The formalisation and transformation of access control policies

Mark A. Slaymaker
Oxford University Computing Laboratory
Kellogg College
Doctor of Philosophy thesis

Supervisor: Andrew Simpson

Hilary 2011

Acknowledgements

I would like to thank my supervisor Andrew Simpson for his encouragement and guidance since beginning this endeavour.

I would also like to thank my colleagues for their encouragement, especially David Power, for his patience in discussing various ideas.

Finally I would like to thank my family and friends, especially: my wife for her support which gave me the fortitude to finish this endeavour, R and G for their unwavering confidence in me, my parents for giving me the opportunity to follow the path that has lead to this work, and Val for many years of assistance that gave me the basic skills and confidence to complete this thesis.

The formalisation and transformation of access control policies

Mark A. Slaymaker
Oxford University Computing Laboratory
Kellogg College
Doctor of Philosophy thesis

Hilary 2011

Abstract

Increasing amounts of data are being collected and stored relating to every aspect of an individual's life, ranging from shopping habits to medical conditions. This data is increasingly being shared for a variety of reasons, from providing vast quantities of data to validate the latest medical hypothesis, to supporting companies in targeting advertising and promotions to individuals that fit a certain profile. In such cases, the data being used often comes from multiple sources — with each of the contributing parties owning, and being legally responsible for, their own data. Within such models of collaboration, access control becomes important to each of the individual data owners. Although they wish to share data and benefit from information that others have provided, they do not wish to give away the entirety of their own data. Rather, they wish to use access control policies that give them control over which aspects of the data can be seen by particular individuals and groups. Each data owner will have access control policies that are carefully crafted and understood — defined in terms of the access control representation that they use, which may be very different from the model of access control utilised by other data owners or by the technology facilitating the data sharing. Achieving interoperability in such circumstances would typically require the rewriting of the policies into a uniform or standard representation — which may give rise to the need to embrace a new access control representation and/or the utilisation of a manual, error-prone, translation.

In this thesis we propose an alternative approach, which embraces heterogeneity, and establishes a framework for automatic transformations of access control policies. This has the benefit of allowing data owners to continue to use their access control paradigm of choice. Of course, it is important that the data owners have some confidence in the fact that the new, transformed, access control policy representation accurately reflects their intentions. To this end, the use of tools for formal modelling and analysis allows us to reason about the translation, and demonstrate that the policies expressed in both representations are equivalent under access control requests; that is, for any given request both access control mechanisms will give an equivalent access decision. For the general case, we might propose a standard intermediate access control representation with transformations to and from each access control policy language of interest. However, for the purpose of this thesis, we have chosen to model the translation between role-based access control (RBAC) and the XML-based policy language, XACML, as a proof of concept of our approach. In addition to the formal models of the access control mechanisms and the translation, we provide, by way of a case study, an example of an implementation which performs the translation.

The contributions of this thesis are as follows. First, we propose an approach to resolving issues of authorisation heterogeneity within distributed contexts, with the requirements being derived from nearly eight years of work in developing secure, distributed systems. Our second contribution is the formal description of two popular approaches to access control: RBAC and XACML. Our third contribution is the development of an Alloy model of our transformation process. Finally, we have developed an application that validates our approach, and supports the transformation process by allowing policy writers to state, with confidence, that two different representations of the same policy are equivalent.

Contents

1 Introduction **1**

2 Background **8**

2.1 Outline . . . 8
2.2 A brief introduction to access control . . . 8
2.2.1 Mandatory access control . . . 9
2.2.2 Discretionary access control . . . 10
2.2.3 Access control in distributed contexts . . . 11
2.3 RBAC . . . 12
2.3.1 Outline . . . 12
2.3.2 Principles of RBAC . . . 13
2.3.3 The ANSI RBAC standard . . . 14
2.3.4 Role inheritance . . . 15
2.3.5 Criticisms of the ANSI standard . . . 16
2.3.6 RBAC alternatives and extensions . . . 16
2.4 XACML . . . 17
2.4.1 XACML: an overview . . . 17
2.4.2 Describing access control in XACML . . . 18
2.5 Access control and formal methods . . . 22
2.6 Alloy . . . 24
2.7 SOA and web services . . . 27
2.7.1 Motivation and background . . . 27
2.7.2 Virtual organisations . . . 28
2.7.3 sif . . . 29
2.7.4 sif and access control . . . 30
2.7.5 Service-Oriented Federated Authorization . . . 31
2.8 Summary . . . 32

3 Use cases and requirements **33**

3.1 Introduction . . . 33
3.2 Context and middleware applications . . . 33
3.3 Use cases . . . 35
3.4 Requirements . . . 37

3.5 Translation methods . . . 40
3.6 Adopted approach . . . 41
3.7 Summary . . . 42

4 Formal representations of role-based access control 43
4.1 A formal model of RBAC in Z . . . 43
4.1.1 A running example . . . 43
4.1.2 Types . . . 44
4.1.3 Core RBAC . . . 44
4.1.4 General role hierarchies . . . 45
4.1.5 Limited role hierarchies . . . 47
4.1.6 Static mutually exclusive roles . . . 47
4.1.7 Evaluation of access requests . . . 48
4.1.8 Sample requests . . . 49
4.1.9 Evaluations . . . 50
4.2 A formal model of RBAC in Alloy . . . 51
4.2.1 Types . . . 52
4.2.2 RBAC Core . . . 53
4.2.3 RBAC Hierarchy . . . 53
4.3 An Alloy example . . . 54
4.3.1 Definition of the Alloy instance . . . 54
4.3.2 RBAC Core instance . . . 55
4.3.3 RBAC Core test and results . . . 55
4.3.4 RBAC Hierarchy . . . 56
4.3.5 RBAC Hierarchy test and results . . . 57
4.4 Summary . . . 57

