exists for a specific policy.

isnotdefined() predicate is the opposite of the *isdefined()* predicate; it specifies a condition when a specific abstract policy component (the predicate's parameter) is not defined in a policy.

These restrictions can be used anywhere in the compliance policy specification, and they may refer to any previously defined optional compliance policy component. Note, however, that we do not allow circular dependencies.

We now extend the example presented on page 95 to include restrictions.

$c\_student\_role(c\_id : c\_idtype)$
$c\_OPERA(c\_id : c\_idtype)$
$c\_db\_access(c\_id : c\_idtype)$
$c\_rule1 : c\_student\_role(x, y), c\_OPERA\boldsymbol{\{isdefined(c\_OPERA)\}}(x) \vdash c\_db\_access(x)$

We have removed the optionality from the *c_OPERA* prerequisite, and specified a restriction (*isdefined(c_OPERA)*) for this component. If the role *c_OPERA* is not defined in a policy, this rule will behave as if the prerequisite were not there, but if this role is defined, then this prerequisite role will be part of the rule, unlike in the original example, where it was only optional.

Abstract policy components that are marked with the *isdefined* or *isnotdefined* restrictions are considered the same way as optional components, so rules containing such components must be checked similarly to ones that contain optional components.

By assigning such restrictions to rules we can provide alternative rules in scenarios where certain abstract components are not mapped. For example, the following provides two rules of which only one can be checked in a policy. The one that is selected depends on whether the policy contains a role that corresponds to the *c_OPERA* role.

$$rule1\_a\{isdefined(c\_OPERA)\} : ...$$
$$rule1\_b\{isnotdefined(c\_OPERA)\} : ...$$

A restriction can contain any number of such condition predicates, e.g. such a restriction might look like {*isdefined(c_OPERA), isnotdefined(c_student_role)*}. The parameter of a component can be referenced by using a '.' notation, for example, the parameter of the *c_OPERA* abstract component is referred to as *c_OPERA.c_id*.

The *isdefined* and *isnotdefined* predicates are evaluated according to a partial order defined by the dependency relation of optional components and these predicates. For example, if a component contains a restriction which refers to another optional component, then it is not evaluated until an administrator specifies a mapping for that component or explicitly states that the referenced optional component will not be mapped.

Administrators must specify explicitly whether a particular optional component will not be available in a policy. This can be reflected in GUIs, which, in order to simplify mapping specifications, can hide (or display differently) unavailable optional components.

5.3.3 Arbitrary components

In the previous sections we concentrated on how to specify compliance policy components that are optional in a policy, i.e. we were looking at cases in which the compliance policy describes a superset of policy components. But, naturally, a policy may contain components that are not described in a compliance policy, especially as compliance policies are designed to describe a specific aspect of a policy. To support the specification of compliance policies that allow such additional components in a policy we introduce *arbitrary components*, which work as a wild-card for policy components.

We shall denote these arbitrary compliance components with the component type (e.g. role) with an underscore prefix. For example, *_role* symbolises an arbitrary role. If it is present in a compliance policy specification, then the policy whose compliance is checked may contain a single role specification that is not specified elsewhere in the compliance policy. Clearly adding a single such component is of little help; also, saying that a policy may have an arbitrary component does not help much to specify restrictions over it. We therefore extend the restrictions introduced in the previous section; an arbitrary component together with such a restriction forms a template against which the components of a policy can be checked.

All arbitrary components may contain *cardinality restrictions*, which can specify a minimum and/or a maximum number of occurrences for any abstract component template. The two restrictions are **min()** and **max()**. They take one parameter, which is either a number or the "unbounded" keyword. The default value for these two restrictions is one. By specifying these restrictions for an abstract arbitrary component template we require that an evaluated policy matches this template the number of times that is within the min/max range. For example, by adding the *_role*{*min(1), max('unbounded')*} component to the compliance policy we specify that a policy must have at least one role that is not specified otherwise in the compliance policy, but, as the upper limit is unbounded, the policy may have any additional such roles.

For arbitrary component parameters (denoted by *_parameter*) we permit the following restrictions:

datatype may specify a restriction on a parameter's data type. For example, *datatype('integer')*. The parameter data type must refer to a data type in the compliance policy specification.

Cardinality constraints can specify parameter multiplicity within a parameterised component. For example, the arbitrary component *_role(_parameter*{*min(0), max(3), datatype('integer')*}*)* specifies a role that can have up to three parameters whose data type is integer.

For abstract rule component (role, appointment, privilege, or environmental predicate) specifications we allow, as for almost every other abstract component, cardinality restrictions. These cardinality restrictions specify constraints on the number of rule component specifications in the target policy. Cardinality restrictions for these components also imply the *unique* restriction. Appointments and environmental predicates may have an additional *binding* restriction, which can impose limitations on their binding attribute.

Within rules we can specify restrictions to prerequisite containers and to the target container as well. Cardinalities for these containers specify the multiplicity of the container within the rule. For activation containers, i.e. for the prerequisites of activation rules, we permit one more restriction: *membership_override*. This specifies whether the policy prerequisite that corresponds to this abstract prerequisite can override membership specification, i.e. in the case that an abstract rule specifies a prerequisite as a normal prerequisite, and adds this restriction to it, an implementing policy may specify membership condition for the corresponding prerequisite.

As the extra components defined with the help of templates cannot be referenced from other components – e.g. if we define extra parameters for a role we cannot refer to them in a rule – there is no need for constraints similar to ones that we specified for the optional parameters.

5.4 Implicit rules

Rules provide a connection between a set and a single rule component, for example an activation rule specifies that given a specific set of prerequisites it is possible to activate a target role. Some policies might specify this connection via several rules.

For example, if we have two roles (*roleA* and *roleB*) in a policy, we can specify a direct role activation rule ($roleA \vdash roleB$), or we can specify two rules, which make use of an intermediary role (*roleC*), as follows:

$$roleA \vdash roleC$$
$$roleC \vdash roleB$$

As there are no additional prerequisites required, everyone in *roleA* will be able to enter *roleB* (through *roleC*). Whether we use a single rule or two rules to get from *roleA* to *roleB* is irrelevant to the fact that we can somehow enter *roleB* if we have *roleA*.

We introduce implicit rules, which specify a set of preconditions, and a target (a role or privilege). A compliance check tests whether the holder of the equivalents of these components (i.e. the versions translated into the policy environments) can enter the target role, or whether he can acquire the target privilege, possibly by entering some intermediary roles, for which the prerequisites are already – or can be – satisfied.

Prerequisite roles in implicit rules behave very much the same way as the prerequisite roles of policy rules. Their parameters may also contain both rule variables and constants. Variables of the implicit rule are handled as constants in the policy environment (Skolem constants). These constants are slightly different from the constants we allow in the compliance policy's abstract prerequisites. The only difference is that the latter constants must be converted to constants used in the policy.

Note that abstract environmental predicates in implicit rules, and their corresponding policy environmental predicates, are not evaluated during compliance checking. Because of skolemisations, which uses the same Skolem constants for every abstract rule variable, the semantics of a translated environmental predicate is not needed. Such abstract implicit rules only ensure that corresponding environmental predicates are used in a chain of policy rules, and their parameters are set in accordance with the abstract rule.

As an environmental predicate, apart from its name, tells little about its semantics, just as in the case of all the other components, verbal comments are necessary.

We allow the use of membership conditions in implicit activation rules. In such cases we require that the component which is marked as a membership condition is a membership condition in the policy, but also, that the intermediary roles entered with the help of membership condition rules are also specified as membership conditions.

Checking implicit rules is more expensive than the checks of direct rules, but this does not affect runtime policy enforcement as both these checks are performed at policy specification time, and not policy enforcement time.

For implicit rule checks we can use the unification engine of OASIS, as the underlying concept of this check is the same as the one of runtime policy decision making.

5.4.1 Negation

Similarly to checking for the existence of rules we can check for their nonexistence. Just as in the check for implicit rules, this check requires a set of prerequisites (roles, appointments and environmental predicates, their parameters assigned a variable or a constant value), and a target role or privilege, which once again can be parameterised. As this check is the opposite of the implicit rule checks, it performs the same steps, but negates the result.

To learn whether it is impossible to enter a specific role is rather difficult. First of all, the entire policy must be considered. However, in principle it would be possible to consider the information flow restrictions of contexts, and thus consider only a part of a policy marked with specific contexts, while excluding other policy parts; support for this can be added to our model, but we leave it as future work. A problem with negation is the complexity of policies and the fact that, through environmental predicates, some policy decisions can be made outside the control of the policy enforcer. For example, if a rule has an environmental predicate, whose semantics is unknown at policy checking time, we cannot consider this rule in our checks. Even if the predicate evaluates to true all the time! Thus our negations mean that, given only the prerequisites, we cannot activate the target via intermediary roles, assuming that every environmental predicate that is not included in the prerequisite list would evaluate to false.

5.4.2 Static Separation of Duties

Related to negation and implicit rules is the check for static separation of duties. In the same way as previous implicit rules, this check requires a non-empty set of prerequisites whose parameters are bound to variables. Instead of a target role or target privilege this implicit rule expects a set of targets (either roles or privileges). These targets we call the conflicting targets. The prerequisites would normally contain the initial roles of specific users or user groups.

This rule can be decomposed into a number of implicit rules that have the same prerequisites – the prerequisites of the separation of duty rule – and a single element from the target set. For a separation of duty check to pass it is required that at most one of these implicit rules is satisfied, and the remaining ones are not satisfied.

Example

As an example consider the following compliance policy:

$$\begin{array}{l} c\_clerk(id : c\_integer) \\ c\_approve(order : c\_integer) \\ c\_initiate(order : c\_integer) \\ c\_ssod : c\_clerk(x) \vdash c\_approve(y), c\_initiate(y) \end{array}$$

This compliance policy specifies that there is a clerk role (*c_clerk*), two privileges (*c_approve* and *c_initiate*), and a static separation of duty rule, that requires that if someone has the *c_clerk* role (or actually the corresponding role in a real policy), then he may only be able to get either the equivalent of the *c_approve* or the equivalent of the *c_initiate* privilege.

The policy we map this compliance policy to is as follows:

$$\begin{array}{l} clerk(id : integer) \\ approve(o : integer) \\ initiate(o : integer) \\ manager \\ auth\_rule1 : clerk(x) \vdash approve(y) \\ auth\_rule2 : manager \vdash approve(y) \\ act\_rule1 : clerk(x) \vdash manager \end{array}$$

The mapping is as follows:

compliance policy	maps to **(map())**	**policy**
c_clerk(x)	$\sim$	*clerk(f(x))*
c_approve(x)	$\sim$	*approve(g(x))*
c_initiate(x)	$\sim$	*initiate(g(x))*

First we decompose the static separation of duty rules into two implicit rules:

$$\begin{array}{l} tmp1 : c\_clerk(x) \vdash c\_approve(y) \\ tmp2 : c\_clerk(x) \vdash c\_initiate(y) \end{array}$$

We then translate these rules with the help of mappings to the policy environment:

$$\begin{array}{l} tmp1 : clerk(z) \vdash approve(w) \\ tmp2 : clerk(z) \vdash initiate(w) \end{array}$$

We can now check whether the above implicit rules can be satisfied in the policy. For this we use OASIS's unification algorithm, which returns that the first rule is satisfied, as having *clerk(z)* (z is a Skolem constant) we can acquire the requested *approve(w)* (w is also a Skolem constant) privilege via the policy's *auth_rule1* authorisation rule. The second implicit rule (*tmp2*) is also satisfiable, as the target privilege can be acquired through the use of the policy's rules *act_rule1* and *auth_rule2*, in which a clerk first enters a general manager role, which is assigned the requested privilege.

As both of these implicit rules are satisfiable, the compliance check, as expected, fails, thus indicating that our original requirement that the two conflicting privileges are not accessible to a single role does not hold for the policy.

5.4.3 Contexts

Contexts, which we introduced in Chapter 4, like compliance policies are a type of meta-policy. They are more tightly integrated to policies, as they annotate policy components, as opposed to compliance policies, which are defined in terms of an abstract policy model. Since contexts are so tightly integrated with policies it seems natural to include restrictions on them in compliance policies.

Contexts could be used in two different ways in compliance policies, but we only allow one of these uses.

Contexts for compliance

First, compliance policies could make use of contexts in the same way as has been done for policies. In this case the contexts of the compliance policy can be used to organise the compliance policy itself, and to impose restrictions on how abstract rules are assembled.

Compliance policies, like policies, could thus include a context element specification, and their abstract components can be annotated with contexts. Such context elements cannot be optional, as they are used only for checking the compliance policy itself.

But, since compliance policies are expected to describe only specific aspects of policies, they should be simple and relatively small in size. Consequently *we do not allow context specification for compliance policies* in the same sense as we have done for policy specifications.

Abstract contexts

The second way in which contexts could be applied to compliance policies is through specifying a context model that is expected to be implemented in a policy. In this case the compliance policy contains an abstract context model, and this must be mapped to the context model of the policy.

The specification of abstract contexts is, as in the case of policy contexts, a specification of labels (context elements) and the information flow relation. Abstract policy elements of the compliance policy are annotated with these context labels in the same way as in the case of policies.

As the contexts in compliance policies specify what contexts should be like in policies, we can include further restrictions. One such restriction is negative information flow, which specifies that information flow between two specific context elements should not be permitted in a policy. The reason for the need for such negative restrictions is that in policy contexts undesired information flow is specified implicitly through the information flow relation. Without negative restrictions we could only check whether information may flow from a particular context element to another; thus a set of context elements that allow all possible information flows would always satisfy the abstract context's specification.

The compliance check for contexts consist of the following stages:

1. First a mapping between the abstract context labels and the context labels of the policy must be provided. Whether we allow the mapping of two abstract context labels to a single policy context label is specified in the mapping restriction (by using the *unique* keyword) of the abstract context element.

2. We must then check whether negative information flow restrictions specified on the abstract context elements are satisfied by the contexts of the checked policy.

3. Once this mapping is provided, together with the mappings of the compliance policy components, the rules of the policy can be checked for context information flow.

In the following example we specify two abstract context elements (c_A and c_B) in the compliance policy. We permit information flow from c_A to c_B, but not the other way around. We annotate the compliance policy components as follows (for simplicity we only display the contexts for rule components – the parameters have exactly the same contexts as their parent component, and rules have no contexts associated with them):

$$\begin{array}{l} c\_student\_role^{[\texttt{c\_A}]}(c\_id : c\_idtype) \\ c\_db\_access^{[\texttt{c\_B}]}(c\_id : c\_idtype) \\ c\_rule1 : c\_student\_role(x, y) \vdash c\_db\_access(x) \end{array}$$

Based on the components' contexts the context annotated rule is:

$$c\_rule1 : c\_student\_role^{[\texttt{c\_A}]}(x, y) \vdash c\_db\_access^{[\texttt{c\_B}]}(x)$$

This rule is permitted, as information flow is permitted between the prerequisite rule component's contexts and the target's context.

If we map these contexts to a policy's contexts (A and B, with information flow permitted from A to B and from B to A, i.e. the two contexts are distinguished only from the perspective of administrative grouping), the compliance check will fail, as expected, at stage 3. The reason for this is that in the policy information may flow from the equivalent of c_B (B) to the equivalent of c_A (A), but this is not permitted by our negative information flow restriction in the compliance policy.

Together with abstract contexts, we can use negative information flow restrictions to specify static separation of duty (SoD) like constraints on policies. We can achieve this by two context labels between which information flow is prohibited, and annotating conflicting policy components with different context labels.

This way of specifying SoD is different to our SoD rules, as this one requires the use of contexts. It is thus more heavyweight than our SoD rules; however, such specification can be used for conflicting sets of roles, as opposed to conflicting roles.

5.4.4 Optionality and templates

Unlike the case of other abstract rules, we disallow the use of optional components in implicit rules. The reason for this is that all prerequisites of these rules behave as if they were optional, and implicit rules are not mapped directly onto policy rules.

Similarly, we disallow the use of template components (or arbitrary components), as we require an unambiguous set of prerequisites to test implicit rules. While it is possible to describe policy components with the help of templates, it is more difficult to map their parameters, and this would unnecessarily complicate implicit rules.

5.5 Compliance as a policy component

Conformance to the restrictions of a compliance policy results in a certificate that itself can be considered as part of a policy. This can be used as a prerequisite to other compliance checks, thus forming a dependency between compliance policies. Indeed, because of the similarity to policies, compliance policies could be specified hierarchically, where higher-level compliance policies specify constraints for lower-level ones. In such a hierarchy, a top-level compliance policy may only specify an abstract component model (a list of abstract roles for example), and lower-level compliance policies would provide rules to reflect various aspects of access control. Our current implementation does not support such hierarchies, but such extensions are not too complicated.

Access to compliance policies does not need to be as finely specified as for policies. Compliance policies are guidelines for policy administrators. Administrators who specify such guidelines are expected to be competent. A compliance policy should describe a specific aspect of a policy, thus a policy usually must conform to many compliance policies.

5.6 Discussion

The examples we have used in this chapter are OASIS specific; however, many of the restrictions specified in compliance policies can be applied to other RBAC models.

5.6.1 Related work

There are many areas that relate to our work presented in this chapter. We have already mentioned federated databases, which address the issue of semantic heterogeneity.

Another broad related area is the field of XML Schema [W3C01] and XSLT [W3C99]. A policy could be considered as a semi-structured document. Indeed, in our current implementation, while our database-centred policy store is under development, we store policies in XML. XML Schemata and XSLT can specify restrictions over such semi-structured data, but our approach is more RBAC specific, and we take RBAC semantics into consideration.

In [MS93], Moffett and Sloman introduce policy hierarchies. Their work is motivated by a desire to automate the management of very large-scale distributed systems. They introduce different levels of policy abstractions, in which they derive lower-level policies from higher-level ones. This derivation is based on goal refinement, target partitioning or delegation.

As mentioned earlier our compliance policies can also be organised into a hierarchy, but because we expect compliance policies to be small in size, we do not put much emphasis on this feature. Goal refinement is achieved via compliance policies themselves, where the goal is specified in the compliance policy. Support for delegation is provided via OASIS appointments.

Wies considers various policy definitions in [Wie94]. In this work he classifies policies according to different aspects, such as target types, triggering mode, activity, and so forth. This paper also describes policy refinement along a hierarchical policy structure. It differentiates four policy abstraction levels: *Corporate level* (high level policies are directly derived from corporate goals), *task-oriented policies* (these define how management tools should be applied), *functional policies* (specifications at management function level), and *low-level policies* (these refer to objects at the lowest, resource access, level).

While this work provides a good motivation for policy refinement and for our compliance policies in particular, it is fairly high-level, and consequently, it lacks implementation detail.

Mont et al. describe a prototype implementation for policy refinement in [MBG99]. In their work the authors introduce templates, which are higher-level policies that can be understood by non-expert policy administrators. Templates are similar to our compliance policies from the perspective that they are intended to provide a general view to a policy, and can later be refined and translated by policy experts into low-level constructs, which in our case are authorisation and activation rules.

In [KKSM02], Kern et al. introduce *enterprise roles* to address the problem of many target systems that live together within one enterprise. These enterprise roles include general information about the target systems, and they bear similarities to our compliance policies. As in our model, enterprise roles have to be mapped onto the ones used by target systems.

Ao et al. introduce a mechanism called law-governed interaction (LGI) [AMN02] to manage message passing in large heterogeneous environments. While this does not use role-based access control, their policies are organised into a superior/subordinate relation that is analogous to our meta-policies. First, these policy hierarchies, similarly to our contexts, help to organise and classify enterprise policies. Second, they help to regulate long-term evolution of enterprise policies. This second aspect of policy hierarchies coincides with our compliance policies.

5.6.2 Future directions

We have already mentioned hierarchical compliance policy specification as one possible path to follow.

Another interesting path would be to refine arbitrary component templates. While we allow the evaluation of certain conditions in our compliance policies (we can check whether an optional component has been mapped or not), our templates could be extended to support further conditions. Templates could be defined in a condition/restriction form; where a *condition*, using the abstract component model, describes the rule component of a policy, and if this condition matches, then we require the rule to satisfy the *restriction* part of the template.