5 Formal representations of XACML 59
5.1 A formal model of XACML in Z . . . 59
5.1.1 Types . . . 59
5.1.2 Targets . . . 61
5.1.3 Rules . . . 62
5.1.4 Policies . . . 63
5.1.5 Policy sets . . . 64
5.1.6 XACML schema . . . 66
5.1.7 Valid XACML . . . 66
5.2 XACML requests and evaluation of policies . . . 68
5.2.1 Requests . . . 68
5.2.2 The evaluation of XACML . . . 70
5.2.3 Combining algorithms . . . 71
5.2.4 The effect of a rule . . . 74
5.2.5 The effect of a policy . . . 75
5.2.6 The effect of a policy set . . . 76

5.2.7 The effect of an XACML policy . . . 76
5.3 An example . . . 77
5.4 An Alloy model of XACML . . . 82
5.4.1 Alloy definition of Target . . . 83
5.5 Alloy definition of XACML components . . . 84
5.5.1 XACML in Alloy . . . 85
5.5.2 Comments on the Alloy model . . . 85
5.6 Example using XACML in Alloy . . . 86
5.7 Summary . . . 91

6 Translation of RBAC to XACML **92**
6.1 Translation outline . . . 93
6.2 Formal translation from RBAC to XACML using Z . . . 96
6.2.1 Helper functions . . . 96
6.2.2 Translation schemas . . . 97
6.3 A walked-through example translation . . . 103
6.4 Alloy . . . 113
6.4.1 RBAC to XACML . . . 113
6.4.2 A formal translation . . . 113
6.4.3 Validation assertions . . . 116
6.5 Alloy translation of the running example . . . 118
6.6 XACML to RBAC . . . 118
6.7 Summary . . . 120

7 Access control policy tool **121**
7.1 Prototype RBAC policy translation application . . . 121
7.2 Translation application interface . . . 121
7.3 Constraint checking . . . 129
7.4 Translation checking . . . 132
7.5 Testing and validation . . . 133
7.6 Summary . . . 133

8 Case study **135**
8.1 Background . . . 135
8.2 Requirements . . . 137
8.3 Policy capture using application . . . 140
8.4 Formalisation and transformation of the RBAC policy . . . 142
8.4.1 Z representation of RBAC . . . 142
8.4.2 Alloy representation of RBAC . . . 143
8.4.3 Z translation of RBAC to XACML . . . 144
8.5 Application translation . . . 147
8.6 Summary . . . 149

9 Discussion **150**
9.1 Contribution . . . 150
9.1.1 Research question . . . 150
9.1.2 Formalisation and transformation of access policies: in theory and practice 150
9.1.3 Validation . . . 151
9.2 Observations . . . 152
9.2.1 General observations . . . 152
9.2.2 Limitations of the work . . . 153
9.3 Future work . . . 154
9.4 Concluding remarks . . . 155

Bibliography **156**

A XACML representations of the running example **166**
A.1 Example XACML policy set . . . 166
A.2 A formal representation of the running example in XACML . . . 170
A.3 An example XACML request . . . 173
A.4 A formal representation of an XACML request . . . 174

B Capturing use cases and requirements **175**
B.1 eDiaMoND . . . 175
B.2 NeuroGrid . . . 176
B.3 Elicitation and capture . . . 176

C Alloy validation of the translation **177**
C.1 An Alloy representation of the running example . . . 177
C.2 Alloy tests . . . 177
C.3 Execution results . . . 184

D Translation Tool **191**
D.1 Additional constraints . . . 191
D.2 Validation of the running example . . . 193
D.3 Comparing the RBAC policy with the XACML translation of the running example 193
D.4 The XACML translation of the running example . . . 194

E Case study **199**
E.1 A Z representation of the case study policy . . . 199
E.2 An Alloy representation of the case study policy . . . 199

Glossary **201**

List of Figures

1.1 Access control 2
1.2 Centralised access control in a distributed system 3
1.3 Distributed access control 4
1.4 Hybrid access control 5

2.1 Bell-La Padula read up write down 10
2.2 RBAC conceptual diagram after [79] 13
2.3 XACML data flow [73] 19
2.4 Effects of different combining algorithms 20
2.5 A fragment of XACML 21
2.6 A generic virtual organisation 28
2.7 Access control architecture in sif 30

3.1 Access control with local data access 36
3.2 Local data access with new access control paradigm 37
3.3 Access control with distributed and local data access 38
3.4 Translation properties 39
3.5 Translating directly between different access control representations 40
3.6 Translation via a generic access control representation 41

4.1 RBAC Hierarchy relations 46

5.1 Effects of different combining algorithms 71

6.1 RBAC to XACML translation 94
6.2 Testing the conversion for a failure by ignoring `NotApplicable` 117

7.1 RBAC policy translation tool 122
7.2 RBAC policy translation tool: user panel 122
7.3 RBAC policy translation tool: action panel 123
7.4 RBAC policy translation tool: resource panel 123
7.5 RBAC policy translation tool: permission panel 124
7.6 RBAC policy translation tool: role panel 124
7.7 RBAC policy translation tool: user-role panel 125
7.8 RBAC policy translation tool: role-role panel 126

7.9 RBAC policy translation tool: role-permission panel 126
7.10 RBAC policy translation tool: summary panel 127
7.11 RBAC policy translation tool: test panel 128
7.12 RBAC policy translation tool: translate panel showing XACML 128
7.13 RBAC policy translation tool: translate panel showing comparison results 129
7.14 Translation tool showing RBAC policy validation 131
7.15 Translation tool showing testing of translated policy 132

8.1 Outline of the case study 137
8.2 Representation of the RBAC policy of the case study 139
8.3 Entering the users for the RBAC policy 141
8.4 Summary of RBAC roles and permissions for Conner 141
8.5 Checking a request and conformance to the Alloy model 142
8.6 Translation application XACML fragment relating to role *stat*1 145
8.7 XACML resulting from translation and testing of the conversion 147

Chapter 1

Introduction

Ever increasing amounts of data are being collected and stored relating to every aspect of an individual's life, from shopping habits to medical conditions. Issues pertaining to the maintenance of the privacy of the data in conjunction with the attitudes of the individual data subjects has long been the subject of research. For example, [9] examines the attitudes of respondents with respect to their sensitivity about personal data in relation to e-commmerce sites and classifies people into three broad categories:

- *privacy fundamentalists*, who made up 17% of the respondents, who are very concerned about any use of data and are generally unwilling to divulge data to websites even in the presence of privacy protection measures;
- the *pragmatic majority*, who made up 56% of respondents, who have concerns but to a lesser degree than the fundamentalists and will generally provide data when privacy protections are in place; and
- *marginally concerned*, who made up 27% of respondents, who are generally happy to provide data in most circumstances and only express very mild concerns about privacy.

These findings were broadly similar to those of other studies (such as [108]) which concentrate on USA and western European attitudes. In contrast, [61] is an initial study that looks at attitudes to privacy in India and finds that in general respondents were less concerned about sharing data than those in studies conducted in the United States of America. As an example, it was found that 29% of the Indian respondents were always happy to share health information with a website compared with only 6% of respondents in the United States of America.

When the wider picture of personal data is considered, it is necessary to include medical data as well as data held by the state necessary for the provision of services. Although it is commonly understood that a person's medical data should not be disclosed to anyone without a legitimate relationship that requires access to the data to provide care to that individual, there are circumstances when data pertaining to individuals could be aggregated for public health reasons and research. The collection of data for public health use, its values and ways of sharing it are discussed in [110]; also highlighted are the difficulties arising from acquiring data from multiple sources which have inconsistent representations as well as the additional overhead presented by the need to remove a number of identifying fields.

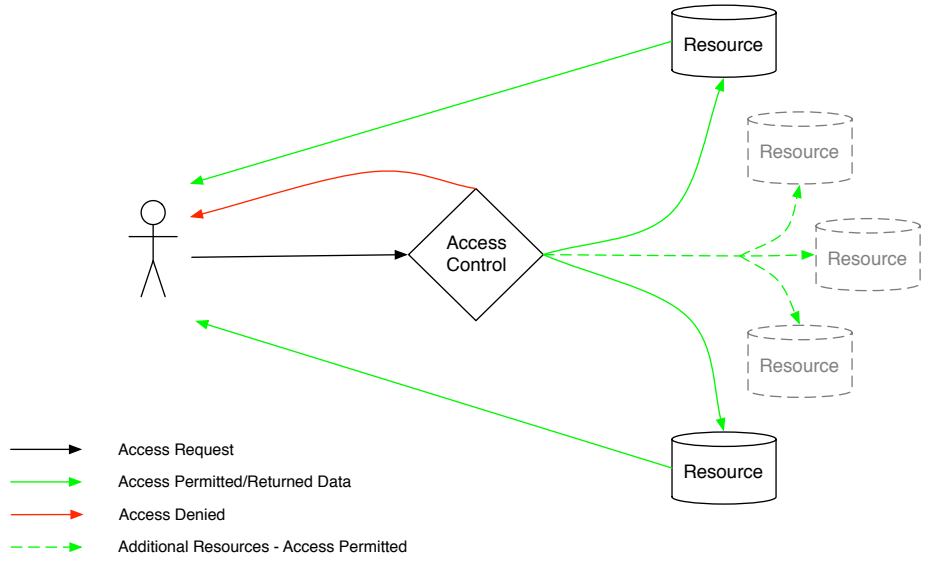

Figure 1.1: Access control

The importance of protecting the privacy of data subjects is key to the discussion in [10], where the concept of Hippocratic databases is outlined. They propose the use of key privacy principles to underpin the fundamental principle of a database. They base their principles on the United States Privacy Act of 1974 [2].

Such issues have come to the fore in recent years due primarily to the rise in social networking sites and the linking of data repositories. For example, in April 2001 an 'e-Science Programme' was launched in the UK, with the intention of utilising advanced computing techniques to provide researchers with access to widely distributed data and computational resources to facilitate 'bigger' and 'better' research. Alongside this, many projects concerning the distributed sharing of data and computational resources have received a great deal of funding from a variety of sources. Projects concerned with data sharing include: AstroGrid [64], which was predicated on producing a virtual observatory by enabling many data centres to provide competing and co-operating data services; Enabling Grids for E-sciencE (EGEE) [63], which is a large multi-science Grid infrastructure, federating some 250 resource centres world-wide, providing some 40,000 CPUs and several Petabytes of storage; and Exploitation of Switch Lightpaths for e-Science Applications (ESLEA) [72, 71], which investigated the use of switched lightpath networks to provide the necessary infrastructure for sharing ever increasing amounts of data. Typically, in such contexts data owners wish to share their data so they can perform queries across much larger data sets, to enable them to test their hypothesis, or simply to get access to more interesting data sets.