5.6.3 Summary

In this chapter we introduced compliance policies, which use an abstract component model to specify minimum and maximum requirements for policies. We can prescribe what components must be present in a checked policy, what components are optional, and what restrictions extra components must satisfy. In addition, we introduce implicit rules (including negation and static separation of duty), which consider transitive rule evaluations, and form a powerful means to analyse policies. Since compliance policies are specified at an abstract level they are suitable for expressing the desired properties of different policy domains. Because the expected lifetime of compliance policies is longer than the lifetime of policies, this abstraction is used to support policy evolution, in which long-term goals can be encapsulated in compliance policies.

We have considered compliance policies from the aspect of policy refinement, but in large scale enterprises it is natural to have many policies working in parallel. We shall discuss this in the next chapter, together with other related work, such as [HGPS99], [BdVS02], and [Hom02], which address both aspects of our compliance policies.

6 Interface policy

Large organisations tend to be distributed both geographically and structurally. Different branches or sub-organisations usually have varying levels of autonomy, which might include autonomy over access control specification. In this chapter we shall refer to such autonomous units as domains. In the previous chapter we considered how to restrict freedom in such domains by requiring compliance with restrictions specified in our meta-policies. We also indicated the potential for compliance policies to facilitate cooperation among different domains. In this chapter we explore this aspect of meta-policies, i.e. we consider the use of meta-policies to enable and ease cooperation among domains with different RBAC policies.

The work here is inspired by the National Health Service (NHS) environment, for which the British government set the goal of enabling doctors to access patients' Electronic Health Records (EHR) independently of the locations of doctor and patient. As patient data is stored at hospitals – that are autonomous, and have a considerable level of freedom in specifying local access control policies – this application scenario encompasses many policy domains between which there must be cooperation.

In this chapter we first review the service level agreements of OASIS, and point out some problems that affect long-term policy administration. This is followed by a detailed description of our interface policies, in which we show how to use them through examples. Finally, we review some related work and conclude this chapter.

6.1 Service Level Agreements

OASIS's architecture was designed with distribution in mind, and ever since its first model, OASIS has supported cooperation among different policy domains.

Cooperation between two OASIS servers, each enforcing different policies, is achieved through Service Level Agreements (SLA), which we have already reviewed briefly in Section 2.3.2.

These bilateral contracts specify what roles (referred to as local or *exported roles*) can be used remotely, i.e. whether other policies may use them as foreign roles in their specifications. From the remote policy domain's perspective such roles are *imported roles*.

In addition to the set of roles that can be used remotely, SLAs contain information about how to set up an event channel between the cooperating domains. This channel is used for event notifications to support the revocation of foreign roles that are marked as membership conditions.

While this design is perfectly suitable for setting up cooperation between two domains, in the long term it proves inadequate for large-scale systems, in which there are many cooperating domains whose policies are administered separately.

The first problem arises from policy modifications. Policies evolve, and such evolutionary changes must be reflected in the SLAs. If two policy domains are connected via an SLA, and the policy of one of these domains changes, then the SLA must also be updated, and, since the modified roles could have changed their names and parameter signatures, the second policy might also need to be updated. An SLA update thus involves the coordinated cooperation of the policy administrators of the two domains!

A domain may set up SLAs with a number of other domains, and upon a local policy change policy administrators must update all these SLAs. Thus, a local policy change triggers the revision and possible modification of all the policy domains this local domain is connected to. Such implied remote policy changes could trigger further policy changes, resulting in a cascade.

Many SLAs are not used at all, but they still need to be maintained and updated when policies evolve. For example, in the case of the National Health Service, the EHRs of individual patients are potentially stored at many different hospitals. It is expected that all hospital domains are interconnected via SLAs, but some of these SLAs are idle, e.g. if the two hospitals share no patients. On the other hand, maintaining SLAs can become a serious burden on policy administrators. Even though the SLAs in question are likely to be very similar, it is required that each pair of administrators agree on the SLA that is used.

To overcome these problems we propose *interface policies*, which introduce an intermediary entity between two cooperating domains.

An interface policy can be looked at as being a policy domain itself, with which other domains set up virtual service level agreements. In our terminology we refer to such virtual service level agreements as *mappings*. Whenever two domains need to cooperate, based on the mappings between these two domains, and the interface policy, a service level agreement can be set up automatically. This approach has many advantages. First, whenever a policy changes, only the mappings to such interface policies need to be updated, after which all the previous SLAs are regenerated. Second, the interface policy can act as a mediator to overcome representational differences between different domains. Third, due to the longer expected lifetime of interface policies, major changes that could affect a large number of cooperating policy domains become less frequent.

The use of interface policies is illustrated in Figure 6.1, which shows a set of policy domains that interact with each other using SLAs (left hand side), and the same policy domains using interface policies to generate the same SLAs.

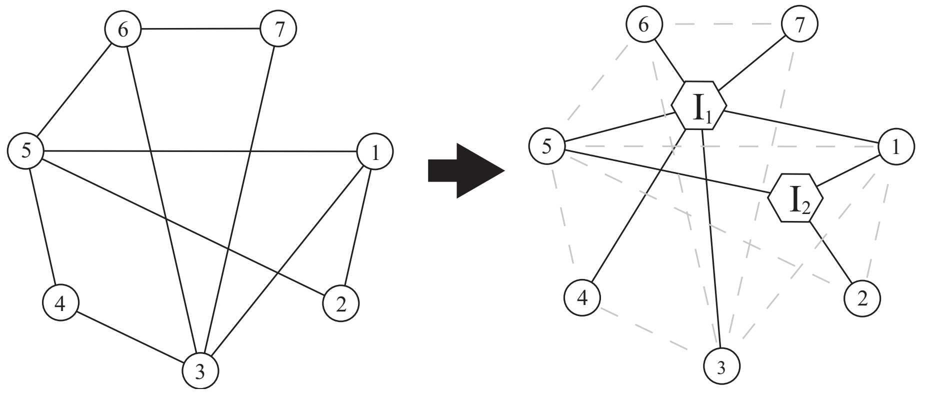


Figure 6.1: The use of interface policies.

In this particular example we have seven policy domains (numbered from one to seven), and ten SLAs[1]. We assume that these SLAs are fairly similar, and we use interface policies to replace them. In the example we introduce two interface policies, and replace the ten SLAs with eight mappings. The SLAs are then generated automatically. Reducing ten SLAs to eight mappings may not seem to be a large improvement, but these mappings are between interface policies and policies, not between policies. If policy number three changes in our original model, the policy change must be reflected in the four SLAs this policy has (between $3-4$, $3-6$, $3-7$, and $3-1$), and this can result in consequent policy changes. With interface policies only one mapping needs to be updated. Furthermore, we could (depending on the mapping directions, i.e. whether a policy is component exporting or importing) now generate SLAs between policies three and five without extra cost. The cost of using interface policies further reduces as the system scales.

Our interface policies build on compliance policies, hence there is a further advantage to our interface policies, viz. we can specify restrictions for the cooperating policy domains. With these restrictions a policy domain that is exporting a role can specify an expected behaviour for the role importing domain, and vice versa.

6.2 Interface policy structure

Interface policies build on compliance policies; hence, they also contain an abstract policy model and a set of restrictions. But, unlike compliance policies, which specify a set of restrictions for a single policy, interface policies must specify two sets of restrictions, as there are two policies involved. These two sets are for component exporting and component importing policies respectively. Just as with compliance policies, a policy needs to map its components to the abstract components of an interface policy, and only then can this policy be checked against the rules of the interface policy. For such compliances we need to know whether a policy is intended to be a component exporting party, component importing party, or both in a particular SLA. We shall refer to this information as the *mode* of a policy. Note that this mode is mapping related, and for a given policy it may differ in various SLA mappings.

The use of interface policies is shown in Figure 6.2.

This figure shows the steps that are necessary for setting up cooperation between two or more domains. First, an interface policy must be specified. We shall discuss its structure later in this section, at the moment the only thing we need to know is that it is similar to a compliance policy, or in fact, to two compliance policies that share their abstract component model. The second step involves the mapping of policies to the interface policy, and performing a test on them, which is similar to the compliance check introduced in the previous chapter. Once this is done, based on the mappings we can set up a service level agreement between a component exporting and a component importing policy. Note that an interface policy can be used for more than one pair of policies, and thus it can be reused many times. If a policy changes, then only the mapping between the policy and the interface policy needs to be updated, and the necessary SLAs will be regenerated automatically. This saves much time for policy administrators, and this also removes from their shoulders the burden of constant checking for remote policy modifications. SLA updates will no longer require the synchronised collaboration of both cooperating policy administrators.

[1]Note that SLAs are directed; however, this is not indicated in the figure, since it is irrelevant to our example.

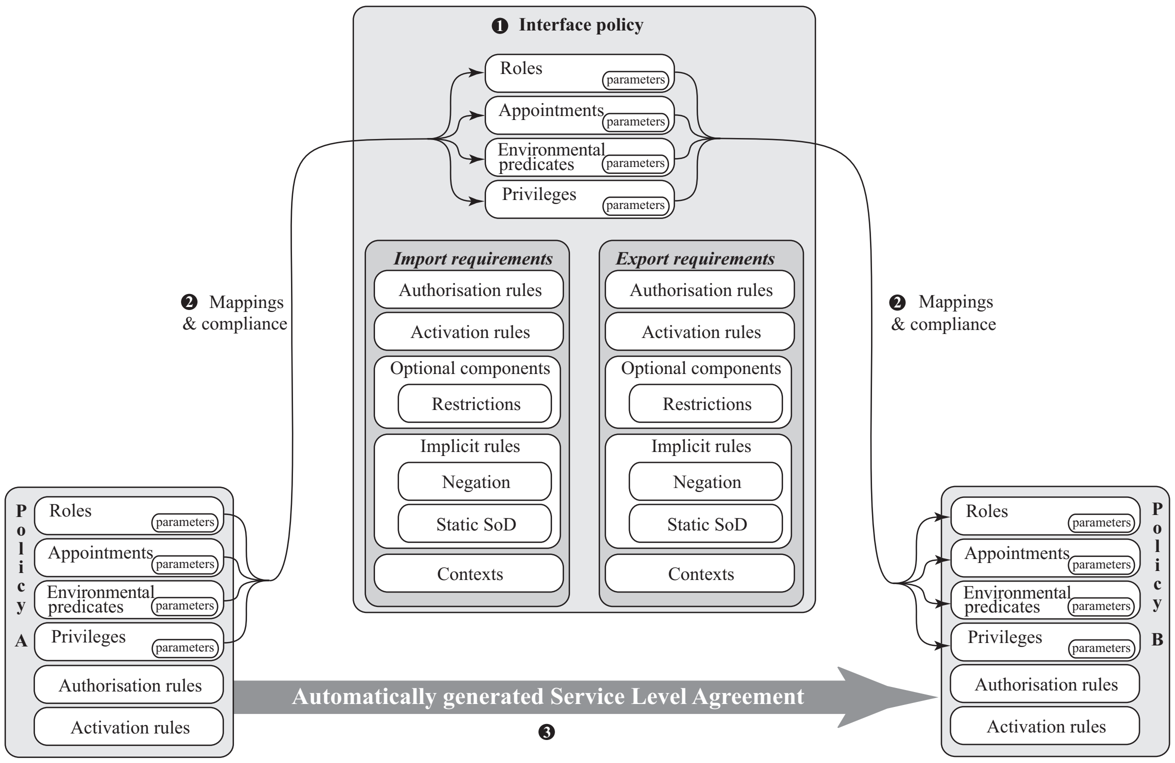


Figure 6.2: Automatically generating SLAs.

6.2.1 Primary interface policy components

An interface specification contains the specification of the abstract policy components that can be shared between two policies. These components can be roles, appointments, environmental predicates, or privileges.

The other two major components of compliance policies are the constraints introduced in Chapter 5. These constraints describe the required properties for both component exporting and component importing policies.

Unlike for compliance policies, where data types were less important, in interface policies we require the use of real data types, which must be understood by the policy enforcing environments of the cooperating parties. This requirement arises because we shall convert data values to and from values of these data types.

Interface policies must specify restrictions for two kinds of mappings, one for component exporting and one for component importing policies. These two restrictions are specified independently.

Some abstract components might not be required to conform to both the component exporting and importing policies. We indicate whether conformance is required with the help of the *imp* and *exp* restrictions.

Example interface policy

We next begin to describe a running example that we shall use in this chapter to show how interface policies work. In this example we would like to enable the cooperation of a few hospitals. We assume that these hospitals have a role that corresponds to a general ward doctor role. This assumption is based on NHS regulations.

As a first step, we specify a data type and an abstract role with one single parameter. We shall use the *i_* prefix in our naming to indicate that a component belongs to our interface policy.

$$\begin{array}{l} i\_NHS\_id:\ xsd{:}integer \\ i\_ward\_doctor\{imp, exp\}(i\_id : i\_NHS\_id) \end{array}$$

With the {*imp, exp*} restriction we indicate that the abstract role must be mapped to by both component importing and component exporting policies.

6.2.2 Mapping restrictions

As we shall convert the component parameters of the exporting policy to data types used in the interface policy, we must require that the functions used in such mappings are concrete functions, i.e. they are known to the policy enforcer. Also, we must specify the mapping direction as follows:

For parameters that will be exported from a policy we require that a function is provided that translates parameter values from the policy's data types to the interface policy's data types. For parameters that will be imported by a policy we require that a mapping function is provided that converts from the interface policy's data types to ones that are used in the importing policy.

As mapping requirements could differ for importing and exporting policies we introduce some new restrictions for the abstract role, appointment, environmental predicate, and privilege components.

Bijections and lossy mapping restrictions: Just as in the case of compliance policy components we require mappings to be bijections, unless they are marked explicitly to allow lossy functions. The *lossy* restriction will allow lossy mappings for both the importing and exporting policies, but this can also be controlled by the *imp_lossy* and *exp_lossy* restrictions, which allow lossy mappings for import and export policies respectively. Note that for import mappings the mapping direction is different from the one in compliance mappings. Accordingly, shared abstract components that are also used in the import side compliance check are involved in mappings of opposite directions. For such components we require bijective mappings.

As in the case of compliance policies, the mapping functions can take many arguments, thus they can split and merge parameter values to overcome semantic differences in the way parameters are stored. Some functions also might make use of external services, such as environmental predicates; for example functions might be used to get a property of a user session. The default bijection restrictions are stricter than the *lossy* ones, since bijections may always be used instead of lossy functions. A mapping function may have more than one parameter. In such cases the strictest parameter restriction shall be considered.

Example mapping restriction

We shall extend our interface policy to include restrictions on the kind of mappings we accept.

$$\begin{array}{l} i\_NHS\_id : xsd : integer \\ i\_ward\_doctor(i\_id\boldsymbol{\{exp\_lossy\}} : i\_NHS\_id) \end{array}$$

We specify that export mappings for our parameter (*i_id*) could use lossy functions. This allows the exporting domain to use multiple identifiers for the same doctor as long as these identifiers can be converted to an NHS doctor id.

Active components: Imported rule components could be used as membership conditions in a component importing policy. If so, the importing policy enforcer needs to know whether a particular imported rule component is revoked or not. SLAs usually specify an event channel for informing importing domains about rule component revocations, however this channel consumes resources (e.g. it has a heartbeat as part of its protocol). To avoid unnecessary resource consumption we require the explicit specification of activity for components, i.e., in the shared abstract policy component we must specify whether the validity of this component can be monitored. In order to be able to check such validity, the exporting policy must keep track of all the policy components that are exported and need to be monitored.

To indicate that a rule component (role, appointment or environmental predicate) could be used as a membership condition in a component importing policy we use the *imp_active* restriction.

Queriable components: The representation of a shared abstract role could differ in two different policies. Our interface policies only ensure that we are able to translate a policy component given in the exporting policy's domain to a role of the importing domain. This process might involve lossy functions, therefore we cannot guarantee invertibility.

Although, in cases when only bijection mappings are used (and function inverses are known), we could translate requested import policy components to components of the exporting policy, we do not support this functionality.

As a consequence, a principal that needs to request a remote policy component that requires more than one local (to the exporting policy) component, must present all of the local prerequisites to the component importing policy domain. How does a principal know about what prerequisites it might need, given that it knows little about the internals of the remote policy? We answer this in the next section.

6.2.3 Restrictions on policies

Interface policies may contain restrictions on both component importing and exporting policies. These restrictions are the same as those we have seen in compliance policies. They are able to check whether an importing or exporting policy contains a specific rule, whether they provide a means to enter some role or acquire some privilege, and so forth. With their help, together with the abstract model of the interface policy, it is possible to specify restrictions over the use of the foreign policy components. For example, it can be specified that a role to be shared is used only for privilege assignments in the role importing policy, and that this role satisfies some separation of duty like constraints in the role exporting policy. Note, however, that it is sometimes possible to circumvent these restrictions with the help of badly specified mappings or with additional interface policies that outsource some policy decisions.

Example interface restrictions

We extend our example with a restriction on the importing domain:

$$\begin{array}{ll}
\multicolumn{2}{l}{i\_NHS\_id : xsd : integer} \\
\multicolumn{2}{l}{i\_patient\_id\{imp\} : xsd : integer} \\
\multicolumn{2}{l}{i\_ward\_doctor(i\_id\{exp\_lossy\} : i\_NHS\_id)} \\
\\
\multicolumn{2}{l}{i\_read\_haematology\_record\{imp\}(i\_patient : i\_patient\_id,} \\
\multicolumn{2}{l}{\quad \_parameter\{min(0), max(unbounded)\})} \\
\multicolumn{2}{l}{i\_patient\_doctor\{imp\}(i\_patient : i\_patient\_id,\ i\_doctor : i\_NHS\_id,} \\
\multicolumn{2}{l}{\quad \_parameter\{min(0), max(unbounded)\})} \\
\multicolumn{2}{l}{\text{Restriction for the importing domain:}} \\
i\_rule1\{imp\} : & i\_ward\_doctor(x),\ i\_patient\_doctor(x, y) \\
& \vdash i\_read\_haematology\_record(y)
\end{array}$$

Here we first added a new data type (*i_patient_id*) and a privilege (*i_read_haematology_record*) that uses this data type for its parameter. We only require that these two are present in policies that act as a component importing policy.

We also specify an implicit rule for the importing policy, in which we require that if a principal holds the equivalents of the rule (*i_rule1*) prerequisites, then this principal must be able to get the equivalent of the target privilege (*i_read_haematology_record*) without providing any further prerequisites.

The compliance restrictions also must include wild-card elements to permit both the importing and exporting policies to have additional policy components.

6.3 Compliance

Once an interface policy is specified it is ready to be used by policies. In the same way as for compliance policies, the first step is to set up a mapping between a policy and the interface policy. We shall discuss this in Section 6.3.1. Once this mapping is provided we can perform the compliance check (see Section 6.3.2).

6.3.1 Mapping

Mappings between a policy and an interface policy must be set up in accordance with the mode of the policy and the mapping restrictions of the interface policy. For component exporting policies these mappings are the same as for compliance policy mappings, since both for policy restrictions and exported components it is required to map policy components to interface policy components.

However, for import policies there are two mapping directions involved. The first direction, from the policy to the interface policy, is required for the compliance restrictions, if any. The second direction, for the ‘shared’ policy components, is from the interface policy components to the policy components. Clearly, lossy functions are not allowed if a policy component needs to be mapped in both of these directions.

Example mapping

In order to continue with our example we need to specify two policies that will be mapped onto the interface policy.

To increase readability we shall use here the prefixes *a_* and *b_* in the exporting and importing policies respectively.

Export policy: The first policy, which will be the component exporting one, is as follows:

$$\begin{array}{l} a\_string \\ a\_ward\_doctor(name : a\_string) \end{array}$$

In this policy we specify only a data type and a role (for simplicity we include only the necessary elements). This role (a_ward_doctor) has a single parameter of data type *a_string*. This string is used to uniquely identify a doctor in a hospital that is using this policy. For example, an instance of such a role could look like *a_ward_doctor('Alice')*.

The enforcer of this policy must provide a function to translate the parameter of the local ward doctor role to the parameter of the interface policy's role. This function is *a_get_NHS_id(a_string)*.