Recently it has become common to describe large distributed systems that support the sharing of data and computational resources by the term *grid* [42] or *cloud* [15]. Although cloud computing makes use of most of the same components as grid computing, cloud computing is primarily geared towards commercial enterprises. The commercial focus of cloud computing results in white papers and other commercial literature being used for the dissemination of information regarding cloud computing. For the remainder of this discussion we will use the term grid to also

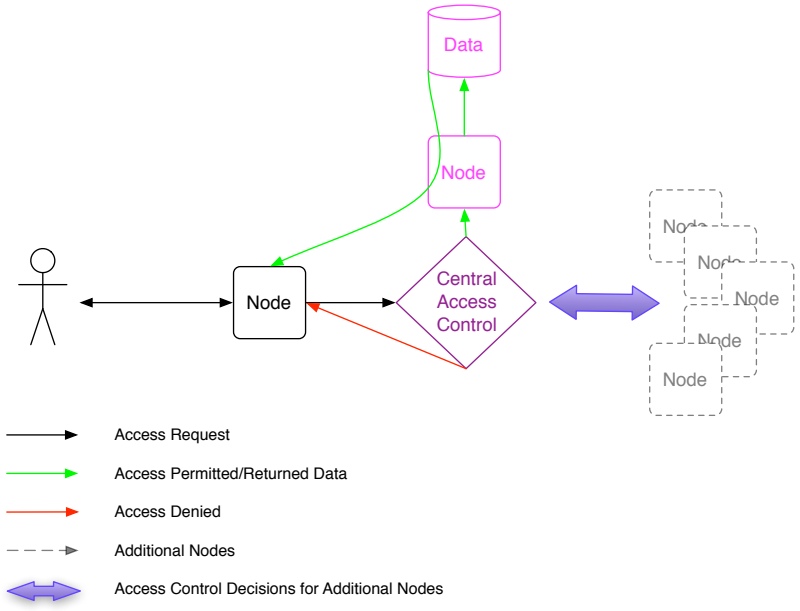

Figure 1.2: Centralised access control in a distributed system

encompass the similar concepts of cloud computing. In this terminology a distributed system designed for the sharing of data is termed a *data grid*. The data can be shared either within an organisation or across organisational boundaries. When disparate organisations come together to facilitate the sharing of data across organisational boundaries, a *virtual organisation* (VO) is created.

However the data is obtained, one of the key issues that is always present is that of access control. In general terms, access control is about permitting authorised entities to make use of resources as well as preventing unauthorised entities from gaining access to resources. For example in Figure 1.1 a user makes a request to access a resource (black line). This request is processed by an access control system which will either reject the request (red line) or accept the request and pass it on to the resource (green line), which will, subsequently, perform the actions specified by the request.

The part of the access control mechanism which enforces the authorised access relationships between the subjects and resources of a system is encapsulated in the concept of a *reference monitor* [13]. The implementation of a *reference monitor* is termed a *reference validation mechanism* [13].

When we consider distributed data there can be additional complications. Distributed access control can essentially be provided in three ways: centralised access control — illustrated in Figure 1.2; fully distributed access control — illustrated in Figure 1.3; or via a hybrid approach where there is a central access control mechanism with local policies also being used — illustrated in Figure 1.4.

One thing that a data owner may want to know is 'who is able to see what data?' While this may appear a simple objective, it is not always easy or even possible to achieve. When a

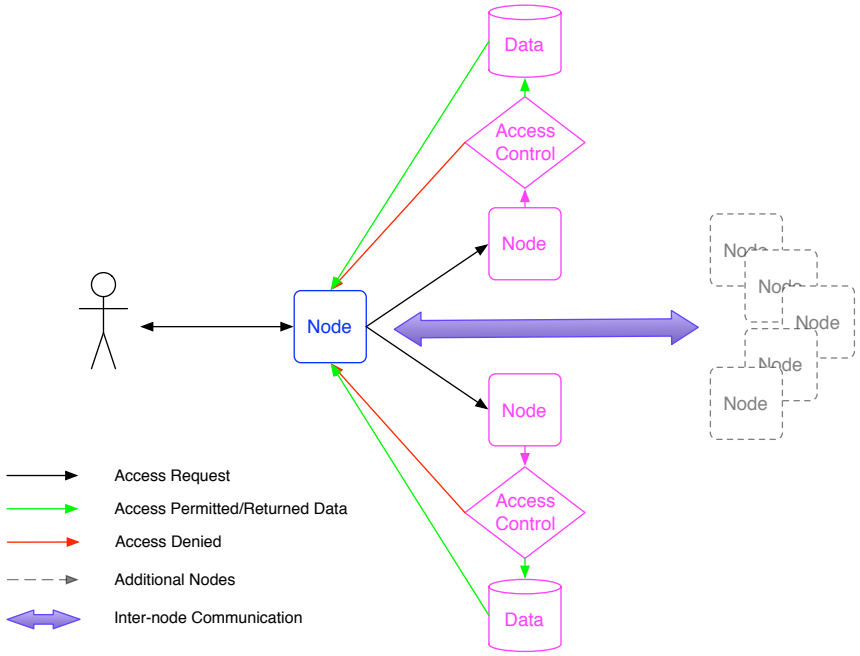

Figure 1.3: Distributed access control

simple local system or a fully centralised system is considered, it is usually possible to ascertain who can access what data. However, when a fully decentralised or a hybrid access control system is used to secure the data, it is not generally possible to get a global picture of who can see what data. Additionally, the number of possible ways of aggregating data further complicates the overall picture as it is potentially possible to get the same information from different sources and combine it in a way that was unintended.