Using this function the mapping of the component exporting policy to the interface policy can be given as follows:

$$a\_ward\_doctor(x) \rightarrow i\_ward\_doctor(a\_get\_NHS\_id(x))$$

Thus, whenever we have an instance of the *a_ward_doctor* role, we can transform it to the format specified by the interface policy.

The requirement for the *a_get_NHS_id* function is that it must be able to map every possible local parameter to a parameter of the interface policy role, but it need not be invertible (this is specified by the *exp_lossy* restriction).

Import policy: Because the interface policy specifies compliance restrictions for the importing policy, we must provide a slightly more complicated policy:

$$\begin{array}{l} b\_doctor\_id \\ b\_patient\_id \\ b\_record\_id \\ b\_ward\_doctor(doctor : b\_doctor\_id) \\ b\_patient\_of(doctor : b\_doctor\_id,\ patient? : b\_patient\_id) \\ b\_read\_record(patient : b\_patient\_id,\ type : b\_record\_id) \\ b\_authrule1 : ward\_doctor(x),\ b\_patient\_of(x, y) \vdash b\_read\_record(y) \end{array}$$

This policy has three data types (b_doctor_id, $b\_patient\_id$, and b_record_id), a ward doctor role (b_ward_doctor), an environmental predicate ($b\_patient\_of$) that checks whether a patient is treated by a doctor, and a privilege that gives access to a patient's particular record that is specified by the *type* parameter.

The mapping of the components is as follows:

$$\begin{array}{rl} i\_ward\_doctor(x) & \rightarrow\ b\_ward\_doctor(b\_f(x)) \\ i\_patient\_doctor(y, z) & \rightarrow\ b\_patient\_of(b\_f(z), b\_g(y)) \\ i\_read\_haematology\_record(w) & \rightarrow b\_read\_record(b\_g(w),\ 'haematology') \end{array}$$

For this mapping the conversion functions *b_f* and *b_g* need to be provided by the policy environment of the importing policy. *b_f* transforms a value of *i_NHS_id* type to a value of *b_doctor_id* data type. This forms the basis of the mapping between the abstract ward doctor and the policy's ward doctor role. Similarly, with the help of *b_g*, which takes care of the conversion between *i_patient_id* and *b_patient_id*, we can map the abstract environmental predicate *i_patient_doctor*. Note that the abstract predicate allows additional parameters for the policy parameters, but in this policy there were no such extra parameters. Nevertheless, the mapping has still taken care of the parameter order, which is reversed in the importing policy.

Finally the mapping for the privilege *i_read_haematology_record* is given. Note that the privilege in the policy has more parameters, which is permitted by the interface policy specification. Also, the policy's *b_read_record* privilege is much more powerful than the abstract privilege of the interface policy, but this is restricted by a constant *'haematology'*. This constant can be looked at as a constant function that takes no parameters and returns a single value.

In this example we have seen that an abstract privilege could be mapped onto a more powerful one. Similarly a role could be mapped onto a much more powerful policy role. To avoid such mappings, when possible, the abstract policy components must provide plenty of verbal comments. In the case of roles, negative implicit rules can help to mitigate the above problem, by restricting the set of privileges a policy role, to which the abstract role is mapped, may have.

6.3.2 Compliance check

The next step before generating an SLA is to check whether a policy, be it component exporting or importing, complies with the restrictions specified in the interface policy. These are based on the checks of compliance policies. However, for component importing policies we must check whether imported policy components are used as membership conditions in accordance with the abstract policy components' activity restriction.

Example compliance check

In our example the compliance check for the component exporting policy ensures that the necessary mappings are provided. For the importing policy the compliance check includes, in addition to the mapping checks, the examination of the policy rules, in order to find a set of rules that satisfy the interface policy's implicit rule.

The implicit rule is:

$$\begin{aligned} i\_rule1\{imp\}: \quad & i\_ward\_doctor(x),\ i\_patient\_doctor(x,y) \\ & \vdash i\_read\_haematology\_record(y) \end{aligned}$$

This translates to:

$$\begin{aligned} translated\_rule1: b\_ward\_doctor(b\_f(x)),\ & b\_patient\_of(b\_f(x), b\_g(y)) \\ & \vdash b\_read\_record(b\_g(y),\ 'haematology') \end{aligned}$$

Note that as x and y were variables, in this check $b\_f(x)$ and $b\_g(y)$ behave as if they were simple constants (Skolem constants). This is an implicit rule, but luckily there is a direct rule in the policy that satisfies this implicit rule. (If there were no such direct rules, then it would be necessary to check whether the privilege could be acquired via any intermediary roles that can be activated with the prerequisites of the translated implicit rule or via other intermediary roles.) The remaining part of the policy still needs to be checked, but these are irrelevant wild-card checks.

As a final step we must analyse the importing policy's role activation rules to see whether the imported role, which did not contain the *imp_active* keyword among its restrictions, is used as a membership condition.

6.4 Generating SLAs

In order to generate service level agreements between two policies they must be mapped to an interface policy. One such mapping must be in component exporting mode, while the other is component importing. In an SLA we join these two mappings.

Example SLA generation

According to our interface policy there is only one role – *i_ward_doctor* – that is used in the cooperation between an exporting and an importing domain. Therefore, if a principal in the exporting domain has activated the *a_ward_doctor('Alice')* role then he should be able to use this role, or actually a translated version of it, in the importing policy domain. The first part of the translation is done on the exporting policy side. This part translates a role instance to an abstract role instance. For example:

$$i\_ward\_doctor(\text{'}Alice\text{'}) \rightarrow i\_ward\_doctor(a\_get\_NHS\_id(\text{'}Alice\text{'}))$$

This abstract role instance can be sent to the policy component importer, which performs the second step of the translation:

$$i\_ward\_doctor(a\_get\_NHS\_id(\text{'}Alice\text{'})) \rightarrow b\_ward\_doctor(b\_f(a\_get\_NHS\_id(\text{'}Alice\text{'})))$$

Thus we are able to translate a role of the exporting policy to a role of the importing policy. This role can then be used in the importing domain in accordance with its policy to either activate further roles or to acquire privileges.

6.4.1 Trust between domains

A policy may comply with many interface policies. Consequently, two policies that intend to cooperate may share more than one interface policy with which they comply in the required way (import and export directions). Interface policies are not equivalent. Some allow the remote use of powerful policy components, while others may concern fairly weak privileges. There are many factors that can be considered when making a decision about which interface policy to choose.

Just as in the case of compliance policies, compliance with an interface policy does not necessarily mean that the mappings are correct. Whether a service level agreement is set up automatically between two domains depends on the domain administrators.

An interesting extension to our work could be to base SLA generation on trust [BFK99]. Such trust could consider past cooperation experience, recommendation, micro payments, or other trust, e.g. trust in the third parties that approved a policy domain's compliance with a particular interface policy.

Based on the trust levels between two domains, we can choose what interface policy to use to set up an SLA. This is illustrated in Figure 6.3.

Our meta-policies store a great deal of information about policies, and as some of them are abstracted away from a policy, their content is less sensitive than the content of a policy itself. Our framework allows controlled querying of this information, and this can be used in trust negotiation protocols, such as ones described in [Yao02], which uses OASIS-like policies and X.509, [SB02], that advocates the use of micro-payments, and [SWY^{+}02], that uses an incremental trust negotiation protocol, which discloses trust policies based on trust itself.

Trust and RBAC have also been discussed in [LMW02] and [WFSM02].

The way that trust is determined is not part of the model presented in this thesis, nor is it part of our policy management framework, however this future direction is worth pursuing.

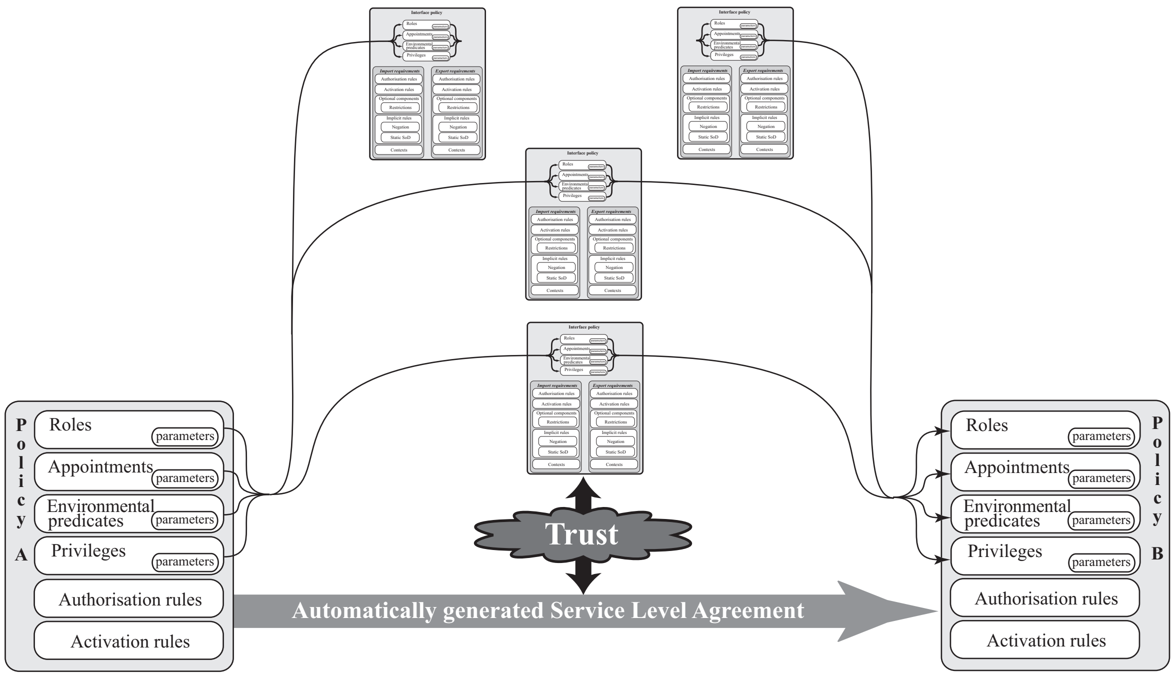


Figure 6.3: Automatically generating SLAs based on trust.

6.5 Discussion

Interface policies, while definitely a help, often cannot be reconciled, and thus mapped onto policies that intend to cooperate. In such cases, either policies must be modified, or a new interface policy specified. Still, interface policies can serve as implementation templates. Their component model can be adopted by policy domains, and used as a skeleton for the local policy. An added bonus of using interface (or compliance) policies as templates is that it will be simple to specify mapping to this particular meta-policy, and the mapping functions will be bijections. This will facilitate interoperation between policies that have been implemented using the same interface policy as a template.

6.5.1 Future work

We have already mentioned one possible path to follow in extending interface policies. This was to use trust in deciding what domains may cooperate, and use the same trust to select a service level agreement.

Another possible extension is to consider the mapping between interface policies themselves (see Figure 6.4). This would introduce further translation steps in SLAs, but could be useful in environments where the administration of interface policies is localised. In such cases, policies cooperate under different local interface policies, and mapping between these interfaces could enable the cooperation of these policy groups.

6.5.2 Related work

Our work relies on research done in the area of federated databases, which we described in Chapter 5. There we mentioned only the use of semantic values as a method of handling semantic heterogeneity. Another popular method to tackle this problem is to use mediators [Hul97]. Such mediators form a separate component, and define an integrated view of many

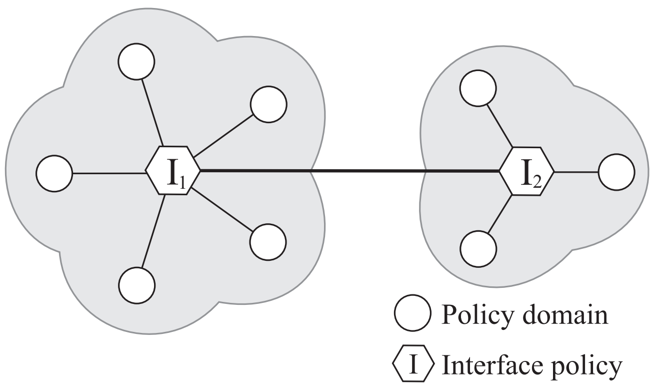


Figure 6.4: Mapping between interface policies.

databases in a federation. This view provides its own schema, and queries can be made only against this schema.

Such mediators were used by Hombrecher to reconcile semantic and structural heterogeneity problems in event systems [Hom02]. In his thesis, Hombrecher uses event gateways to convert events (which, being structured data, are very similar to both database entities or policy components) from one representation to another. In Hombrecher's model all conversion takes place in such gateways, which are hosted on dedicated computers. In our model an interface policy is only a specification, and conversions take place in two phases at the cooperating parties. Also, our approach can, in addition to conversions, specify restrictions over the cooperating domains.

Many other middlewares, such as CORBA [OMG02], MPI [MPI97], and JMS [Sun01], try to address the problem of various representations used by cooperating computers. A general overview of the associated problems is discussed in [Has00]. Our approach builds on the results of such research, and our implementation utilises these middlewares by partially adopting their data type and component systems.

XML, while facilitating data sharing among diverse applications, introduces many challanges that resemble those of heterogeneous databases. Data shared by diverse applications differ, and such differences must be reconciled maually [SR01], similarly to how it is done with our interface mappings.

Interface policies enable cooperation between two or more RBAC, specifically OASIS RBAC, policies; therefore, next we narrow our focus to related work on access control policies.

An outstanding paper by Hale et al. [HGPS99] considers security policy coordination in heterogeneous and distributed environments. Similarly to federated databases, their work uses mediators to reconcile conflicts. This work is more general than ours, since it considers various access control models, among them MAC, DAC, RBAC, and TBAC (Task-Based Access Control). As a consequence, their work must abstract away from the individual policies of these models. Hale et al. do this with the help of a ticket-based primitive access control model, which uses *locks* and *keys* – for the subjects and objects of the access control respectively – to determine what subject has access to what object. They show how to convert a policy of each of the above four more sophisticated models into such a primitive policy. Note, however, that conversion to a primitive policy language requires sacrifices, and as a consequence, many activity-related constraints of RBAC models cannot be supported.

The idea of using a primitive policy language has also appeared in the work of Bonatti, De Capitani di Vimercati, and Samarati [BdVS02]. In their paper the authors use (subject, object, authorisation) triples to express various access control policies. Their algebra is able to express unknown policies, which are equivalent to our environmental predicates.

However, both the work of Hale et al. and Bonatti et al. consider much more general access control models than our interface policies, which stay at the level of RBAC. We specify constraints over the RBAC policy itself, hence the name meta-policy, and not on the access that is permitted. That is, we can express requirements on the RBAC policy component level, as opposed to the user, object, access level.

6.5.3 Summary

In this chapter we introduced interface policies – our final class of meta-policies. We showed how to use interface policies to set up cooperation between two or more policy domains by specifying a set of shared components and restrictions for the cooperating parties. We also described how to generate service level agreements between two policies automatically, and how this process can make use of inter-domain trust relationships.

7 Controlling access to policy

An important motivation for the research presented in this thesis comes from policy storage and access control to such stored policies.

Access control policies change, and in the previous chapters we considered how this change can be restricted at a high level. However, access control policies constitute resources themselves. The policies discussed in this thesis are built up from policy components (see Chapter 3), access to which can be controlled by the same mechanisms that the policies contain. In this chapter we shall explore how to use RBAC to restrict access to our RBAC policies.

Policy components could be accessed through various APIs. However, due to the fact that policy components can be accessed in a limited number of ways, these APIs will be similar. The elementary steps of these API functions can be described with the help of privileges. These privileges will also define the granularity of access control to policies. Since our policy components are rather complex, it is difficult to encapsulate fine-grained accesses in a single privilege or API call. Breaking up component accesses into many steps can introduce transient component states, which can lead to component inconsistencies. To avoid such inconsistencies in policy components we introduce in Section 7.1 the concept of *binding*.

In Section 7.1.3 we specify the privileges for accessing policy components. In this session we also demonstrate the expressiveness of these privileges and we describe how to aggregate component accesses through the use of wild-cards.

In Section 7.2 we extend our privileges with contexts, thus enabling more flexible scoping.

Finally we shall review related research and provide a short summary of this chapter.

7.1 Consistency of policy components

Basic policy components are accessed in various ways during policy evolution. As policies are modified, we must ensure that policy modification results in a consistent policy, i.e. for example there are no references to unspecified roles or data types. There are several levels of consistency requirements. We have already considered higher, policy-level ones in the form of meta-policies. In this chapter we shall focus on lower, component-level consistency requirements.

7.1.1 Binding policy components

Most APIs, like the one defined in our proof of concept implementation (Desert), consist of methods that have a fixed number of arguments. However, many policy components refer to a variable number of other policy components. For example, a role can have any number of parameters, or a rule can have any number of prerequisites. Consequently, a single method call to create or to delete a policy component is insufficient. On the other hand, if more than one method is used for a policy component creation or deletion, inconsistent component states may be introduced. For example, if the specification of a role consists of adding role parameters one by one, the role can only be used when its last parameter has been added, and intermediate states should not be visible to methods that would use this role. To address this problem we introduce the concept of *binding*. Whenever a policy component is created it is in a so called *unbound state*. This means that it is not part of the final policy, thus it cannot be used to construct other components. When the component reaches a state that is considered to be final the component can be bound to the policy. After a successful binding the component becomes part of the policy specification and its state changes to *bound state*. This binding behaves the same way as transaction commits in database management systems. The process of binding a component to a policy includes consistency checks, which we shall describe in Section 7.1.2.

Similarly, policy components cannot be modified freely. A new parameter cannot be added to a role if this role, without its new parameter, is used as a prerequisite in some rules. Therefore, before a component can be modified or deleted it must be in the unbound state. Unbinding a component also requires some consistency checks, as it must be ensured that the component that is unbound is not used by other bound components.

This state transition is illustrated in Figure 7.1. This figure shows that a policy compo-

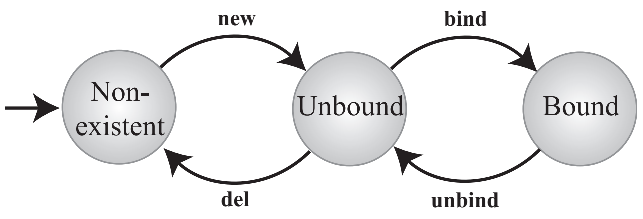


Figure 7.1: The state transitions of a policy component.

nent created by a *(new)* method will be in the *unbound* state. Once in this unbound state a component may be deleted by a *del* access method. Alternatively, from the unbound state the policy component can enter a bound state via *bind* method calls. These method calls are also responsible for maintaining general policy consistency, thus the consistency of the policy should not change if a new component is bound to it. Similarly, *unbinding* methods unbind a policy component only if this operation does not effect the consistency of the policy.

Rules are exempt from the above state restrictions, since they do not have other policy components depending on them, and thus they cannot be referenced while they are in an inconsistent state. However, rules that are in the bound state must be consistent. What this means we shall discuss in the next section (Section 7.1.2).

Separating the addition of policy components into two stages, which lead first to unbound then to bound state, has some additional advantages, since it will enable us to differentiate between accesses that modify existing components and accesses that create new components. In Section 7.1.3 we shall discuss this in more detail.

7.1.2 Consistency of policy components

In the following we list some operation level consistency requirements for each basic policy component.

Data types

Data types can only be unbound if they are not used in any policy component specification. The only operation that is allowed for a bound data type is the addition or removal of a child in the data type inheritance hierarchy. For this operation the parent data type must be in bound state, while the child data type must be in unbound state.

Data types in policy specifications refer to real data types, whose general availability can also be checked. Note that this check is the responsibility of the policy enforcer.

Finally, we must ensure that within a policy a bound data type is uniquely identified by its name.

Functions

Functions must have a valid (bound) data type as their return type. Similarly all the parameter data types must refer to a valid data type, and the parameter names must differ from each other. A function must also refer to a real function that is accessible to the policy enforcer; however, this consistency requirement could be checked at a later stage, as information needed to check this might not be available at policy specification time. Thus, it is the responsibility of the policy enforcer to check if the policy enforcing environment supports the real-world functions to which policy specifications refer.

Just as in the case of the data types the function identifier must be unique in a policy. Our implementation currently does not support polymorphism, thus function names must be unique within policies. Note, however, that we can still refer to concrete functions that are polymorphic.