Each pairing of data sources is likely to yield different results for each user requesting federation depending on who the user is and the location that the user is associated with. This is because, in a distributed access control system, the data owners can specify the policies and can vary the access on whichever criteria they deem appropriate.

When considering the sharing of distributed data across multiple administrative domains it is not uncommon for the data owners to have chosen to use different access control paradigms. When this is the case, there will be some access control policies expressed in a different representation than that required by the middleware facilitating the aggregation of the data sources. This leaves an unresolved problem pertaining to the translation of access control policies from one representation to another to facilitate the use of middleware to enable the sharing of data sources — which, inevitably, can lead to a loss of confidence in this respect. Through our work we have found a general reluctance on the part of data owners to rewrite access control policies using a different representation; there is also a tendency to express a general concern over performing ad-hoc translation between the differing access control representations.

We would argue that the most appropriate way to solve this problem is to provide a reliable and robust translation mechanism that the data owners can have confidence in. The use of

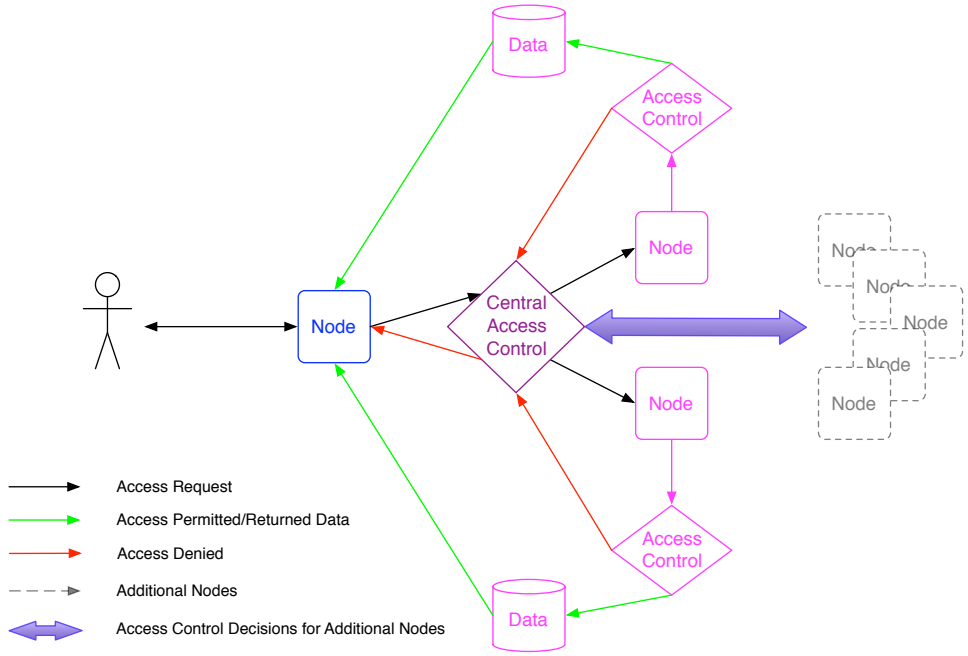

Figure 1.4: Hybrid access control

formal methods to underpin the translation process has been chosen as the most appropriate way of achieving the goal of providing reliable translations between access control paradigms with some degree of assurance of correctness. The process of developing the formal models of the access control paradigms has enabled a greater understanding of each access control paradigm as well as providing a mechanism for highlighting any ambiguities or inconsistencies that may exist. The use of complementary approaches (discussed in more detail below) allows for a conceptually simple model which can be type-checked, together with a slightly more abstract model that can be model-checked to provide a level of confidence that the overall intent is met.

With the removal of any ambiguities it is then possible to make assertions about the meaning of a particular policy and have more confidence in any translations that are defined. Furthermore it allows us to reason about the combination of policies. In addition, the use of model checking tools provides further confidence in the formal model of the translation process. The modelling representations chosen are Z [100, 109, 53], which has been used to provide a formal description of the access control paradigms of interest along with a formal description of the translation process, and Alloy [55], which has been utilised to perform model checking within a finite scope by finding a valid model which meets the specification.

The combination of Z and Alloy has been used in this work as Z is both expressive and accessible, but has the drawback of having limited tool support, while Alloy — although less expressive — does provide a means of gaining confidence in a model for a particular scope without formally proving it to be correct. While there is limited tool support for Z, all of the models presented in the following have been checked using the type checker supplied with the CZT toolkit [1] and Fuzz [101].

The work in this thesis aims to provide a reliable approach to converting access control policies between different paradigms based upon formal models that maintain the intent of the original access control policy when considering requests for access to resources. Thus, the primary research question driving the work in this thesis is:

> *Is it feasible to produce a framework, based upon formal modelling and analysis techniques, to facilitate the automated translation of policies between different access control representations while maintaining the intent of the original policy?*

It would be impractical to provide translations between every possible access control representation; as such, so we have chosen two more commonly used representations to use as a proof of concept of the approach. The access control representations used are role-based access control (RBAC) and eXtensible Access Control Markup Language (XACML). A key reason for this choice is the difference in expressibility between the representations: RBAC is able to capture simple access control policies and, conceptually, is easy to understand; XACML allows the capturing of a wide spectrum of access control policies — from the simple to the complex — but at the cost of being verbose and difficult to read.

To provide additional validation of our models and processes, a policy construction tool has also been developed. The application has been developed to ensure that the constraints and relationships defined by the formal model are captured. This application is tested by real world use cases which have been derived from requirements provided by collaborators from the Generic Infrastructure for Medical Informatics (GIMI) project [91, 93]. The access control model of choice for the exemplar developers was RBAC, whereas the middleware developed utilised XACML. It is the policies provided by these developers that were used to validate the translation process.

We have published the following papers as a result of this thesis:

- [85], which was joint work with Power and Simpson, provides a formal description of RBAC, which is covered in more detail in Chapter 4.
- [98] provides a formal model of XACML in Z, along with an early translation of core RBAC to XACML which are expanded in Chapters 5 and 6 respectively.
- [86] details the use of an application to capture RBAC policies and check they conform to additional predicates over and above those imposed by RBAC. Chapter 7 expands upon this and includes details of additional functionality concerned with translating RBAC policies into XACML policies.

The structure of the remainder of this thesis is as follows. In Chapter 2 we provide the motivation for this work, along with general background including a brief overview of different access control paradigms. Chapter 3 outlines the requirements that we have used to guide our work, as well as the middleware framework that supports the use cases and applications. The following two chapters, 4 and 5, give the formal description of RBAC and XACML respectively. In Chapter 6 we provide the formal model of the translation from RBAC to XACML. This chapter includes a Z model along with a walk-through of a sample translation. In addition, an Alloy model of the translation process is provided. Chapter 7 then provides an overview of the prototype application which has been developed, based on the formal models, to facilitate

translation between RBAC and XACML for policy writers. The translation application described in Chapter 7 is then used to support a case study in Chapter 8. Finally, in Chapter 9, we discuss the contribution of this thesis, provide a critique of the tool support available for this type of work, and outline possible future avenues of research that build upon this work.

Chapter 2

Background

2.1 Outline

This chapter presents an overview of the wider context in which this work has been carried out and provides brief details of the system that has been used to test and validate this research. The primary aim of this chapter is to provide general background information pertinent to the work presented in this thesis, rather than provide an exhaustive and critical evaluation of access control and the other topics covered. It is assumed that the reader is familiar with the formal method Z [100, 109], but is not familiar with Alloy [54, 55].

First, a brief overview of access control is provided. This initially covers the classic ideas of mandatory and discretionary access control including some of the traditional techniques for capturing access control information. This is followed by a discussion of access control in the distributed context. An outline of RBAC and XACML, the two access control representations which are being used to validate this work, then follows. Next a brief discussion of how formal methods have previously been applied to access control is included.

An introduction to the Alloy language and the Alloy Analyzer is then provided as this is used to provide assurance in the translation techniques that have been developed.

Finally the use of Service Oriented Architecture (SOA) techniques and Web Services (WS) to underpin distributed systems is discussed. Included within this section is a description of the middleware developed during the aforementioned GIMI project, known as service-oriented interoperability framework (sif) [92, 97]. The case study in Chapter 8 utilises an application that runs on the sif middleware.

2.2 A brief introduction to access control

This section gives a brief introduction to access control, which can be implemented in different ways depending on the nature of the artefact being protected. Historically, the access being controlled was to physical places or physical artefacts, with the placement of guards on entrances and bars on windows being typical of this type of physical access control.

In the modern age, however, access control has become a more complex issue; although it is still necessary to control physical access to systems, of course. This is because once physical

access has been gained a lot of other controls can be circumvented relatively easily. The physical removal of hardware to another location can enable even a password-protected system to be compromised.

In Information security, access control can be thought of as a combination of authentication and authorisation. In addition to these aspects, there may be an audit mechanism.

The authentication mechanism is designed to check that the requester is who they claim to be. The requester usually provides evidence that is used to uniquely identify them. The authentication mechanism then validates that the requester is who they claim to be.

The authorisation mechanism is then used to determine if the authenticated requester has the rights to perform a particular task. There is no fixed set of permitted actions that an arbitrary authorisation mechanism must deal with. However, it is likely that an authorisation mechanism would be able to deal with the three basic types of access: Read, Write, and Execute.

For the remainder of this thesis, the term *access control* should be taken to refer to authorisation processes and mechanisms.

The main focus of this work is the definition and implementation of access control to software-based computer systems. In this context, access control is a set of constraints that limit the access a user has to a computer system. Therefore, the access control policies are the method for defining what a user can do directly. In addition to limiting the direct actions of a user, it should also limit the actions of applications executing on the user's behalf. This will allow control to be maintained over the use of applications and consequent exposure of data.

There have been many proposals relating to access control mechanisms and what types of access control are available over the years. Some of these are described within the following subsections.

2.2.1 Mandatory access control

Mandatory access control (MAC) describes access control policies which are mandated by a central authority and enforced independently of the actions of a user. Typically a mandatory policy takes the form of a *multilevel security policy*, which is based on the assignment of classifications to *subject* and *objects*.

An example of MAC concerned with confidentiality was first represented by the Bell-La Padula model [21]. This style of MAC can be described as a multi-level security system suitable for military environments. In the model defined in [21], access to an object is based on two classifications: the classification of the object, represented by a security label; and the clearance classification of the user, also represented by a label. Typical values for the labels and the ordering of the labels are: $unclassified \leq confidential \leq secret \leq topsecret$. The policy enforcement is then based on the following two principles.

- No Read Up: this principle states that a user has read access to an object if they have a clearance *greater than or equal* to the object's classification.
- No Write Down: this principle states that a user can write to an object only if they have a clearance *less than or equal* to the object's classification.