Functions, just as every other policy component, cannot be unbound if they are referenced from other policy components.

Environmental predicates

For environmental predicate specifications the consistency checks are the same as in the case of functions. Note that the association of environmental predicate names to real-word services/environmental predicates can be complicated, since they can refer to external entities such as web services or other OASIS services.

Privileges, appointments, and roles

From the perspective of consistency, privileges, appointments, and roles behave similarly. They must satisfy the same restrictions as functions, viz. their parameters must refer to bound data types, and they must be unique within their parent component.

At policy specification level the names of these components must also be unique, just as in the case of other components.

Similarly, these components can only be unbound if no bound components reference them.

Authorisation and activation rules

The modification restrictions to both authorisation and activation rules are more relaxed as these rules do not have other policy components that depend on them.

We distinguish two types of operations on these rules. The first type does not affect the component's consistency. Examples include the addition of a variable, the removal of an unused variable, the addition of a prerequisite, and its removal if it does not affect parameter binding. The second type of operations affect consistency. An example for such an operation is the deletion of the target container.

Name	*Parameters*
`type_new`	`(t type_id, rt string, p policy_id )`
`type_del`	`(t type_id, p policy_id )`
`type_inherit_new`	`(parent type_id, child type_id, p policy_id )`
`type_inherit_del`	`(parent type_id, child type_id, p policy_id )`
`type_bind`	`(t type_id, p policy_id )`
`type_unbind`	`(t type_id, p policy_id )`

Table 7.1: Privileges for accessing data types.

We allow both categories of these operations in the unbound state of a rule, but for bound rules we restrict the set of permitted operations to ones that leave modified rules consistent. Such differentiation of access methods will increase the expressive power of our policy access privileges. We shall provide an example for this in Section 7.1.3.

We require that the variables of a consistent rule refer to valid data types only.

The prerequisites and the targets of the rules are embedded into containers. Such containers must only refer to valid policy components, and they must bind each parameter either to a valid function container, or to a constant of the appropriate data type, or to a variable of the rule. Restrictions on free variables (as described in [BMY02]) must also be satisfied.

Consistent rules thus can only contain containers that are consistent. Other consistency restrictions include the uniqueness of rule names.

7.1.3 Privileges to basic policy components

Next we look at the privileges required for access control to a policy management API. Like the consistency restrictions, these are organised according to the policy structure; for each component we shall discuss the relevant privileges and give small examples.

Data types

The privileges associated with data type management are given in Table 7.1. These correspond to the operations of adding, deleting, and modifying data types (like adding and deleting inheritance relations).

These privileges are not method calls, however they resemble the methods that could be part of an API for accessing policy components.

All the privileges include a parameter that identifies a policy. This allows the specification of policies that control access to number of policies.

The *type_new* privilege is required for associating a *t* type id (usually a string) to a data type reference (*rt*), which is usually given as URI.

The *type_del* privilege allows the deletion of a data type from a policy. Note that this is just a permission. Whether a principal that has such a privilege can delete a data type may depend on additional conditions, such as the state of the data type in question, i.e. whether it is in bound or unbound state.

The *type_inherit_new* and *type_inherit_del* privileges deal with permissions for the addition and deletion of type inheritance relations.

To give a role *my_role* the privilege of setting an inheritance relation between two types – in our case let these two types be identified by the strings ‘integer’ and ‘natural’ – one should use a rule like:

$$my\_role,\ string\_eq(x?,\ \text{‘}integer\text{’}),\ string\_eq(y?,\ \text{‘}natural\text{’})\ \vdash type\_inherit\_new(x?,\ y?)$$

The above example is obviously contrived and such rules would be of little use in real policies. However, free variables can also be used in authorisation rules. In our rule, omitting the *string_eq*($x?$, *‘float’*) and *string_eq*($y?$, *‘natural’*) prerequisites would result in giving the *my_role* permission to add any data type hierarchy relation. Or, removing just one of these prerequisites, for instance *string_eq*($x?$, *‘float’*), would permit the role to add inheritance relationships where the child data type is restricted to the data type identified by ‘*natural*’.

Functions

Functions can be created (in a policy this means an association with a function known by the policy engine), deleted, and modified. Associated privileges are given in Table 7.2.

Name	*Parameters*
func_new	(f func_id, rf string, p policy_id)
func_del	(f func_id, p policy_id, p policy_id)
func_retval_new	(f func_id, ret_type type_id, p policy_id)
func_param_new	(f func_id, param_name string, param_type type_id, p policy_id)
func_param_del	(f func_id, param_name string, p policy_id)
func_bind	(f func_id, p policy_id)
func_unbind	(f func_id, p policy_id)

Table 7.2: Privileges for accessing functions.

In order to create a new function a principal must hold the *func_new* privilege. The parameters of this privilege may specify restrictions on the identifier of the function, on its reference to a real function (e.g., we may issue a privilege that allows the creation of an integer to string conversion function only) and on the policy of which this function should be a part.

To set the data type of the return value of a function a principal needs the *func_retval_new* privilege.

With the help of *func_param_new* privilege we can restrict the parameters that can be added to a particular – or to any – function.

The *func_bind* and *func_unbind* privileges control rights to bind and to unbind a function policy component. An advantage of having bound and unbound components is that we can differentiate between creating new policy components and modifying existing ones.

To allow a user to modify a function we can grant him both the *bind* and *unbind* privileges. However, to allow a user to add a new function, without allowing him to modify existing ones, we can grant him the $func\_bind$ privilege only.

Roles

In Table 7.3 we list the privileges associated with role specification. These privileges control whether a principal may or may not create, modify, or delete a role. Note that these privileges concern only the existence of roles. How these roles are used is specified by rules, which we shall discuss later.

Name	*Parameters*
`role_new`	`(r role_id, p policy_id )`
`role_del`	`(r role_id, p policy_id )`
`role_param_new`	`(r role_id, param_name string, param_type type_id, p policy_id )`
`role_param_del`	`(r role_id, param_name string, p policy_id )`
`role_bind`	`(r role_id, p policy_id )`
`role_unbind`	`(r role_id, p policy_id )`

Table 7.3: Privileges for accessing roles.

Appointments

Appointment-related privileges are similar to those of roles. They are specified in Table 7.4.

Name	*Parameters*
`app_new`	`(a appointment_id, ra string, p policy_id )`
`app_del`	`(a appointment_id, p policy_id )`
`app_param_new`	`(a appointment_id, param_name string, param_type type_id, p policy_id )`
`app_param_del`	`(a appointment_id, param_name string, p policy_id)`
`app_bind`	`(a appointment_id, p policy_id)`
`app_unbind`	`(a appointment_id, p policy_id)`

Table 7.4: Privileges for accessing appointments.

Appointments must be bound to real-world appointment certificate types, so that at policy enforcement time the integrity of these certificates can be checked adequately. This binding is done with the help of the *ra* parameter.

Environmental Predicates

The specification of environmental predicates is managed in the same way as in the case of functions. These privileges are listed in Table 7.5.

Privileges

The privileges for managing privilege in a policy are listed in Table 7.6. They are structurally the same as those for role access.

It is possible to extend our privileges to support control over privilege inheritance. Such an extension is given in Table 7.7.

A similar extension could be provided to support hierarchical relations between roles.

Name	*Parameters*
env_new	(e env_id, re string, p policy_id)
env_del	(e env_id, p policy_id)
env_param_new	(e env_id, param_name string, param_type type_id, in_out boolean, p policy_id)
env_param_del	(e env_id, pname string, p policy_id)
env_bind	(e env_id, p policy_id)
env_unbind	(e env_id, p policy_id)

Table 7.5: Privileges for accessing environmental predicates.

Name	*Parameters*
priv_new	(p priv_id, p policy_id)
priv_del	(p priv_id, p policy_id)
priv_param_new	(p priv_id, param_name string, param_type type_id, p policy_id)
priv_param_del	(p priv_id, param_name string, p policy_id)
priv_bind	(p priv_id, p policy_id)
priv_unbind	(p priv_id, p policy_id)

Table 7.6: Privileges for accessing privileges.

Name	*Parameters*
priv_dep_new	(p priv_id, priv_dependent priv_id, p policy_id)
priv_dep_del	(p priv_id, priv_dependent priv_id, p policy_id)

Table 7.7: Privileges for accessing privilege hierarchies.

Authorisation rules

Authorisation rules are among the most critical components of a policy from the point of view of both access control and self-administration. A role with privileges to modify rules can control the assignment of privileges to policy roles, which may even include the role itself, thus it is possible that a role has the power to modify its own privileges.

The privileges and methods we introduce in Table 7.8 and Table 7.10 allow a very fine-grained control that not only allows limiting the use of certain prerequisites, but also can impose restrictions on the parameters of the prerequisites. This is achieved with the help of simple environmental predicates – e.g. arithmetic operators – that can be part of authorisation rules.

Rule prerequisites are encapsulated into containers, privileges for which are given in Table 7.8.

These privileges allow the creation of containers for privileges, roles, appointments, and environmental predicates. As the privileges for these components are separated, it is simple

Name	*Parameters*
`cont_priv_new`	`(c contp_id, p priv_id, r arule_id, p policy_id)`
`cont_role_new`	`(c contr_id, r role_id, r arule_id, p policy_id)`
`cont_app_new`	`(c conta_id, a appointment_id, r arule_id, p policy_id)`
`cont_env_new`	`(c conte_id, e env_id, r arule_id, p policy_id)`
`cont_del`	`(c {priv, role, app, env}_id, r arule_id, p policy_id)`
`cont{prae}_param_new`	`(c cont{prae}_id, t term_id, io boolean, r arule_id, p policy_id)`
`cont{prae}_param_del`	`(c cont{prae}_id, nr int, r arule_id, p policy_id)`

Table 7.8: Privileges for accessing authorisation rule containers.

to specify policies which allow the addition of role prerequisites, but disallow appointment prerequisites.

Container parameters are handled through the *cont{prae}_param_new* and *cont{prae}_param _del* privileges, where {*prae*} is replaced with the letters *p*, *r*, *a*, or *e*, indicating privileges, roles, appointments and environmental predicates respectively. These parameters can refer to a rule variable, or they can use a constant or a function return value (i.e. refer to another container that wraps around a function). The privileges that control the creation of such parameter values are listed in Table 7.9.

Name	*Parameters*
`term_const_new`	`(t term_id, t type_id, value string, p policy_id)`
`term_func_new`	`(t term_id, f cont_func_id, p policy_id)`
`term_var_new`	`(t term_id, t type_id, p policy_id)`
`term_del`	`(t term_id, p policy_id)`

Table 7.9: Privileges for accessing terms.

Privileges associated with adding or removing such containers to or from rules, together with privileges for creating, deleting, binding, and unbinding rules, are specified in Table 7.10.

We next show an example where we assign a permission to a role *my_role* to use a specific privilege as target privilege when creating an authorisation rule. The privilege we allow to be used is *read_EHR(z, f)*, which grants permission to read a field *f* in the electronic health record of a patient *z*. We also want to limit the field which can be read to a constant value *'haematology field'*. In other words, we want to restrict the privilege to $read\_EHR(z,\ \text{'}haematology\ field\text{'})$. This would allow the role that has the above privilege to create rules like:

$$...\vdash read\_EHR(z,\ \text{'}haematology\ field\text{'})$$

Name	*Parameters*
`arule_{new/del}`	`(r arule_id, p policy_id)`
`arule_var_{new/del}`	`(r arule_id, v term_id, p policy_id)`
`arule_priv_{new/del}`	`(r arule_id, p contp_id, p policy_id)`
`arule_prole_{new/del}`	`(r arule_id, r contr_id, p policy_id)`
`arule_papp_{new/del}`	`(r arule_id, a conta_id, p policy_id)`
`arule_penv_{new/del}`	`(r arule_id, p conte_id, p policy_id)`
`arule_bind`	`(r arule_id, p policy_id)`
`arule_unbind`	`(r arule_id, p policy_id)`

Table 7.10: Privileges for accessing authorisation rules.

Assuming that the role has all the other necessary privileges to create an authorisation rule and it can also use the necessary data types, we need to add a rule that gives permission to *my_role* to create an authorisation rule to use the privilege $read\_EHR(z,\ \text{'}haematology\ field\text{'})$:

$$\begin{aligned} &my\_role, \\ &contp\_pname(x, y?), \\ &y = \text{'}read\_EHR\text{'}, \\ contp\_getparam(x, 2) = {} &\text{'}haematology\ field\text{'} \ \vdash \ cont\_priv\_new(x?) \end{aligned}$$

In this rule we use $contp\_pname(x, y?)$ and $contp\_getparam(x, n)$ predicates that return the predicate name and the n^{th} parameter of a predicate container respectively. (The return type of $contp\_getparam(x, n)$ is a *term*, and as the = compares two terms, the string '*haematology field*' is transformed into a term which contains a string.)

We can control the kind of prerequisites a role is allowed to use through the privileges for prerequisite containers.

For example, to allow a role to use *secretrole* as a prerequisite we can use the following rule:

$$my\_role,\ x = \ \text{'}secretrole\text{'} \vdash \ contr\_new(x?)$$

Role activation rules

The containers used by activation rules differ from those used by authorisation rules, since activation containers include information about membership condition. This must be reflected by the privileges, thus in Table 7.11 we define separate privileges for role activation rule containers. Note that the privileges related to role containers (specified in Table 7.8) are also used for specifying access restrictions for activation rules. The reason for this is that the target role of activation roles is embedded into a simple (i.e. not active) container.

With the help of role activation rule privileges we can control access to the prerequisites and the target role of activation rules. These privileges, as in the case of authorisations rules, can express rights to add or delete activation rules, add or delete prerequisites to activation rules, and so forth.

Note that the privileges that manage restrictions on what prerequisites can be used, and with what parameters, are those related to containers.

Name	*Parameters*
`acont_role_new`	`(c acontr_id, r role_id, r arule_id, p policy_id)`
`acont_app_new`	`(c aconta_id, a appointment_id, r arule_id, p policy_id)`
`acont_env_new`	`(c aconte_id, e env_id, r arule_id, p policy_id)`
`acont_del`	`(c {priv, role, app, env}_id, r arule_id, p policy_id)`
`acont{rae}_param_new`	`(c acont{rae}_id, t term_id, io boolean, r arule_id, p policy_id)`
`acont{rae}_param_del`	`(c acont{rae}_id, nr int, r arule_id, p policy_id)`

Table 7.11: Privileges for accessing authorisation rule containers.

7.1.4 Example policy access

In the following example we show the privileges that are requested while building a new policy. This sequence of privilege requests closely follows the methods that were used to create the policy in question.

The policy we shall construct consists of one rule that has three prerequisites (one of each type) with each prerequisite having a single parameter.

The rule is:

$$local\_user(h\_id?),\ employed\_medic(h\_id?),\ on\_duty(h\_id) \vdash doctor\_on\_duty(h\_id)$$

The first prerequisite *local_user*(*h_id*?) is a prerequisite role, the *employed_medic*(*h_id*?) is a prerequisite appointment certificate, and the last prerequisite *on_duty*(*h_id*) is an environmental predicate.

The sequence of privileges requested when this policy was created is shown in Figure 7.2.

We shall next go through the method calls that resulted in these privilege requests.

In line 1 a new data type *local_uid* was created, and later bound to a data type known to the policy engine. This step was needed because the data type had not yet been defined in the policy.

In line 13 a new role was created with one parameter (*h_id* of type *local_uid*, line 14). This role was bound – i.e. made available for other policy components – in line 15.

Similarly to the role definition, in lines 17 and 18 an appointment was created and bound in line 19. Lines 21 to 23 reflect privileges used to specify an environmental predicate with one parameter.

Like the prerequisites, the target role, together with its single parameter, was created and bound in lines 25 to 27.

Lines 29 through 31 created an empty rule (identified by *'erule1'*) with one variable (identified by *'var_x'*). This variable is specific to this rule, and can only be used in the target role and the prerequisites for this rule.

The privileges that were requested when the rule components were embedded into containers are shown in lines 33 to 40.

```
 1  type_new(‘local_uid’, ‘xsd:integer’, policy1)
 2  type_bind(‘local_uid’, policy1)
 3
 4  func_new        (‘local_uid:tostring’, ‘xsd:Integer.toString’, policy1)
 5  func_retval_new (‘local_uid:tostring’, ‘string’, policy1)
 6  func_param_new  (‘local_uid:tostring’, ‘x’, ‘local_uid’, policy1)
 7
 8  func_new        (‘local_uid:parse’, ‘xsd:Integer.parse’)
 9  func_retval_new (‘local_uid:parse’, ‘local_uid’, policy1)
10  func_param_new  (‘local_uid:parse’, ‘value’, ‘string’, policy1)
11
12
13  role_new       (‘local_user’, policy1)
14  role_param_new (‘local_user’, ‘h_id’, ‘local_uid’, policy1)
15  role_bind      (‘local_user’, policy1)
16
17  app_new       (‘employed_medic’, ‘app://...’, policy1)
18  app_param_new (‘employed_medic’, ‘h_id’, ‘local_uid’, policy1)
19  app_bind      (‘employed_medic’, policy1)
20
21  env_new       (‘on_duty’, ‘oasis:...:indb’, policy1)
22  env_param_new (‘on_duty’, ‘h_id’, ‘local_uid’, ‘true’, policy1)
23  env_bind      (‘on_duty’, policy1)
24
25  role_new      (‘doctor_on_duty’, policy1)
26  role_param_new(‘doctor_on_duty’, ‘h_id’, ’local_uid’, policy1)
27  role_bind     (‘doctor_on_duty’, policy1)
28
29  erule_new    (‘erule1’, policy1)
30  term_var_new (‘var_x’, ‘local_uid’)
31  erule_var_new(‘erule1’, ‘var_x’)
32
33  cont_role_new      (‘rc1’, ‘local_user’, policy1)
34  cont_role_param_new(‘rc1’, ‘var_x’, OUT, policy1)
35  cont_app_new       (‘ac1’, ‘employed_medic’, policy1)
36  cont_app_param_new (‘ac1’, ‘var_x’, OUT, policy1)
37  cont_env_new       (‘pc1’, ‘on_duty’, policy1)
38  cont_env_param_new (‘pc1’, ‘var_x’, IN, policy1)
39  cont_role_new      (‘rc2’, ‘doctor_on_duty’, policy1)
40  cont_role_param_new(‘rc2’, ‘var_x’, IN, policy1)
41
42  erule_target_new(‘erule1’, ‘rc2’, policy1)
43  erule_prole_new (‘erule1’, ‘rc1’, policy1)
44  erule_papp_new  (‘erule1’, ‘ac1’, policy1)
45  erule_penv_new  (‘erule1’, ‘pc1’, policy1)
46  erule_bind      (‘erule1’, policy1)
```

Figure 7.2: Example privilege request sequence.

Once the containers had been filled they became ready to be added to the rule created in line 29. This can be followed in lines 42 to 45.

Finally, the rule, now ready, was made available for the policy through binding (line 46).

This example illustrates the complexity of the privileges involved in access control specifications for policies. Policy maintainers cannot be expected to specify privileges for every possible policy access. However, the use of wild-cards (free-variables in authorisation rules) significantly simplifies this task. With their help we can specify rules that can allow generic policy modification. For example, by specifying an authorisation rule in which the target privilege has the form *role_new(x, policy1)* we can allow the principal that is active in the target role of this rule to create any new role in *policy1*. This single rule thus would cover the privilege requests in lines 13 and 25.

The privileges we specified can be used by administrators directly, but they can also be used in applications that provide a GUI to policies. As the privileges closely follow the policy structure, errors or access denials can be easily managed, and proper feedback can be given to the users.

7.1.5 Component visibility

In general, it makes little sense to restrict read access to individual policy components like roles or data types. We believe that read access to policies must be handled at policy level. The reason for this is consistency. For example, adding a role to a policy without knowing all the other roles in the policy could lead to problems like name conflicts, in which case the uniqueness of role identifiers within a policy is not ensured.

7.2 Extending with contexts

We have seen how to specify policies that allow general policy modification by using free variables in authorisation rules. With the help of such rules we can authorise a role to modify any role; however, it is desirable to specify access control to a specific portion of a policy. In Chapter 4 we introduced contexts, which can form policy component groups. These groups can be referred to through the context element that groups the components together. We therefore extend our privileges to contain context information. This extension adds a parameter to the privileges. Thus, for example, instead of the

```
priv_new   ( p priv_id, p policy_id )
```

privilege, we shall have a privilege like:

```
priv_new   ( p priv_id, p policy_id, c context)
```

Whenever a policy component is accessed, a privilege with the component's context is requested. If the principal can acquire the privileges in question, then it may perform the requested operation. If there is more than one context element present in a context, then the principal must have access to components belonging to each of these context elements. For example, if a role (*a_role*) is marked with `[A,B]` as its context, and a principal would like to delete this role, then it must have rights to the following privilege:

$$priv\_del(\text{`}a\_role\text{'},\ [\mathbf{A},\mathbf{B}])$$

Privileges having a compound context parameter can be decomposed into privileges that have contexts consisting of a single context element, thus the above privilege could be decomposed into the following two privileges:

$$priv\_del(\text{`}a\_role\text{'},\ [\mathbf{A}])$$
$$priv\_del(\text{`}a\_role\text{'},\ [\mathbf{B}])$$

A principal that intends to delete the role *a_role* thus must have rights to both of the above privileges.

7.2.1 Context modifications

The context of a new component must be specified when the component is created. Context information must thus be known at the time the relevant privilege is requested. However, a component's contexts may change. To control this we introduce privileges that allow the addition and removal of context elements.

These are in the form of $\star$*_context_add* and $\star$*_context_del*, where $\star$ refers to the component type. These privileges contain a parameter that can restrict the component whose context may be modified. These privileges also include the context element that is to be added to or deleted from a component's context.

The list of privileges that includes these context extensions is included in Appendix B.

7.3 Access to meta-policies

Conformance with compliance and interface policies requires some information, such as mapping information. This, together with a certificate indicating a successful compliance check, is policy specific, thus it makes sense to store such compliance information together with the relevant policies. Who may add or delete such compliances is controlled via a set of privileges. These privileges contain an identifier that refers to the compliance policy or interface policy, and their versions.

Similarly, the specification of new context labels is controlled via a privilege. These are listed in Table 7.12.

Name	*Parameters*
`context_mod`	`(i context_policy_id, p policy_id)`
`compliance_{new/del}`	`(c compliance_policy_id, p policy_id)`
`interface_{new/del}`	`(c interface_policy_id, p policy_id)`

Table 7.12: Privileges for accessing authorisation rules.

Note that the granularity we provide to access meta-policies is more coarse-grained than what we allow for policies. For example, in the *interface_new* privilege we do not include parameters that could restrict the set of policy components that could be used for an interface policy compliance check.

The reason for providing less control over meta-policy related components is that we expect meta-policies to be more persistent and simpler then policies; we also assume that meta-policies are administered by fewer people, all of whom can take a high-level view.

7.4 Discussion

An access control policy is a set of rules and the component definitions for these rules. The privileges presented in this chapter are not different from the privileges that are used in other access control policies, thus they can be part of any parameterised RBAC policy. However, they have a special meaning to the service that is responsible for policy storage and access control to it. Therefore, a policy may include the above privileges in order to control access to itself. Roles that can control part of a policy that defines them could potentially authorise themselves to control additional parts of a policy. This is clearly a very dangerous way to administer policy, but our meta-policies can help to achieve control. For example, all the administrative privileges can be assigned a context element like *desert.admin*. This not only helps to separate administrative privileges, but through information flow restrictions it can constrain the set of roles that can be assigned administrative privileges. Our meta-policies also help to maintain control over policy changes, and we shall discuss this in the following chapter.

7.4.1 Related work

We next review the work done by other researchers in the area of policy administration. Unfortunately, since most of the research in the RBAC community focused on model extensions (such as delegation, constraints, and hierarchies) administration has received little attention. The ARBAC model, defined by Sandhu et al. is the most significant work in this field. Since its initial introduction in [SBC$^+$97] ARBAC has undergone several changes. The original is referred to as ARBAC97 consists of three components: URA97, PRA97, and RRA97. These acronyms stand for user-role assignment, permission-role assignment, and role-role assignment respectively. This is the first work that considers RBAC policies from the resource perspective, and proposes the use of RBAC itself to control access to RBAC policies.

ARBAC97 introduces administrative roles, and defines a small set of parameterised privileges that control the way administrative roles may modify user-role, permission-role, and role-role relations. Note that the policies whose access control can be managed through these privileges use the NIST model; accordingly, these privileges do not support role parameters!

ARBAC97 introduces role ranges, that help to restrict authority in a role hierarchy. This points to a problem in distributed policy administration – namely policy component grouping, or in other words scoping.

In [Cra02], Crampton and Loizou address the scoping problem of ARBAC, however their work concentrates mainly on role hierarchies, and how portions of this hierarchy could be identified. Our solution to such scoping utilises contexts, which provide a more flexible means to form policy component groups.

Using RBAC to manage policies can lead to other problems as well, one of which is discussed by Schaad and Moffett in [SM02b]. They use the Alloy language to analyse ARBAC policies in order to detect separation of duty conflicts. We address the same problem through our compliance policies, which check for policy properties before policies are permitted to become active. We shall discuss this, together with the stages at which our policies are checked, in the next chapter.

Many problems of ARBAC97 were realised by the authors; consequently, they introduced extensions to their model. One of their revised models, ARBAC02, is presented in [OS02]. It adds the concept of organisational units to address dependency problems arising from the use of hierarchies. However, this model still lacks support for flexible scoping.

Our privileges were designed to work with a more general RBAC model, OASIS. As a consequence these privileges are more complex, and their number is also much larger. However they support parameters, more general prerequisites, and more complex rules. In our policy management framework we are able to enforce general policy restrictions, and with the help of our contexts we can group policy components in a flexible way. Unlike in ARBAC, roles are not separated into two distinct categories (administrative and non-administrative roles), and we can use the same roles that are specified by a policy to control the policy itself. Note, however, that such separation can easily be achieved through contexts; indeed, through contexts we can even organise the administrative policy components into further subgroups.

In [KSM03] Kern et al. describe another approach to using RBAC for self-administration.

Primary concepts of their model, called the Enterprise Role-Based Access Control Model (ERBAC), are enterprise roles and scopes. These enterprise roles are general roles that span over more than one target system. Their concept is similar to the abstract roles we have in both interface and compliance policies. Scopes in ERBAC provide flexible component grouping, similarly to our contexts. Unfortunately the ERBAC model lacks support for parameters.

7.4.2 Future work

Our privilege specification for policy access opens plenty of opportunities for further work.

We have specified privileges only. Since these privileges are fine-grained, to give a role rights to create a role, modify it, delete it, and manage its parameters, we must use many privileges. However, as some of these privileges occur together in policies, it could be desirable to provide more support for their grouping. This could be done with the help of roles. For example, we could specify a *role_modifier* role that would encompass all of the above mentioned role modification privileges. This role could be defined as follows:

$$role\_modifier(r\ role\_id, pol\ policy\_id, c\ context)$$

$$\begin{aligned}
&role\_modifier(r,p,c) \vdash role\_new(r,p,c)\\
&role\_modifier(r,p,c) \vdash role\_del(r,p,c)\\
&role\_modifier(r,p,c) \vdash role\_param\_new(r,x,y,p,c)\\
&role\_modifier(r,p,c) \vdash role\_param\_del(r,x,y,p,c)\\
&role\_modifier(r,p,c) \vdash role\_bind(r,p,c)\\
&role\_modifier(r,p,c) \vdash role\_unbind(r,p,c)
\end{aligned}$$

Policy specification would be simplified if such roles were already available.

If privilege inheritance were supported, then these aggregating roles could be expressed through the privilege hierarchy, making it unnecessary to use such functional roles, and thus avoiding their misuse (e.g. using these roles as preconditions in role activation rules).

In our work we have not described fine-grained access control to contexts. It could be interesting to explore context management, since the hierarchical structure of context elements could be exploited in environments where policy management follows a hierarchical domain structure. For example, in a university a computing service may be responsible for the top level context elements, and departments may have authority over their context elements, which are sub-elements of the university's context element.

7.4.3 Summary

In this chapter we have introduced a set of privileges for managing policies that consist of the policy components defined in this thesis. Our motivation for doing so arises from the usual setup of policy administration, i.e. there are more and less competent administrators, who share the task of policy administration. By providing means to specify restrictions on how policies may be modified, more competent administrators can provide their colleagues with a playground or sandbox environment, thus saving time and separating high and low-level administration tasks. This is vital in case of a large number of administrators, and when policy administration is distributed. In this chapter we have also given examples that demonstrate the fine-grained control that our privileges can achieve.

Further, by allowing our administrative privileges to contain context parameters, we provide a powerful means to specify access control to groups of policy components. This extension complements the method of using free variables in authorisation rules to provide general policy modification privileges. Additionally, through context parameters we introduce an indirection between policy administration privileges and policy components, extending the applicability of the administrative privileges to successive policy versions. This latter property is vital for policy management and policy evolution, both of which we shall discuss in the next chapter.

8 Policy administration

In this chapter we consider policy administration, and we describe how meta-policies and policy administration privileges can be used to support policy evolution. As part of this description we look at the various steps involved in such evolutionary processes, and indicate how these steps affect each other. Since policies change it is important to keep track of their successive versions. Thus, in this chapter we also describe the version information our policies store, and indicate how such information can be used to support graceful policy change. Finally, we describe Desert, our policy administration framework.

8.1 Policy evolution

Much of the work we have presented in this thesis is motivated by the vision of a distributed policy store, which controls access, facilitates maintenance, and ensures the consistency of the policies it stores.

We have presented a layered architecture for such policy stores in [BEWM03]. Its three constituent layers are as follows:

Policy management: This layer concerns consistency and restrictions at policy level. At this level we can find restrictions such as those specified by our meta-policies. This level is also responsible for managing policy version information, which we describe in Section 8.2.

Policy component management: The middle layer of our architecture uses an API, provided by the lowest layer, to create, delete, and modify policy components. Its purpose is to ensure component-level consistency, and to enforce component-level access control.

Active predicate management: This is the lowest level of our proposed architecture. It uses the API of the underlying storage architecture, which in our case is the `t5` predicate store we developed to support active predicate storage in the PostgreSQL [Loc01] RDBMS (see [BEWM03]). The purpose of this level is to store policy components in a distributed manner, while ensuring component level integrity.

8.1.1 Policy changes

There are many consistency and access control requirements that need to be satisfied during policy modification. The expected sequence of actions and checks performed during policy change is illustrated in Figure 8.1.

The policy store is responsible for enforcing access control to the policies that it stores. If a principal initiates a policy modification, the policy store first checks whether the principal is

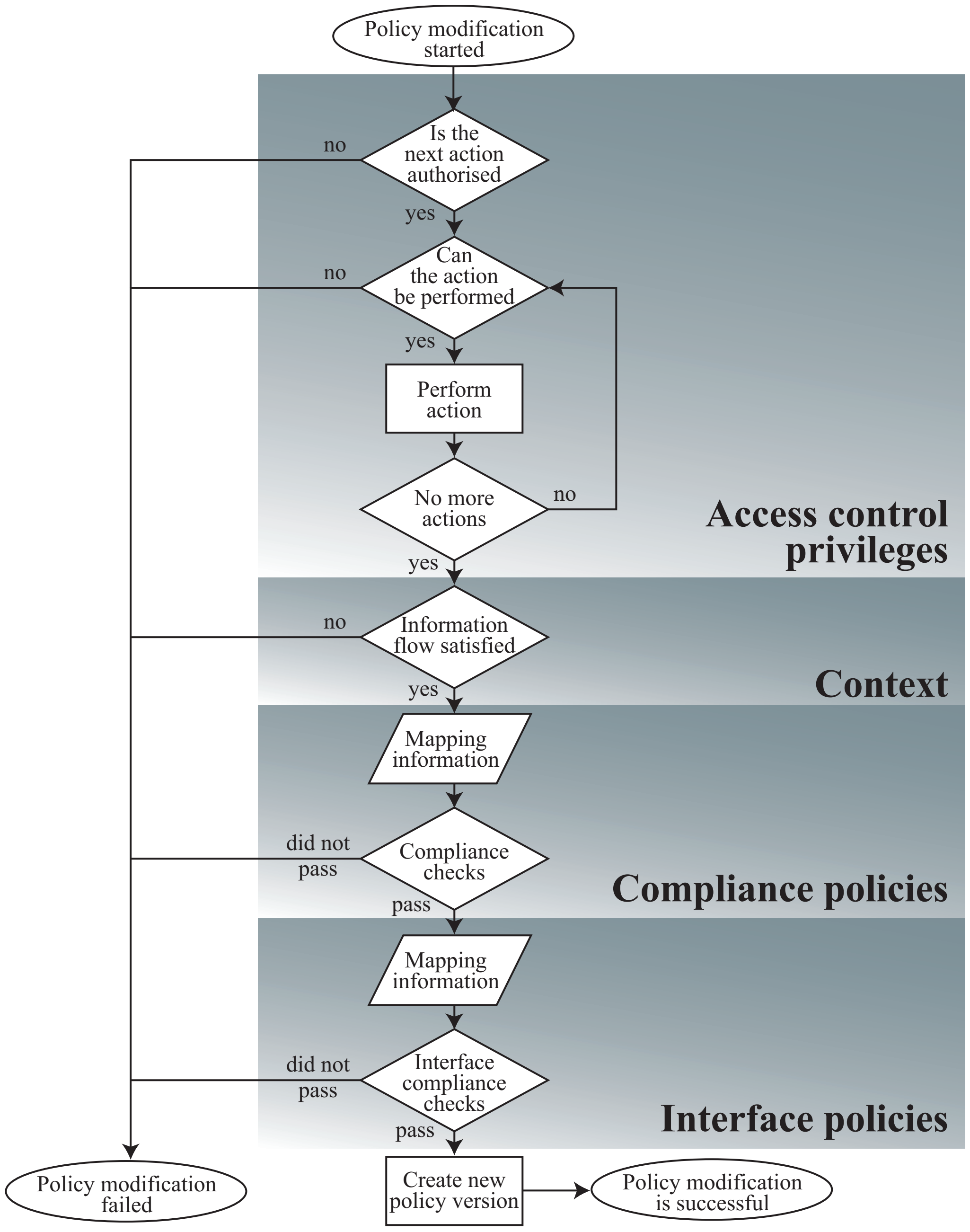

Policy modification started
Is the next action authorised
no
yes
Can the action be performed
no
yes
Perform action
No more actions
no
yes
Access control privileges
Information flow satisfied
no
yes
Context
Mapping information
Compliance checks
did not pass
pass
Compliance policies
Mapping information
Interface compliance checks
did not pass
pass
Interface policies
Policy modification failed
Create new policy version
Policy modification is successful

Figure 8.1: A policy evolution step.

authorised to perform the requested actions, i.e. the policy store checks whether the principal holds the privileges required for the requested operation (as discussed in Chapter 7). The requested policy modification is performed only if the principal is authorised, and the modification can be performed (i.e. it satisfies state-related restrictions, such as 'a component may only be deleted if it is in unbound state' (see Section 7.1.1)).

Policy modifications do not necessarily have to be made by a single principal. A series of modifications could be made in the frame of a transaction, in which many principals may modify a policy. The way in which such modifications are grouped is not part of our work.

Once all the modifications are completed we must check whether there is any policy component left in an unbound state. We require that, at the end of the policy modifications, all undeleted policy components are bound.

In the next step the restrictions specified by meta-policies are checked: first contexts, then compliance and interface policies. Note that for many of these checks, extra information, such as mappings, are required.

Once all of the meta-policy tests are successfully finished, the new policy version is ready to replace the current policy.

Meta-policies, just like policies, may change over time. A major difference however, is the expected frequency of changes. We illustrate this in Figure 8.2.

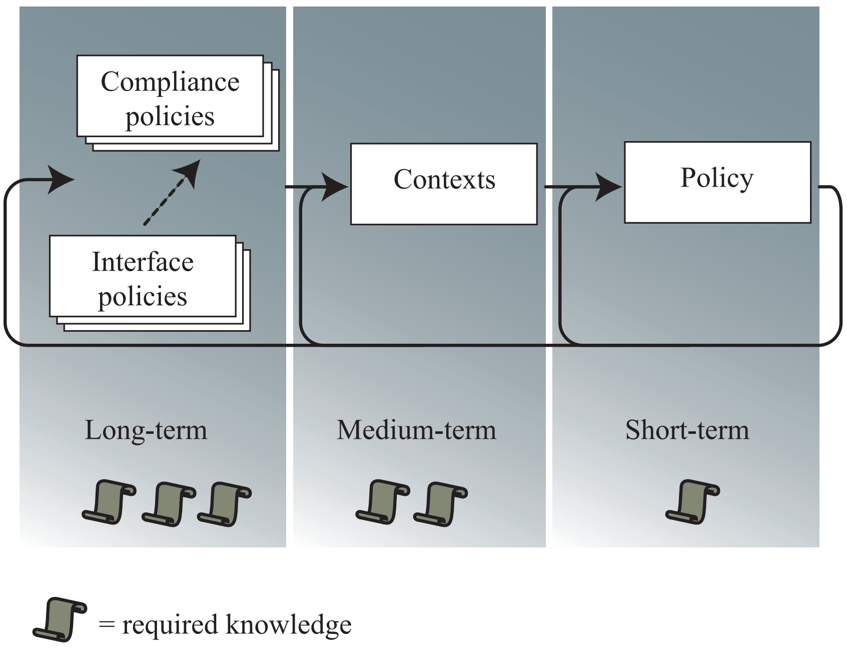


Figure 8.2: Policy and meta-policy lifetime.

Since meta-policies encapsulate long-term design decisions, we expect that changes will occur most frequently at policy level.

There is a dependency relationship between policies and the meta-policies they comply with. If a policy changes it needs to maintain compliance with its meta-policies (unless compliance is no longer required). If a meta-policy is modified, then all the dependent policies and meta-policies need to be updated to reflect the requirements set by the new meta-policies. Since meta-policies can be used to specify restrictions to many policies, a change to a meta-policy is relatively expensive.

Meta-policies specify constraints over policies. This allows competent policy administrators with a view of higher-level requirements to create a sandbox environment for policy administrators with more limited jurisdiction. In such a sandbox a policy administrator may modify only a restricted set of policy components (this restriction may be governed by contexts). The way they may modify the accessible policy components can also be restricted through the policy management privileges. Furthermore, when modifying a policy, administrators must follow the general guidelines expressed by compliance and interface policies.

In environments such as the National Health Service, our compliance policies can help local policy administrators to follow general NHS guidelines. Compliance policies are particularly useful when new initial policies are set up, since compliance policies can be used as templates. In the case of a new hospital, policies based on compliance policies and interface policies that serve to enable inter-hospital cooperation can be used as initial hospital policies. Such synthesised policies can be later extended according to local needs, but since they closely follow the structure of meta-policies, mappings for both compliance checks and inter-hospital cooperation will be easy to specify.

Meta-policies are expected to be maintained by specialised administrators who have a high-level view. Using such meta-policies for guidance, administrators with less training and experience can create and modify local policies.

Through this task sharing, our sandboxes may help to reduce overall policy administration training costs in large organisations.

8.2 Version control

When a series of authorised policy modifications is completed, and the resulting policy satisfies all the required restrictions given in the form of contexts, compliance policies, and interface policies, the new policy may replace its predecessor. Technically this means the assignment of a new version number to the new policy, after which the previous policy is replaced with the new one.

However, many components of our access control architecture are dependent on policies and their particular versions:

Currently active sessions: Sessions that are active at the time of policy version change may have roles of the previous policy version active.

Active service level agreements: A special case of the above problem is when a session is active, and some roles belonging to the previous policy version are used remotely, and the remote role is marked as a membership condition (i.e. an event channel has been set up and the validity of the role is being monitored).

Environmental predicates: Environmental predicates can depend on external services, and modification to a policy may change such dependencies. Furthermore, similarly to the first problem, an environmental predicate may be a membership condition for policy components of an active session, in which case such external links must also be handled during policy evolution.