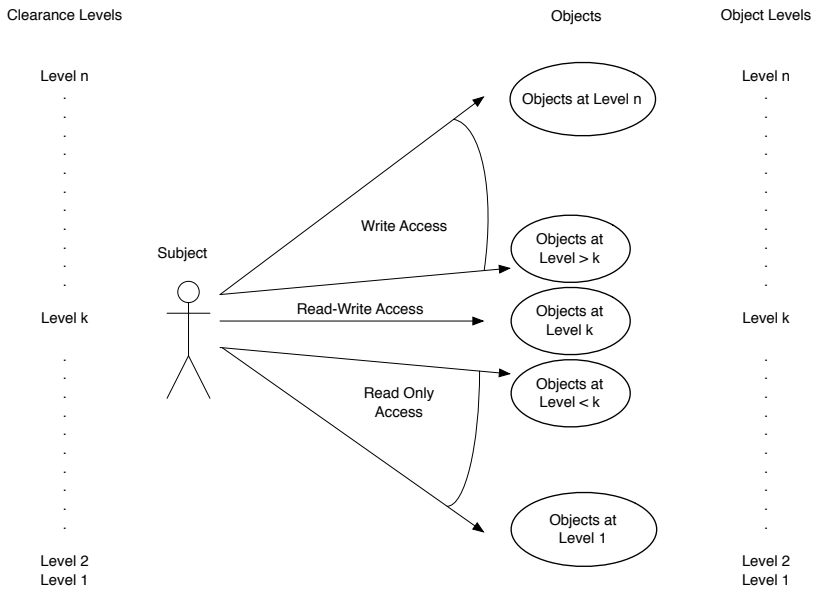

Figure 2.1: Bell-La Padula read up write down

The enforcement of the above two principles forces information flow to be one way. The second of the two principles prevents the accidental or malicious declassification of information by a user. This is illustrated in Figure 2.1, which shows that a subject with clearance level k can read and write to objects with an object level of k. However, they can only append to objects with a level $> k$ and only read objects with a level $< k$.

While the Bell-La Padula model is designed to protect the confidentiality of information by preventing the unauthorised release of information, it does nothing to prevent the unauthorised modification. The model proposed in [23] applies similar principles to protect the integrity of the information by assigning each subject and object an integrity level.

From the above description, it is clear that a MAC system is very restrictive and rigid. This may be suitable for the military or military-like environments, where a fixed set of labels and two access rules meet the requirements. A typical organisation, however, will normally need a more flexible system of access control.

2.2.2 Discretionary access control

Discretionary access control (DAC) is typically defined as a means of restricting access to objects based on the identity of the user and/or the groups to which they belong. The discretionary part of the control is the ability of the user to pass on an access permission they have to another user. A DAC system specifies that the user owns objects and has the ability to manage the access to the objects themselves. The access to a particular object can be passed on directly or indirectly.

Access control matrices

Traditionally, access control systems can be conceived as a simple matrix of access rights. An early example of this representation is described in [62]. The simple matrix format for access control description usually provide consideration of (S, O, A) where S is a set of subjects, O is a set of objects, and A is a set of access rights. The access control matrix provides an explicit lookup of the access rights that any subject has to any object.

An access control matrix is severely limited as it does not allow for complex requirements. In addition, access control matrices are an inefficient method of controlling access and have rarely been implemented in real systems. It becomes increasingly impractical as a system when there are a large number of objects and subjects to consider. An access control matrix is more normally utilised as a way of visualising access control that has been implemented as an access control list (ACL) or a capability list. An ACL is essentially a column view of the matrix with a list being associated with each object, with each element of the list indicating a subject and their rights on the object; a capability list is essentially a row view of the matrix with a list being associated with each subject, with each element of the list indicating an object and the privileges held by the subject.

Unix access control

The default unix access control mechanism is an example of a DAC. Each object within the unix file system has three collections of permissions associated with it. The collections of permissions relate to the owner of the object, a group associated with the object and the rest of the world. Each collection of permissions is made up of three elements. These elements relate to Read-/Write/Execute permissions. It is therefore possible to define an object's access to be limited to just the owner, the owner and group or the owner, group, and world.

The use of ACLs is an enhancement to standard unix access control. An ACL allows the definition of finer control over the access to objects than is possible with traditional unix access control. ACLs have been implemented in one form or another in many modern operating systems. They allow for the access to files to be controlled in a flexible way, allowing individual users to be granted different rights to the same object.

2.2.3 Access control in distributed contexts

Even when considering access control for a simple independent system, it can sometimes be difficult to obtain a clear picture of who has the authority to perform particular operations. When access control in a distributed context is considered, this difficulty can be greatly exacerbated, especially if each of the systems utilises a different access control paradigm. To facilitate analysis of the combination of policies from these different representations, it would be useful to be able to translate them into a common format. The need to consider access control in a distributed environment has been recognised for a number of years, with analysis primarily being based on the assumption that a single access control paradigm has been used.

Early recognition that little work had been done when considering access control in collaborative environments was highlighted in [90], which describes potential requirements and a model for capturing access control in a collaborative context. Other early work includes 'a calculus for

access control in distributed system' [5], which presents the logical components necessary for an access control list to be utilised for deciding which requests should be granted.

The provision of access control for collaborative virtual environments is considered in [28]. The proposed approach is predicated on the concept of boundaries, clearance classifications and graph theory. In this approach, permissions are not associated with subjects and objects but instead they are associated with the space that objects and subjects inhabit. This means the access rights to an object depend upon where the object is located and if the subject can access the space. In general, a boundary is given a classification which dictates the clearance level required by a subject to traverse the boundary. Access graphs are then used to represent the different routes that can be followed through the regions of space based on clearances. This work is extended in [29], which also considers in more detail the general requirements for collaborative access control.