Appointment certificates: Appointment certificates, while designed to serve as long-term credentials, may contain arbitrary prerequisites (see [BMY02]). Such prerequisites refer either to other appointment certificates, or to a policy. For example, an appointment certificate, i.e. an appointment instance, may require the principal intending to use the appointment certificate to be active in a specific role. This role is part of a policy specification, which may change in time. We must ensure that the role that has the same name and parameter signature in the new version of a policy is compatible with the one that is referred to in the appointment certificate.

There are many ways to address these problems. A simplistic solution would be to abort all sessions that are active at the time of policy update. While this solution is simple to implement, in the real world a more graceful policy change mechanism is desired. The other extreme is supporting full version management for policies, in which we may have branching policy versions. At the moment we do not feel that such a heavy-weight version control mechanism is justified.

The solution we propose lies between the above two alternatives. We do not allow version branching; accordingly, our version graph will be a list. Whenever a new policy version is introduced we must provide compability information. This specifies whether a policy component is compatible with its previous version.

We allow compatibility only between rule components and data types that have the same name and parameter signature. We do not keep version information about rules. In Figure 8.3 we illustrate some example version compatibility information. In the figure we indicate roles that are incompatible with their last (current) version by putting them into square brackets.

	Version 1	**Version 2**	**Version 3**
roleA:	$[roleA(x\ int)]$	$roleA(x\ int, y\ int)$	$roleA(x\ int, y\ int)$
roleB:	$[roleB(x\ int)]$	$[roleB(x\ int, y\ int)]$	$roleB(x\ int, y\ int)$

Figure 8.3: Example version compatibility information.

In this example there are three successive versions of two roles (*roleA* and *roleB*). The first two versions of *roleA* cannot be declared compatible, since there is a change to its parameter signature. However, the second and third versions of roleA can be, and are specified to be, compatible (since their name and parameter signatures are the same). The second role, *roleB*, is structurally the same as the *roleA*, however we declared its current, third version as incompatible with previous versions, even though its structure would allow compatibility.

If a component is not compatible with its previous version, then all its older versions active in a session must be declared invalid when the new policy takes effect. Such invalidation implies the revocation of the non-compatible roles, and all policy components that are dependant on these components through membership conditions. Compatible components on the other hand may persist after a policy change. Such handling of compability information enables administrators to perform policy modification in a graceful way.

It is very expensive and dangerous to revoke roles that are not membership conditions. Users might be performing important operations that a role revocation might endanger. Whether an incompatible role should be revoked or not depends much on how critical or important the role is. Our contexts could help to specify such revocation behaviour.

The entire policy version may be marked as incompatible with its previous version. In this case administrators can ensure that all the sessions active at the time of version change are aborted or phased out gradually.

Policy components that monitor membership conditions for remote policy enforcers handle component revocation the same way as in the case of normal revocation, i.e. they notify the remote domains about the revocation. Whether a new version of the event channel needs to be set up depends on the underlying messaging infrastructure.

During a sequence of policy changes the active policy must maintain a version number for every policy component, and accept policy components that have a version number between this stored number and the current policy version. This number can be compared against the policy version number of specific appointment certificates.

8.2.1 Auditing

Audit trails are vital in access control policy enforcement. Since we keep track of policies and the versions that are used to authorise a principal to perform an action, we can find the policy version that was in force at any time for audit purposes.

8.3 Implementation - Desert

To test the ideas introduced in this thesis we have developed a proof-of-concept policy management framework, DESERT.

Its development constitutes part of a larger set of projects in which members of the OPERA research group are involved. In the following we briefly describe the foci of these projects:

Event middleware: This project focuses on publish/subscribe event middleware for large-scale distributed systems. Among the results of this project is Hermes [PB02], a novel type- and attribute-based publish/subscribe architecture.

Ongoing research in this area involves efficient composite event detection ([PS02]) and security ([BEP$^+$03]). Security is of particular interest to us, since, as described in [BEP$^+$03], we use OASIS RBAC to control access to event types, event subscription, advertisement, and publication. Note that the enforcement of this access control to events is handled by OASIS, which itself relies on event technology (see service level agreements in Section 2.3.2).

SECURE: The SECURE (Secure Environments for Collaboration among Ubiquitous Roaming Entities) project is investigating the use of trust in global computing. Results of their work could be incorporated into our interface policies as described in Section 6.4.1. On the other hand, the SECURE project can benefit from using our compliance policies – and policies conforming to them – as an information source for establishing trust.

OASIS: This project maintains the OASIS access control model and its many implementations to reflect mainstream technologies such as X.509 certificates, web services, SOAP, J2EE, and XML.

Policy store and policy management: The primary aim of this project is to design and implement a distributed policy store that controls access to its policies and supports policy evolution. The work in this thesis is part of this initiative.

The main components of the DESERT policy management framework are shown in Figure 8.4.

These components are as follows:

Policy component interfaces introduce an indirection level between stored policies and policies used by OASIS or some other RBAC policy enforcer.

An implementation of the policy component interfaces adds a protection layer to the policy and its components. It is also responsible for ensuring component, policy, and context-level consistency. Furthermore, this layer uses a set of privileges that control policy component access.

A compliance checker, which tests conformance to compliance and interface policies.

The SLA generator creates service level agreements between policies based on the two policies, an interface policy, and the relevant mappings.

Visualisation tools. Part of the DESERT framework is a set of visualisation classes that help to display or manage policy components, meta-policies and additional information, such as mappings.

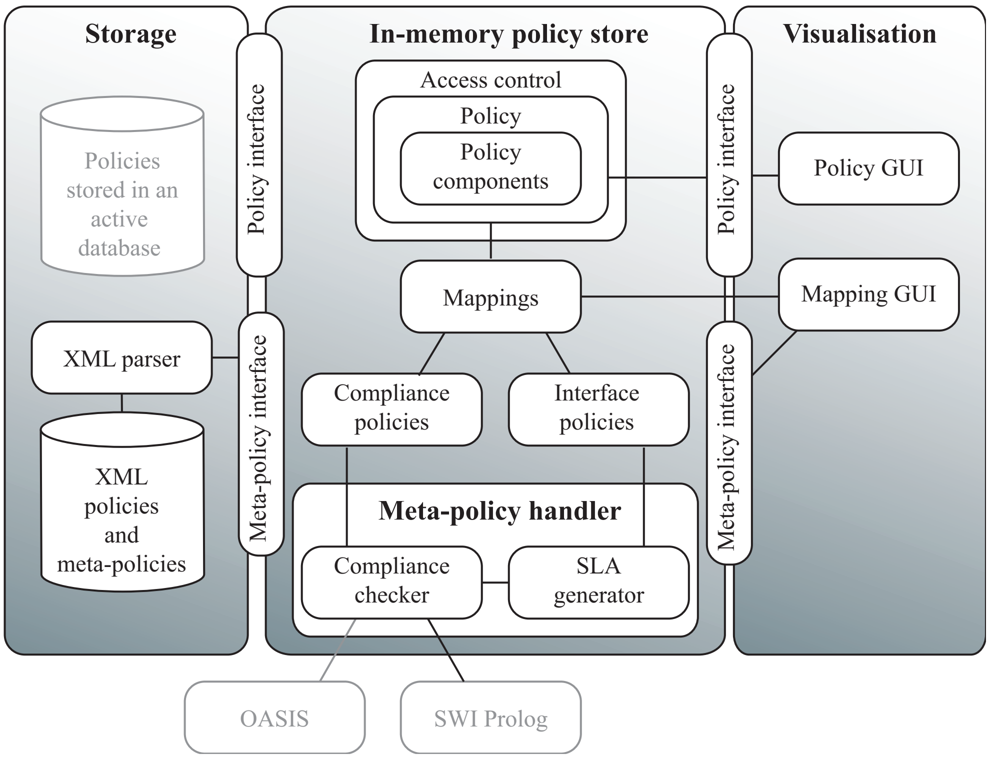


Figure 8.4: The DESERT architecture.

External components. DESERT makes use of external components, such as OASIS, Prolog, and a policy store. These are not part of our framework (in the figure this is indicated by grey colour). Interaction with such external components is handled via Java interfaces.

In the following we shall look at these components in more detail.

8.3.1 Policy component interfaces

At the time we started to implement DESERT there were two ongoing OASIS developments. In order to support each of them, and to facilitate later development, we defined a set of Java interfaces to policies and to policy components. These specify the API through which policies and policy components should be accessed both within and outside the DESERT framework.

These interfaces allow us to create, modify, and delete policy components independently of the underlying implementation.

Contexts

Contexts are more policy-specific than interface and compliance policies (see Section 3.2). They can either be integrated into a policy, or stored separately together with some binding information. For the internal policy representation we have chosen the first option, thus the methods specified in our interfaces contain context handling parameters.

Meta-policy interfaces

Compliance and interface policies are expected to be stored in a similar fashion as our access control policies, thus we provide a set of interfaces that manage the creation and maintenance of meta-policies.

8.3.2 Component implementation

We implement the above interfaces in order to have an in-memory representation of a policy and its components. The general structure of these classes is the same as the structure we presented in Chapter 3. Note that all the implementing classes use interfaces (some of which are marker interfaces) to access policy components. This allows any component implementation to be swapped without requiring changes to the rest of the source code.

In accordance with the interface specification, these policy components are able to store context information, and can check information flow restrictions.

Access control

To support access control to policy components a set of privileges is defined. These use the same component class as other policy privileges (*PrivilegeImpl*), and are instantiated by default for every policy management framework. A full list of these privileges is provided in Appendix B.

In our implementation the policy management privileges are requested whenever a policy or policy component method is invoked. If the principal invoking a method holds the privilege, the method execution will proceed, otherwise an exception is raised. Privilege requests are handled through a special class (*AccessController*). Its implementation is responsible for establishing a connection with an OASIS service, and for forwarding privilege requests. Since at the moment our Desert framework is not bound to any particular OASIS implementation, we provide a temporary implementation for the *AccessController*. This is used mainly for debugging purposes, since it gives manual control over privilege checking, and it can also produce a list of requested privileges (we used this for the example in Section 7.1.4).

XML policy loader

The policy store that uses a distributed active database management system was being developed in parallel with Desert, therefore we needed to find alternative ways to store policies and meta-policies persistently. For this purpose we defined a set of XML schemata (an example schema is included in Appendix C) and provided tools to read XML policies and meta-policies.

Such XML representations of policies and their components can also be used for exchanging policies or parts of them, for example in a service level agreement. The integrity of such XML files can be ensured by using digital signatures.

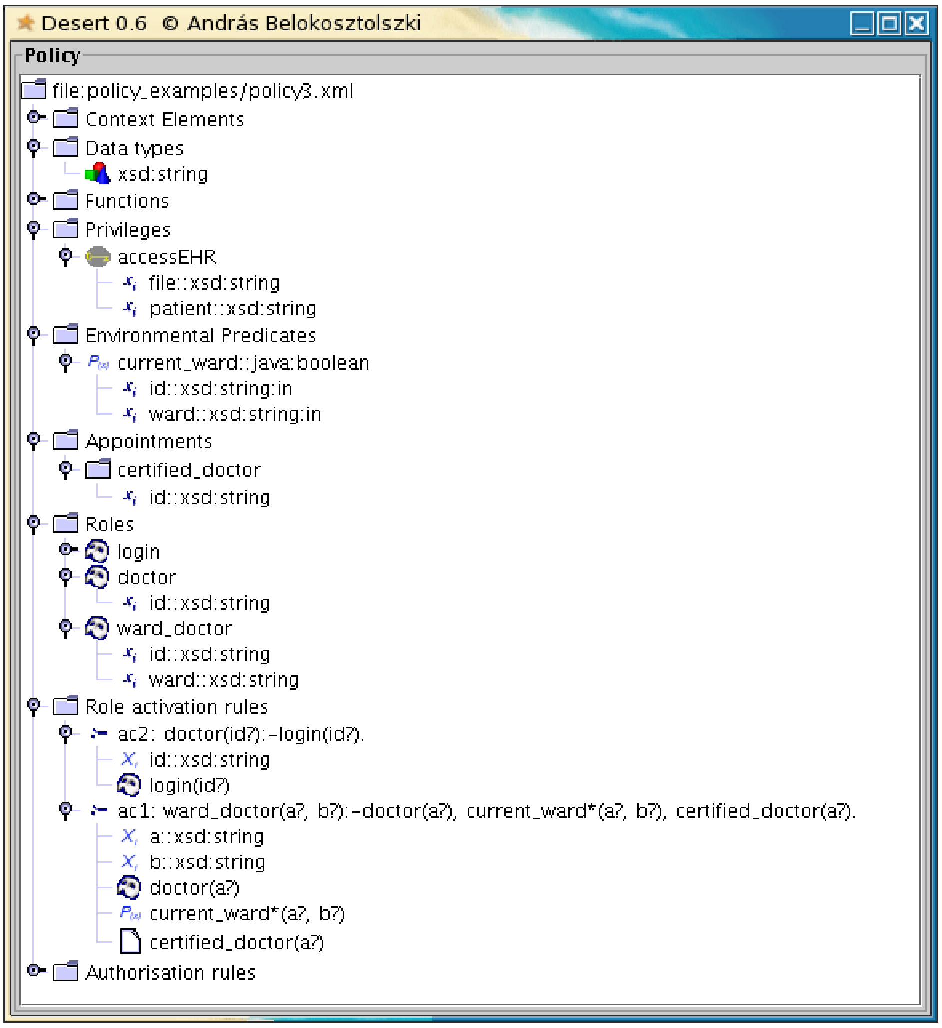
Consistency

The policy component classes, that implement the above interfaces also take care of policy consistency in the same way as described in Section 3.1. These classes also enforce context information flow restrictions.

Visualisation

To visualise policies we provide a set of classes, which use the policy and policy component interfaces to display information about a policy (see Figure 8.5).

These are intended to be used in GUIs that need to display our policies. One such application that uses our policy visualisation classes is a mapping editor, which we shall describe on page 142.



```
Desert 0.6  © András Belokosztolszki
Policy
file:policy_examples/policy3.xml
  Context Elements
  Data types
      xsd:string
  Functions
  Privileges
      accessEHR
          file::xsd:string
          patient::xsd:string
  Environmental Predicates
      current_ward::java:boolean
          id::xsd:string:in
          ward::xsd:string:in
  Appointments
      certified_doctor
          id::xsd:string
  Roles
      login
      doctor
          id::xsd:string
      ward_doctor
          id::xsd:string
          ward::xsd:string
  Role activation rules
      ac2: doctor(id?):-login(id?).
          id::xsd:string
          login(id?)
      ac1: ward_doctor(a?, b?):-doctor(a?), current_ward*(a?, b?), certified_doctor(a?).
          a::xsd:string
          b::xsd:string
          doctor(a?)
          current_ward*(a?, b?)
          certified_doctor(a?)
  Authorisation rules
```

Figure 8.5: Policy visualisation in Desert.

8.3.3 Compliance policy checks

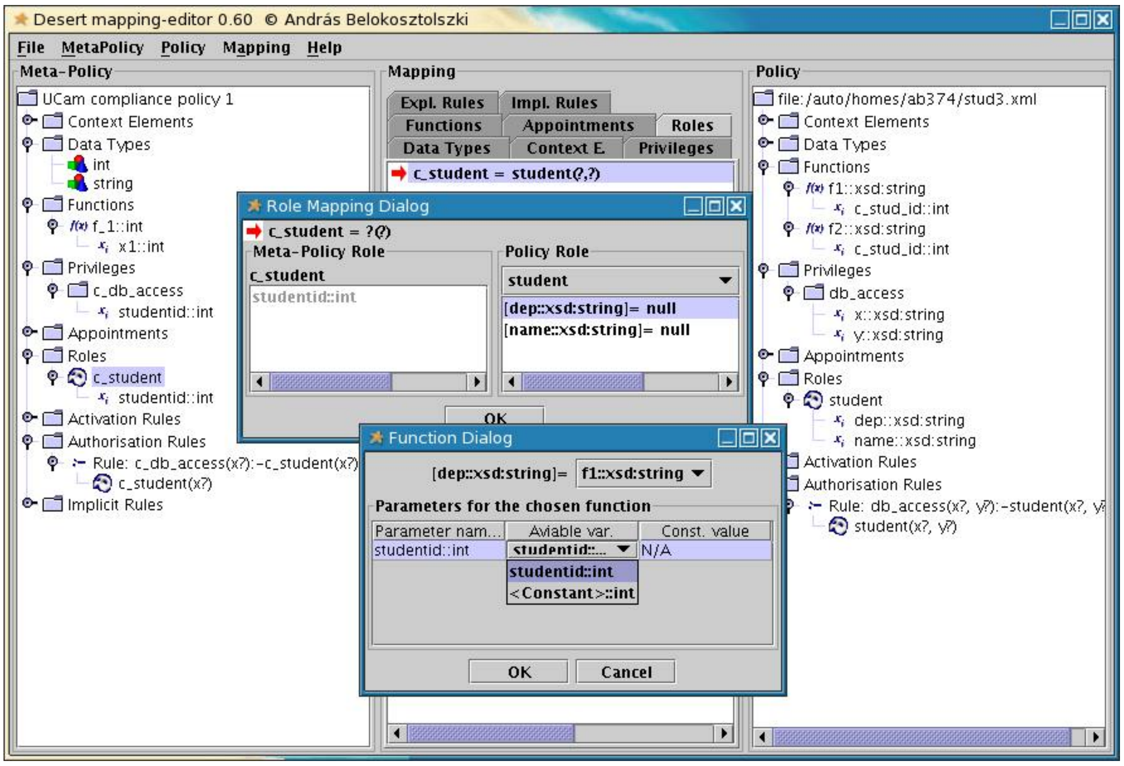
Most of the compliance checks we described in Chapter 5 consist of relatively simple steps that look for specific policy components. The most complex checks involve implicit rules. Since, due to skolemisation, we have no variables that need to be matched, and environmental predicates need not be looked up, there is no need to use OASIS's complex unification engine to perform implicit rule checks. Instead, our implementation uses SWI Prolog to perform compliance tests.

Mapping editor

In order to perform a compliance check, policy administrators need to provide a mapping between a policy and a compliance policy.

In Desert this can be done by manually specifying a mapping in an XML file; however, since such mappings reference many components of both the policy and meta-policy involved, such mapping specifications are likely to be highly error prone.

To simplify the task of mapping we provide a GUI-based editor (see Figure 8.6).


Desert mapping-editor 0.60 © András Belokosztolszki
File MetaPolicy Policy Mapping Help
Meta-Policy
Mapping
Policy
Expl. Rules Impl. Rules
Functions Appointments Roles
Data Types Context E. Privileges
Role Mapping Dialog
Meta-Policy Role
Policy Role
OK
Function Dialog
Parameters for the chosen function
OK Cancel

```
UCam compliance policy 1
  Context Elements
  Data Types
    int
    string
  Functions
    f_1::int
      x1::int
  Privileges
    c_db_access
      studentid::int
  Appointments
  Roles
    c_student
      studentid::int
  Activation Rules
  Authorisation Rules
    Rule: c_db_access(x?):-c_student(x?)
      c_student(x?)
  Implicit Rules
```

c_student = student(?,?)

Role Mapping Dialog: c_student = ?(?) — Meta-Policy Role: c_student, studentid::int — Policy Role: student; [dep::xsd:string]= null; [name::xsd:string]= null

Function Dialog: [dep::xsd:string]= f1::xsd:string

Parameter nam...	Aviable var.	Const. value
studentid::int	studentid::...	N/A
	studentid::int	
	<Constant>::int	

```
file:/auto/homes/ab374/stud3.xml
  Context Elements
  Data Types
  Functions
    f1::xsd:string
      c_stud_id::int
    f2::xsd:string
      c_stud_id::int
  Privileges
    db_access
      x::xsd:string
      y::xsd:string
  Appointments
  Roles
    student
      dep::xsd:string
      name::xsd:string
  Activation Rules
  Authorisation Rules
    Rule: db_access(x?, y?):-student(x?, y
      student(x?, y?)
```

Figure 8.6: DESERT's mapping editor.

8.3.4 Interface policy tools

Our compliance checker also checks for conformance with interface policies. Once two policies (a component exporting and component importing policy) comply with an interface policy, an SLA may be set up between them. This is handled by our SLA generator.

8.4 Discussion

In this chapter we have presented the steps of policy evolution, and discussed how these steps may be constrained with the help of meta-policies. We looked at the expected lifetime of our meta-policies, and considered how these, together with the policy management privileges, could be used to restrict the freedom of some policy administrators.

We have also considered version management issues to enable graceful policy changes, in which compatible policy components of subsequent policy versions may temporarily coexist, making policy changes less noticeable to users.

Finally, we have described our policy management framework, DESERT, which we used to test the ideas presented in this thesis.

We expect DESERT to be part of the final policy management framework that is being developed and maintained by members of the OPERA research group. DESERT's integration should be relatively straightforward, due to its modular design and the interfaces through which its components are accessed. Currently other members of the OPERA group are working on a policy store that is based on a distributed active database management system. Once again, through its interfaces DESERT can make use of such policy stores.

9 Conclusions and future work

With our increasing dependence on computers, requirements for large-scale organisational access control policies will continue to grow. More and more business tasks will be performed relying on infrastructure provided by interconnected computers. Since such business transactions will frequently involve dynamically changing collaborators, it is vital that access control policies adapt rapidly in order to follow organisational needs.

Such requirements demand skilled policy administrators, who are able to change policies to support ad-hoc collaborations, while ensuring that the policies fulfil their fundamental purpose, i.e. they control authorised and unauthorised access.

Dependence on well-designed and well-operating access control policies necessitates tools to structure and to organise policies.

In this thesis we made the initial steps to provide the means for policy administrators to share the task of policy administration in dynamic environments.

In this chapter we first review our contributions, and then consider some possible future research directions.

9.1 Summary of contributions

The primary contribution of our work is a means to organise and structure RBAC policies to support their long-term evolution in an administratively distributed environment. In detail these are the following:

Policy components: We have extended the policy component model of OASIS RBAC in order to support its long-term evolution. Through naming every component in a policy we enable descriptions external to a policy specification to refer to individual access control policy components, that is essential for fine-grained management. We also included constructs in this component model that allow it to explicitly express popular RBAC features, such as role hierarchies.

Meta-policies: We separated organisational policies into parts that directly relate to access control decisions and those that encode the general goals of access control. By doing so we translate organisational policies into a hierarchical access control policy description that encapsulates access control information at different abstraction levels. This allows us to classify information in policies, and use this classification to support policy evolution and distributed policy management.

Contexts: We have introduced contexts, a construct that allows us to structure policy components from multiple aspects. Contexts introduce an indirection between entities that manage policies and the policies themselves, allowing us to refer to certain policy parts independently from their version, naming, and implementation. We have used this indirection to enforce general restrictions on the structure of the policies. Such restrictions include control of information flow and of prerequisite categories.

Compliance policies: We have introduced compliance policies that can express fine-grained restrictions and requirements for policies. Compliance policies can be considered as invariants for successive policy versions. Having their own component model, they are well-suited to describing long-term policy goals for a number of policies, thus helping administrators to maintain certain properties throughout policy evolution.

Interface policies: Building on compliance policies we introduce interface policies to overcome heterogeneity problems in inter-policy collaboration. Our interface policies work as mediators between cooperating policy domains, enabling remote resource access that complies with the organisational policies of the cooperating parties. Furthermore, interface policies can express requirements for the cooperating parties, and conformance to such restrictions can be used as a precondition to collaborations.

Access control to policies: We have introduced a set of privileges that enable OASIS RBAC policies to treat RBAC policies as resources and control access to them. These privileges allow fine-grained policy update security. They may also use contexts, making them better applicable to successive policy versions.

Policy management: We have considered how long-term policy evolution can be supported by our meta-policies and policy management privileges. We discussed version management issues, and how our meta-policies can be used to gradually deploy policies in a top-down manner.

Implementation: Finally, we have described our policy management framework, DESERT. It provides a set of interfaces and their implementation for the policy components we introduced in Chapter 3, and gives practical means to experiment with policies structured in that way, and to check their conformance to meta-policies. It includes visualisation tools that help future policy extensions and development. Also, through the privileges we described in Chapter 7 DESERT can enforce access control to policy modifications, which is vital for its becoming part of a policy store.

9.2 Future directions

Our research can be considered as an initial step in policy administration research. There are many ways to extend our work, some of which we shall discuss next. These are also illustrated in Figure 9.1.

Policy components

Our policy components were designed to extend and enrich OASIS RBAC policy components. They support many features of other RBAC models, but we have not examined whether these components can be used to help conversion from an arbitrary RBAC policy model to another one.

OASIS RBAC policies can become rather complex. Our meta-policies help to organise and structure policies, but the understanding of particular policies can be further aided by using

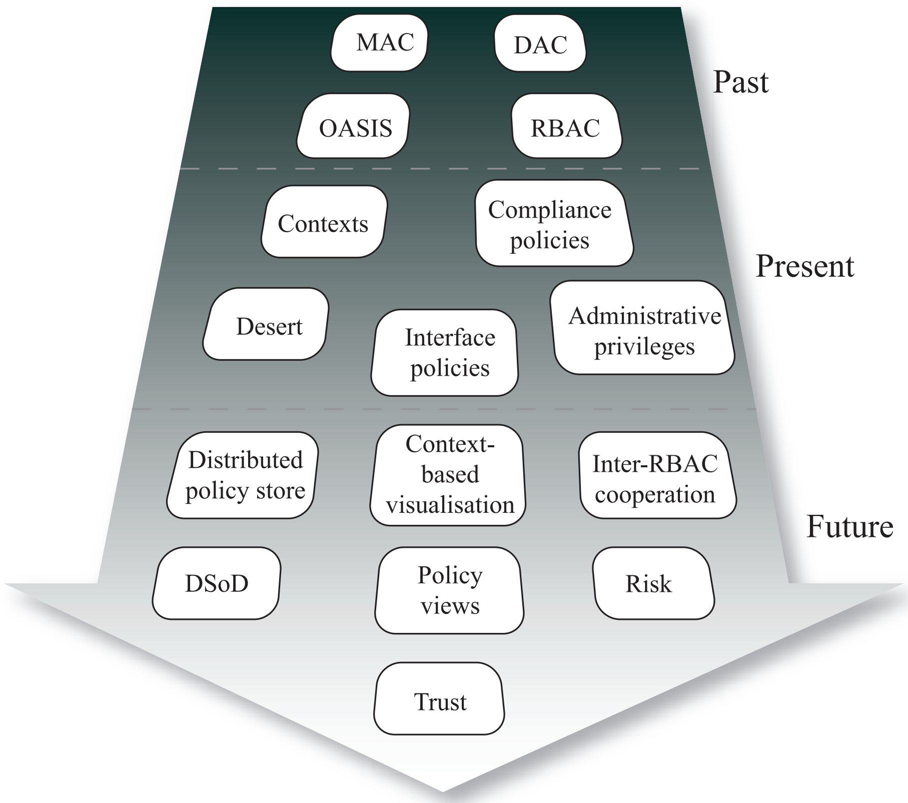


Figure 9.1: Past, present, and the future.

visual representations. An ongoing study visualises policy rules using graphs. Such research is much needed, since policy administrators may have different qualifications, thus some might have very limited knowledge of access control systems. For example, if a patient may be allowed to modify NHS policies pertaining to her, the policy interface must be straightforward for her, as a non computer scientist policy engineer, to use.

Contexts

We have described many uses for contexts. Among others these included organising policy components for easier administration, forming access control groups, and restricting prerequisite groups.

In Section 4.6 we considered additional areas where contexts may be useful. We mentioned how to apply contexts to separate functional and organisational roles, how to associate logging requirements with roles belonging to a particular context, and how to differentiate roles from the perspective of failure. These uses already indicate a large set of possibilities for future research, but the grouping aspect of contexts can be further explored.

For example, David Eyers of the OPERA research group is working on methods to apply contexts to *Dynamic Separation of Duties* (DSoD, see Section 2.2.2). Such an extension will require modification to the OASIS access control decision engine, but the advantage of having DSoD justifies such change.

Another possible extension to contexts is the association of contexts, and roles belonging to particular context elements, with various stages of a business process. Such use of contexts can help *workflow* specifications in which workflow is partly enforced through the information flow restrictions between context labels.

Policy components may be marked with many context elements, thus allowing administrators to group policy components according to various dimensions. This allows different *views to a policy*, and this can be exploited in policy visualisations. For example, on page 62 we used two orthogonal policy component classifications; we differentiated components from the web security and research group perspectives. Depending on which of these two aspects a policy administrator is interested in, policy components may be arranged differently in a visual representation.

Compliance policies

Our compliance policies form a powerful means to specify constraints over policies. They can cater for indirect role activation and negation. It could be an interesting research project to examine how conformance to compliance policies can be tested for different RBAC systems, and not only for OASIS RBAC policies.

We have considered only RBAC-level restrictions on policies, i.e. our basic primitives were roles, privileges, and rules. To overcome differences between various RBAC models, it could be interesting to use some *lower-level primitives*, such as (subject, object, access) triples from [BdVS02], extended with RBAC conditions, to find *equivalent policies*. Such research could be especially promising from the perspective of contexts and information flow restrictions.

Interface policies

If compliance policies can be extended to multiple RBAC systems that differ significantly (e.g. from the aspect of parameter support, typing support, constraint limitations, etc.), a possible way to continue is to enable *cooperation between various RBAC systems*. In such environments interface policies can be used to mediate differences between access control policy domains.

The primary goal of interface policies is to enable cooperation between independent policy domains. Whether cooperation is allowed depends on the relevant policy administrators. However, in certain cases *ad-hoc cooperation* can be set up based on *trust*. A possible extension to our work could be to utilise interface policy compliance in inter-domain trust decisions, and then use interface policies to set up collaborations automatically. Since a policy may comply with many different interface policies, trust information could be used to select the appropriate interface policy.

Policy management privileges

The policy management privileges we have specified provide a powerful means to control access to RBAC policies at a very fine granularity. They can already be applied to various RBACs, thus research in this direction is less interesting. However, as we have mentioned in Section 7.4.2, these policy management privileges could be organised into a hierarchy. Such *privilege hierarchies* would allow more concise policy specifications.

An extension to our privileges could be their association with *risk*. The operations we permit on policies are not equal from the perspective of the potential damage that their misuse can cause. By associating risk with certain privileges one could analyse policies, and point out roles that may potentially be dangerously powerful. Using our contexts we could associate a threshold with roles, and may require that the cumulative risk of roles belonging to a specific context stays below the threshold.

Policy store

Among the main motivations for our research is the development of a *policy store* in which policies are stored and managed by many administrators in a controlled way. We have contributed much to the design of such a policy store, but there are many issues remaining that need to be addressed. Currently there is ongoing work on how to distribute such policy stores. This introduces many challenges, since policies can be modified concurrently and changes need to be synchronised.

Case studies

Once the policy store is ready to be deployed, it will be possible to test various policies in large-scale case studies. Such case studies can help answer questions relating to optimal policy/meta-policy size.

Case-studies may also help us to design better meta-policies, possibly hierarchical meta-policies, that can be easily used as templates for new policy versions, allowing fast access control policy deployment.

9.3 Conclusion

In this thesis we have presented a means to manage RBAC policies, particularly OASIS RBAC policies, in an environment where policies evolve, and are managed in a distributed fashion by many administrators.

A Compliance component restrictions

	lossy	unique	split	optional	isdefined()	isnotdefined()	min()	max()	datatype()	binding()	mbship_override
Abstract policy components											
Role specification		✓	✓	✓	✓	✓					
Appointment specification		✓	✓	✓	✓	✓					
Env. predicate specification		✓	✓	✓	✓	✓					
Function specification		✓	✓	✓	✓	✓					
Privilege specification		✓	✓	✓	✓	✓					
Parameters of the above	✓	✓	✓	✓	✓	✓					
Authorisation rule				✓	✓	✓					
prerequisites				✓	✓	✓					
Activation rule				✓	✓	✓					
prerequisites				✓	✓	✓					
Arbitrary components											
Role				✓	✓	✓	✓	✓			
Appointment				✓	✓	✓	✓	✓		✓	
Env. predicate				✓	✓	✓	✓	✓		✓	
Privilege				✓	✓	✓	✓	✓			
Parameters of the above	✓			✓	✓	✓	✓	✓	✓		
Authorisation rule				✓	✓	✓	✓	✓			
Prerequisite				✓	✓	✓					
Activation rule				✓	✓	✓	✓	✓			
Prerequisites				✓	✓	✓					✓
Other compliance policy components											
Implicit rules				✓	✓	✓	✓	✓			
Prerequisites				✓	✓	✓	✓	✓			✓[1]
Contexts		✓	✓	✓	✓	✓					

Table A.1: Restrictions permitted for compliance policy components

[1]Only for implicit activation rules

B Policy access privileges

In the following table we list the privileges associated with policy component management, which we described in Chapter 7.

Name	***Parameters***
Data type privilges	
type_new	(t type_id, typeinfo string, P,C)
type_del	(t type_id, P,C)
type_inherit_new	(parent type_id, child type_id, P,C)
type_inherit_del	(parent type_id, child type_id, P,C)
type_bind	(t type_id, P,C)
type_unbind	(t type_id, P,C)
type_context_new	(t type_id, c context_element, P,C)
type_context_del	(t type_id, c context_element, P,C)
Function privileges	
func_new	(f func_id, functioninfo string, P,C)
func_del	(f func_id, P,C)
func_retval_new	(f func_id, ret_type type_id, P,C)
func_param_new	(f func_id, param_name string, param_type type_id, P,C)
func_param_del	(f func_id, param_name string, P,C)
func_bind	(f func_id, P,C)
func_unbind	(f func_id, P,C)
func_context_new	(f func_id, c context_element, P,C)
func_context_del	(f func_id, c context_element, P,C)
Environmental predicate privileges	
env_new	(e env_id, environmentinfo string, P,C)
env_del	(e env_id, P,C)
env_param_new	(e env_id, param_name string, param_type type_id, in_out boolean, P,C)
env_param_del	(e env_id, pname string, P,C)
env_bind	(e env_id, P,C)
env_unbind	(e env_id, P,C)
env_context_new	(e env_id, c context_element, P,C)
env_context_del	(e env_id, c context_element, P,C)
Privilege privileges	
priv_new	(p priv_id, P,C)
priv_del	(p priv_id, P,C)
priv_param_new	(p priv_id, param_name string, param_type type_id, P,C)
priv_param_del	(p priv_id, param_name string, P,C)
priv_dep_new	(p priv_id, priv_dependent priv_id, P,C)
priv_dep_del	(p priv_id, priv_dependent priv_id, P,C)
priv_bind	(p priv_id, P,C)
priv_unbind	(p priv_id, P,C)
priv_context_new	(p priv_id, c context_element, P,C)
priv_context_del	(p priv_id, c context_element, P,C)
Role privileges	
role_new	(r role_id, P,C)
role_del	(r role_id, P,C)
	P = pol policy_id, C = c context

Name	***Parameters***
role_param_new	(r role_id, param_name string, param_type type_id, P,C)
role_param_del	(r role_id, param_name string, P,C)
role_bind	(r role_id, P,C)
role_unbind	(r role_id, P,C)
role_context_new	(r role_id, c context_element, P,C)
role_context_del	(r role_id, c context_element, P,C)
Appointment privileges	
app_new	(a appointment_id, appointmentinfo string, P,C)
app_del	(a appointment_id, P,C)
app_param_new	(a appointment_id, param_name string, param_type type_id, P,C)
app_param_del	(a appointment_id, param_name string, P,C)
app_bind	(a appointment_id, P,C)
app_unbind	(a appointment_id, P,C)
app_context_new	(a appointment_id, c context_element, P,C)
app_context_del	(a appointment_id, c context_element, P,C)
Term privileges	
term_const_new	(t term_id, t type_id, value string, P)
term_func_new	(t term_id, f cont_func_id, P)
term_var_new	(t term_id, t type_id, P)
term_del	(t term_id, P)
Container and activation container privileges	
cont_priv_new	(c contp_id, p priv_id, P)
cont_role_new	(c contr_id, r role_id, P)
cont_app_new	(c conta_id, a appointment_id, P)
cont_env_new	(c conte_id, e env_id, P)
cont_del	(c cont{prae}_id, P)
cont{prae}_param_new	(c cont{prae}_id, t term_id, P)
cont{prae}_param_del	(c cont{prae}_id, nr int, P)
acont_role_new	(c acontr_id, r role_id, membership boolean, P)
acont_app_new	(c conta_id, a appointment_id, membership boolean, P)
acont_env_new	(c aconte_id, e env_id, membership boolean, P)
acont_del	(c acont{prae}_id, P)
acont{rae}_param_new	(c acont{rae}_id, t term_id, P)
acont{rae}_param_del	(c acont{rae}_id, nr int, P)
Authorisation rule privileges	
arule_{new,del}	(r arule_id, P,C)
arule_var_{new,del}	(r arule_id, v term_id, P)
arule_priv_{new,del}	(r arule_id, p contp_id, P,C)
arule_prole_{new,del}	(r arule_id, r acontr_id, P,C)
arule_papp_{new,del}	(r arule_id, a conta_id, P,C)
arule_penv_{new,del}	(r arule_id, p aconte_id, P,C)
arule_bind	(r arule_id, P,C)
arule_unbind	(r arule_id, P,C)
arule_context_new	(r arule_id, c context_element, P,C)
arule_context_del	(r arule_id, c context_element, P,C)
Role activation rule privileges	
erule_{new,del}	(r erule_id, P,C)
erule_var_{new,del}	(r erule_id, v term_id, P)
erule_target_{new,del}	(r erule_id, r contr_id, P,C)
erule_prole_{new,del}	(r erule_id, r contr_id, P,C)
erule_papp_{new,del}	(r erule_id, a conta_id, P,C)
erule_penv_{new,del}	(r erule_id, p conte_id, P,C)
erule_bind	(r erule_id, P,C)
erule_unbind	(r erule_id, P,C)
erule_context_new	(r erule_id, c context_element, P,C)
erule_context_del	(r erule_id, c context_element, P,C)
Meta-policy handling privileges	
context_new	(i context_policy_id, p policy_id)
context_del	(i context_policy_id, p policy_id)
compliance_new	(i compliance_policy_id, p policy_id)
compliance_del	(i compliance_policy_id, p policy_id)
interface_new	(i interface_policy_id, p policy_id)
interface_del	(i interface_policy_id, p policy_id)
	P = pol policy_id, C = c context

C OASIS policy schema

This XML schema is used for policy storage and policy distribution.

```
<?xml version="1.0"?>
<xs:schema id="Policy 1.0"
  targetNamespace="http://cl.cam.ac.uk/opera/MetaPolicy/2003/08"
  xmlns:mstns="http://cl.cam.ac.uk/opera/MetaPolicy/2003/08"
  xmlns="http://cl.cam.ac.uk/opera/MetaPolicy/2002/02"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="policies">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="contextelement" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="contextin" type="xs:string" minOccurs="unbounded" />
              <xs:element name="contextout" minOccurs="0" maxOccurs="unbounded" nillable="true">
                <xs:complexType>
                  <xs:simpleContent msdata:ColumnName="contextout_Text" msdata:Ordinal="0">
                    <xs:extension base="xs:string">
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
            <xs:attribute name="parentname" form="unqualified" type="xs:string" use="optional"/>
            <xs:attribute name="initial" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="privilegespec" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="param1" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="param2" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="rolespec" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="param3" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
            <xs:attribute name="context" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="environmentspec" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="param7" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                  <xs:attribute name="mode" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="param8" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                  <xs:attribute name="mode" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
            <xs:attribute name="binding" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="appointmentspec" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="param9" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
            <xs:attribute name="binding" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="activation" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="role1" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable1" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="membership" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="target1" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable2" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="role2" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable3" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="membership" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="environment" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable4" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="variable5" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="membership" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="appointment" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable6" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="target2" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable7" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="variable8" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="authorisation" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="target3" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable9" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="constant" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="value" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
              <xs:element name="role3" minOccurs="0" maxOccurs="unbounded">
```

```
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variable10" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" form="unqualified" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" form="unqualified" type="xs:string" />
                  <xs:attribute name="membership" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" form="unqualified" type="xs:string" />
            <xs:attribute name="context" form="unqualified" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="targetNamespace" form="unqualified" type="xs:string" />
    </xs:complexType>
  </xs:element>
  <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:EnforceConstraints="False">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="policies" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Bibliography

[AKS03] Mohammad A. Al-Kahtani and Ravi Sandhu. Induced role hierarchies with attribute-based RBAC. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, pages 142–148. ACM Press, 2003.

[AMN02] Xuhui Ao, Naftaly Minsky, and Thu Nguyen. A hierarchical policy specification language and enforcement mechanism for governing digital enterprises. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 38–49, 2002.

[AS99] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 43–54, 1999.

[Awi97] Roland Awischus. Role based access control with the security administration manager (SAM). In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 61–68, 1997.

[Bak00] Dixie B. Baker. PCASSO: A model for safe use of the internet in healthcare. *Journal of American Health Information Management Association (AHIMA)*, 71(3):33–38, March 2000.

[BBF00] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: a temporal role-based access control model. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 21–30, 2000.

[BBF01] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, August 2001.

[BD99] Konstantin Beznosov and Yi Deng. A framework for implementing role-based access control using CORBA security service. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 19–30, 1999.

[BdVS02] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):1–35, 2002.

[BE03] András Belokosztolszki and David Eyers. Shielding RBAC infrastructures from cyberterrorism. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security, July 28-31, 2002, Kings College, Cambridge, U.K.*, volume 256 of *IFIP Information Processing*, pages 3–14. Kluwer Academic Publishers, 2003.

[BEM03] András Belokosztolszki, David M. Eyers, and Ken Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *Policy 2003: IEEE Fourth International Workshop on Policies for Distributed Systems and Networks*, pages 99–110, 2003.

[BEP$^+$03] András Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceeding of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, U.S.A., 2003.

[BEWM03] András Belokosztolszki, David M. Eyers, Wei Wang, and Ken Moody. Policy storage for role-based access control systems. In *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, pages 196–201, 2003.

[Bez98] Konstantin Beznosov. Requirements for access control: US Healthcare domain. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, page 43, 1998.

[BFA97] Elisa Bertino, Elena Ferrari, and Vijayalakshmi Atluri. A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 1–12, 1997.

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[BI98] Christophe Bidan and Valérie Issarny. Dealing with multi-policy security in large open distributed systems. In *Computer Security - ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 1998.

[Bib75] Ken J. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, April 1975.

[BJS96] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy (SSP'96)*, 1996.

[BL75] David E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corporation, July 1975.

[BLM01] Jean Bacon, Michael Lloyd, and Ken Moody. Translating role-based access control policy within context. In *Policies for Distributed Systems and Networks, International Workshop (POLICY'01), Bristol, UK*, pages 107–119, 2001.

[BM02] András Belokosztolszki and Ken Moody. Meta-policies for distributed role-based access control systems. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 106–115, 2002.

[BMB$^+$00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, pages 68–77, March 2000.

[BMCO03] Jean Bacon, Ken Moody, David Chadwick, and Oleksandr Otenko. Persistent versus dynamic role membership. In *Proceedings of the IFIP WG 11.3 Conference on Data and Applications Security*, 2003.

[BME04] András Belokosztolszki, Ken Moody, and David M. Eyers. A formal model for hierarchical policy contexts. In *Policy 2004: IEEE Fifth International Workshop on Policies for Distributed Systems and Networks*, 2004.

[BMY01] Jean Bacon, Ken Moody, and Walt Yao. Access control and trust in the use of widely distributed services. In *Middleware'01*, volume 2218, pages 300–315, November 2001.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540, November 2002.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy (SSP'89)*, pages 206–214, 1989.

[BS00a] Ezedin Barka and Ravi Sandhu. Framework for role-based delegation models. In *Sixteenth Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 2000.

[BS00b] Ezedin Barka and Ravi Sandhu. A role-based delegation model and some extensions. In *Twenty-third National Information Systems Security Conference*, Baltimore, MD, October 2000.

[BS03] András Belokosztolszki and Erich Schikuta. An XML based framework for self-describing I/O data. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '03)*, pages 324–332, 2003.

[CLS+01] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dev, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, pages 10–20, 2001.

[CO02] David W. Chadwick and Alexander Otenko. The PERMIS X.509 role based privilege management infrastructure. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 135–140. ACM Press, 2002.

[Col97] Robert M. Colomb. Impact of semantic heterogeneity on federating databases. *The Computer Journal*, 40(5):235–244, 1997.

[Cra02] Jason Crampton. Administrative scope and role hierarchy operations. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 145–154. ACM Press, 2002.

[CS95] Fang Chen and Ravi S. Sandhu. Constraints for role-based access control. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, pages II–39–46, 1995.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy (SSP'87)*, pages 184–195, Los Angeles, CA, April 1987. IEEE Computer Society Press.

[CW93] Stefano Ceri and Jennifer Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, pages 108–119, 1993.

[Dam02] Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 2002.

[DBE$^+$04] Nathan Dimmock, András Belokosztolszki, David Eyers, Jean Bacon, and Ken Moody. Using trust and risk in role-based access control policies. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies (SACMAT'04).* ACM Press, 2004.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Policies for Distributed Systems and Networks, International Workshop (POLICY'01), Bristol, UK*, pages 18–38, 2001.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[Did97] Tor Didriksen. Rule based database access control – a practical approach. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 143–151, 1997.

[ES95] Jeremy Epstein and Ravi Sandhu. NetWare 4 as an example of role-based access control. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, pages II–71–82, 1995.

[Fad99] Glenn Faden. RBAC in UNIX administration. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 95–101, 1999.

[FH97] Eduardo B. Fernandez and J. C. Hawkins. Determining role rights from use cases. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 121–125, 1997.

[FS96] Norbert E. Fuchs and Rolf Schwitter. Attempto controlled english. In *The First International Workshop on Controlled Language Applications*, pages 124–136, 1996.

[FSG$^+$01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, August 2001.

[FSS98] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English - not just another logic specification language. In *Logic Program Synthesis and Transformation*, pages 1–20, 1998.

[Gar02] Gartner, Inc. Best practices: Access rights are not created equal, 2002. `http://security2.gartner.com/story.php.id.102.s.1.jsp`.

[GB98] Cheh Goh and Adrian Baldwin. Towards a more complete model of role. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, pages 55–62, 1998.

[GGF98] Virgil D. Gligor, Aerban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *IEEE Symposium on Security and Privacy (SSP'98)*, pages 172–183, 1998.

[GI96] Luigi Giuri and Pietro Iglio. A formal model for role-based access control with constraints. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop*, pages 136–145, 1996.

[GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 153–159, 1997.

[Giu95] Luigi Giuri. Role-based access control: a natural approach. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, pages II–33–37, 1995.

[Giu98] Luigi Giuri. Role-based access control in Java. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, pages 91–100, 1998.

[Giu99] Luigi Giuri. Role-based access control on the Web using Java. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 11–18, 1999.

[GMS94] Cheng Hian Goh, Stuart E. Madnick, and Michael D. Siegel. Context interchange: overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In *Proceedings of the third International Conference on Information and Knowledge Management*, pages 337–346. ACM Press, 1994.

[HA99] Wei-Kuang Huang and Vijayalakshmi Atluri. SecureFlow: a secure Web-enabled workflow management system. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 83–94, 1999.

[Has00] Wilhelm Hasselbring. Information system integration. *Communications of the ACM*, 43(6):32–38, June 2000.

[Hay96] Richard Hayton. *OASIS An Open Architecture for Secure Interworking Services.* PhD thesis, University of Cambridge, 1996. Technical Report No. 399.

[HB99] Thomas Hildmann and Jörg Barholdt. Managing trust between collaborating companies using outsourced role based access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 105–111, 1999.

[HBM98] Richard J. Hayton, Jean M. Bacon, and Ken Moody. Access control in an open distributed environment. In *Proceedings of IEEE Symposium on Security and Privacy (SSP'98)*, pages 3–14, 1998.

[HGPS99] John Hale, Pablo Galiasso, Mauricio Papa, and Sujeet Shenoi. Security policy coordination for heterogeneous information systems. In *Proceedings of the Fifteenth Annual Computer Security Applications Conference*, pages 219–228, December 1999.

[HL03] Michael Howard and David LeBlanc. *Writing Secure Code.* Microsoft Press, Redmond, Washington, U.S.A., second edition, 2003.

[HM85] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Information Systems (TOIS'85)*, 3(3):253–278, July 1985.

[HM93] Joachim Hammer and Dennis McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Journal for Intelligent and Cooperative Information Systems*, 2(1):51–83, 1993.

[HMM$^+$00] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy (SSP'00)*, pages 2–14, 2000.

[HMT$^+$90] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual specification of security. In *IEEE Transactions os Software Engineering*, pages 1185–1197, 1990.

[Hom02] Alexis B. Hombrecher. *Reconciling Event Taxonomies Across Administrative Domains.* PhD thesis, University of Cambridge, 2002.

[Hul97] Richard Hull. Managing semantic heterogeneity in databases: a theoretical prospective. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 51–61, 1997.

[HV01] Michael Hitchens and Vijay Varadharajan. Tower: A language for role based access control. In *Policies for Distributed Systems and Networks, International Workshop (POLICY'01), Bristol, UK*, pages 88–107, 2001.

[HYBM00] John H. Hine, Walt Yao, Jean Bacon, and Ken Moody. An architecture for distributed OASIS services. In *Middleware'00*, pages 104–120, 2000.

[IO03] Cecilia Ionita and Sylvia Osborn. Privilege administration for the role graph model. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security, July 28-31, 2002, Kings College, Cambridge, U.K.*, volume 256 of *IFIP International Federation for Information Processing*, pages 15–25. Kluwer Academic Publishers, 2003.

[Jae99] Trent Jaeger. On the increasing importance of constraints. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 33–42, 1999.

[JHS75] Michael D. Schroeder Jerome H. Saltzer. The protection of information in computer systems. *IEEE*, 63(9):1278–1308, September 1975.

[JSS97] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorisations. In *Proceedings of IEEE Symposium on Security and Privacy (SSP'97)*, page 3142, 1997.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD'97*, pages 474–485, 1997.

[JSSS01] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS'01)*, 26(2):214–260, 2001.

[JT01] Trent Jaeger and Jonathon E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):158–190, 2001.

[Ken91] William Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *Proceedings of the Seventeenth Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

[KKSM02] Axel Kern, Martin Kuhlmann, Andreas Schaad, and Jonathan Moffett. Observations on the role life-cycle in the context of enterprise security management. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 43–51. ACM Press, 2002.

[KSM03] Axel Kern, Andreas Schaad, and Jonathan Moffett. An administration concept for the enterprise role-based access control model. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, pages 3–11. ACM Press, 2003.

[KSS03] Martin Kuhlmann, Dalia Shohat, and Gerhard Schimpf. Role mining - revealing business roles for security administration using data mining technology. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT'03)*, pages 179–186. ACM Press, 2003.

[Kuh97] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 23–30, 1997.

[Lam71] Butler Lampson. Protection. In *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[Llo00] Michael S. Lloyd. Conversion of NHS access control policy to formal logic. Master's thesis, University of Cambridge, 2000.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy (SSP'02)*, pages 114–130, Berkeley, CA, 2002. IEEE Computer Society Press.

[Loc01] Thomas Lockhart. *PostgreSQL Programmer's Guide*. PostgreSQL Inc., 2001.

[LS97] Emil Lupu and Morris Sloman. Reconciling role based management and role based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 135–141, 1997.

[LS99] Emil Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[LS00] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system, 2000.

[LSPR93] Ee-Peng Lim, Jaideep Srivastava, Satya Prabhakar, and James Richardson. Entity identification in database integration. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 294–301. IEEE Computer Society, 1993.

[Lup98] Emil C. Lupu. *A Role-Based Framework for Distributed Systems Management.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1998.

[LW88] Frederick H. Lochowsky and Carson C. Woo. Role-based security in data base management systems. In Landwehr Ed., editor, *Database Security: Status and Prospects*, pages 209–222, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.

[MBBC02] Daniel Masys, Dixie Baker, Amy Butros, and Kevin E. Cowles. Giving patients access to their medical records via the internet: The PCASSO experience. *Journal of American Health Information Management Association (AHIMA)*, 9(2):181–191, Mar-Apr 2002.

[MBG99] Marco Casassa Mont, Adrian Baldwin, and Cheh Goh. POWER Prototype: Towards Integrated Policy-Based Management. Technical report, Hewlett-Packard Laboratories, Bristol, October 1999. HPL-1999-126.

[Mik02] Zoltán Miklós. Towards an access control mechanism for wide-area publish/subscribe systems. In *International Workshop on Distributed Event-Based Systems (DEBS02)*, pages 516–521, 2002.

[ML99] Jonathan D. Moffett and Emil C. Lupu. The uses of role hierarchies in access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 153–160, 1999.

[Mof98] Jonathan D. Moffett. Control principles and role hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, pages 63–69, 1998.

[MPI97] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Specification, Message Passing Interface Forum, 1997. `http://www-unix.mcs.anl.gov/mpi/mpi-standard/ mpi-report-3.0/mpi2-report.htm`.

[MR00] Axel Mönkeberg and René Rakete. Three for one: role-based access-control management in rapidly changing heterogeneous environments. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 83–88, 2000.

[MS93] Jonathan D. Moffett and Morris S. Sloman. Policy hierarchies for distributed system management. *IEEE JSAC Special Issue on Network Management*, 11(9):1404–1414, 11 1993.

[Nat02] National Health Service. Your guide to the NHS, July 2002. `http://www.nhs.uk/nhsguide/nhs_guide.pdf`.

[NC00] SangYeob Na and SuhHyun Cheon. Role delegation in role-based access control. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 39–44, 2000.

[NO94] Matunda Nyanchama and Sylvia Osborn. Access rights administration in role-based security systems. In *IFIP Workshop on Database Security*, pages 37–56, 1994.

[NO95a] Matunda Nyanchama and Sylvia Osborn. The role graph model. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, pages II–25–31, 1995.

[NO95b] Matunda Nyanchama and Sylvia L. Osborn. Modeling mandatory access control in role-based security systems. In *IFIP Workshop on Database Security*, pages 129–144, 1995.

[NO99] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):3–33, 1999.

[NS02] Gustaf Neumann and Mark Strembeck. A scenario-driven role engineering process for functional rbac roles. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 33–42. ACM Press, 2002.

[OMG02] OMG. The Common Object Request Broker Architecture: Core Specification, Revision 3.0. Specification, Object Management Group (OMG), December 2002.

[OS02] Sejong Oh and Ravi Sandhu. A model for role administration using organization structure. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 155–162. ACM Press, 2002.

[Osb97] Sylvia Osborn. Mandatory access control and role-based access control revisited. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 31–40, 1997.

[Osb02] Sylvia L. Osborn. Information flow analysis of an rbac system. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 163–168. ACM Press, 2002.

[Ott02] Amon Ott. The role compatibility security model. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NORDSEC'02)*, Karlstad, Sweden, 2002.

[PB02] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611–618, Vienna, Austria, July 2002.

[PS99] Joon S. Park and Ravi Sandhu. RBAC on the Web by smart certificates. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, pages 1–9, 1999.

[PS02] Peter R Pietzuch and Brian Shand. A Framework for Object-Based Event Composition in Distributed Systems. Malaga, Spain, June 2002. Presented at the 12th International Network for PhD Students in Object Oriented Systems (PhDOOS) Workshop.

[PSA01] Joon S. Park, Ravi Sandhu, and Gail-Joon Ahn. Role-based access control on the web. *ACM Transactions on Information and System Security (TISSEC)*, 4(1):37–71, February 2001.

[RSW00] Haio Roeckle, Gerhard Schimpf, and Rupert Weidinger. Process-oriented approach for role-finding to implement role-based security administration in a large industrial organization. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 103–110, 2000.

[San88] Ravi Sandhu. Transaction control expressions for separation of duties. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pages 282–286, December 1988.

[San92] Ravi Sandhu. The typed access matrix model. In *Proceedings of the Eleventh IEEE Symposium on Security and Privacy (SSP'92)*, pages 122–136, 1992.

[San93] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[San95] Ravi Sandhu. Roles versus groups. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, pages I–25–26, 1995.

[San98] Ravi Sandhu. Role activation hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, pages 33–40, 1998.

[San00] Ravi Sandhu. Engineering authority and trust in cyberspace: the OM-AM and RBAC way. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 111–119, 2000.

[SB02] Brian Shand and Jean Bacon. Policies in accountable contracts. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 80–91, 2002.

[SBC$^+$97] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 model for role-based administration of roles: preliminary description and outline. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 41–50, 1997.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 47–63, 2000.

[SL90] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, September 1990.

[Slo94] Morris Sloman. Policy driven management for distributed systems. *Network and Systems Management*, 2(4):333–360, 1994.

[SM98] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of the Third ACM Workshop on Role-Based Access Control (RBAC'98)*, pages 47–54, 1998.

[SM02a] Andreas Schaad and Jonathan Moffett. Delegation of obligations. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 25–35, 2002.

[SM02b] Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 13–22. ACM Press, 2002.

[Sma00] Stephen Smalley. A security policy configuration for the security-enhanced linux, 2000. `http://www.nsa.gov/selinux/policy-abs.html`.

[SMJ01] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a European bank: a case study and discussion. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, pages 3–9, 2001.

[SR01] Len Seligman and Arnon Rosenthal. XML's impact on databases and data sharing. *IEEE Computer*, 34(6):59–67, 2001.

[SRJ01] Pierangela Samarati, Michael K. Reiter, and Sushil Jajodia. An authorization model for a public key management service. *ACM Transactions on Information and System Security (TISSEC)*, 4(4):453–482, 2001.

[SS94] Ravi Sandhu and Pierrangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[SSR94] Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. In *ACM Transactions on Database Systems*, volume 19, pages 254–290, 1994.

[Sun01] Sun Microsystems. Java Message Service. Specification, Sun Microsystems, 2001. `http://java.sun.com/products/jms/`.

[SWY$^+$02] Kent Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobson, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, pages 68–79, 2002.

[SZ97] Richard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *PCSFW: Proceedings of the Tenth Computer Security Foundations Workshop*, pages 183–194. IEEE Computer Society Press, 1997.

[Tho97] Roshan K. Thomas. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-Based Access Control (RBAC'97)*, pages 13–19, 1997.

[TJ00] Jonathon E. Tidswell and Trent Jaeger. Integrated constraints and inheritance in DTAC. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control (RBAC'00)*, pages 93–102, 2000.

[TP01] Jonathon E. Tidswell and John M. Potter. A graphical definition of authorization schema in the DTAC model. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, pages 109–120, 2001.

[TS97] Roshan K. Thomas and Ravi Sandhu. Task-based Authorization Controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Proceedings of the IFIP WG 11.3 Workshop on Database Security*, pages 166–181, August 1997.

[W3C99] W3C. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, World Wide Web Consortium, November 1999.

[W3C01] W3C. XML Schema Part 0: Primer. W3C Recommendation, World Wide Web Consortium, May 2001.

[WCEW02] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security issues and requirements in internet-scale publish-subscribe systems. In *Proceedings of the Thirty-Fifth Annual Hawaii International Conference on System Sciences (HICSS'02)*, Big Island, Hawaii, 2002.

[WFSM02] Marc Wilikens, Simone Feriti, Alberto Sanna, and Marcelo Masera. A context-related authorization and access control method based on RBAC: A case study from the health care domain. In *Seventh ACM Symposium on Access Control Models and Technologies (SACMAT'02)*, pages 117–124. ACM Press, 2002.

[Wie94] René Wies. Policy definition and classification: Aspects, criteria, and examples. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, October 1994.

[Yao02] Walt Teh-Ming Yao. *Trust Management for Widely Distributed Systems*. PhD thesis, University of Cambridge, 2002.

[YLS96] Nicholas Yialelis, Emil Lupu, and Morris Sloman. Role based security for distributed object systems. In *Proceedings of the Fifth Workshops on Enabling Technologies: Infrastructure for Collaborating Enterprises (WET-ICE'96)*, pages 80–85, 1996.

[YMB01] Walt Yao, Ken Moody, and Jean Bacon. A model of OASIS role-based access control and its support for active security. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, pages 171–181, 2001.

[ZAC01] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role based delegation. In *Sixth ACM Symposium on Access Control Models and Technologies (SACMAT'01)*, pages 153–162, 2001.