More recent work includes that of [35], which considers the specification of policies for management of networks and distributed system using the Ponder language. The ability to describe reusable components representing organisational units which facilitate the creation of composite specifications for complex organisations. Cassandra, described in [20], provides a small number of constructs which permit the description of complex policy ideas such as delegation.

The design and semantics of SecPAL, a decentralised authorisation language, is covered in [18, 19]. An execution strategy based on translation to Datalog [7] with constraints is covered. The mechanism is essentially based on assertions, which describe a set of conditions which are sufficient to generate a fact.

An approach to access control in a collaborative environment based on user tagging is considered in [106], where it is proposed that users tag each other with the terms they want. Access to resources is then controlled by the resource owner writing policies that are evaluated against the tags users have. The policy definition and evaluation process allows for varying levels of control over the resource from limiting access to only those people directly tagged by the resource owner right through to allowing the tag of interest to have been applied by anyone in the system. In [106] it is stated that the "system is designed for enterprise settings, where every person can have only one user account and most people are honest."

2.3 RBAC

2.3.1 Outline

RBAC has emerged as a popular access control paradigm, both within the research community and among information technology vendors, because it offers support for flexible approaches to authorisation in a manner that is both accessible and manageable. In 2004 an ANSI standard for RBAC [14] was approved. The rest of this section provides a general overview of RBAC along with details about the different types of inheritance that are possible. In addition, comments on some of the limitations of the ANSI standard are also included.

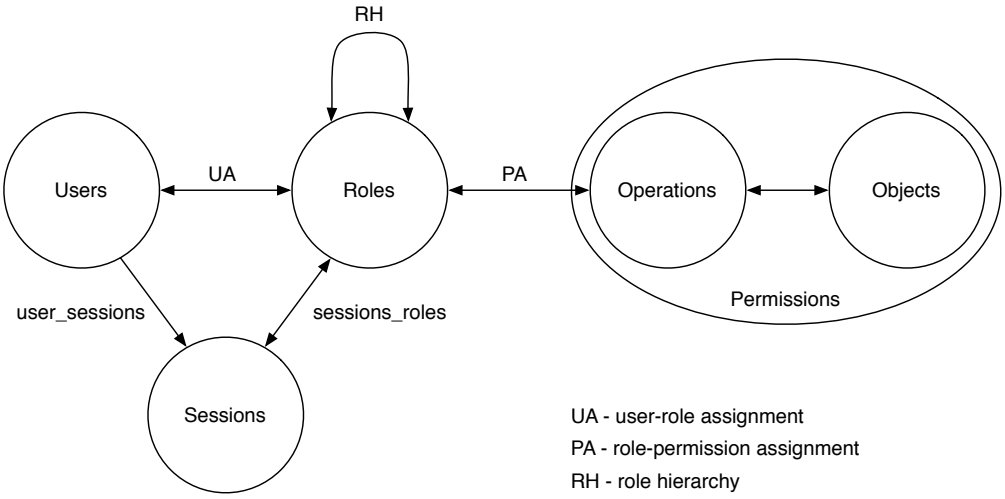

Figure 2.2: RBAC conceptual diagram after [79]

2.3.2 Principles of RBAC

The underlying principle upon which role-based access control is based is the association of permissions with the roles that users may hold within an organisation. The relationships between RBAC's concepts are captured in Figure 2.2.

The term *functional role* was introduced in [36], while the notion of *named protection domains*, which could be organised into hierarchies, was introduced in [16]. Other early role-based systems included [103], [25], and [74].

A NIST study [38] discovered that commercial and governmental access control requirements were not being met by available technologies, with both mandatory access control and discretionary access control models having drawbacks.

A generalised RBAC model was proposed in [39], in which the following definitions are important:

- for each subject, the active role is the one that the subject is currently using;
- each subject may be authorised to perform one or more roles; and
- each role may be authorised to perform one or more transactions.

Further, three rules are required [40], which may be stated in the following fashion.

- **Role assignment**: A subject can execute a transaction only if the subject is associated with a role.
- **Role authorisation**: A subject's active role must be authorised for the subject.
- **Transaction authorisation**: A subject can execute a transaction only if the transaction is authorised for the subject's active role.

Here, a role is considered to be a collection of permissions, with all users being granted permissions only through the roles to which they have been assigned. It is assumed that users and permissions change often, while roles change less often.

2.3.3 The ANSI RBAC standard

In 2004 the American National Standards Institute (ANSI) approved an RBAC standard [14]. In the foreword of the standard, we are told the following.

> "In recent years, vendors have begun implementing role-based access control features in their database management systems, security management and network operating system products, without general agreement on the definition of RBAC features. This lack of a widely accepted model results in uncertainty and confusion about RBAC's utility and meaning. This standard seeks to resolve this situation by using a reference model to define RBAC features and then describing the functional specifications for those features."

The time-line for the development of the standard was as follows.

1. The initial draft of the standard was proposed at the 2000 ACM Workshop on RBAC [89].
2. A panel was held at the ACM workshop, with comments being published in the proceedings [57].
3. A second version of the standard appeared in ACM Transactions on Information and Systems Security in 2001 [41].
4. The final version of the standard was approved in February 2004 and subsequently published as ANSI INCITS 359-2004 [14].

There are four standard components in the ANSI standard for role-based access control systems:

- *Core RBAC* is mandatory in any RBAC system.
- Any combination of *role hierarchy* (which may be general or limited—but not both), *static separation of duty*, and *dynamic separation of duty* can be utilised in a particular RBAC system.

We consider each of these components in turn.

Core RBAC

Core RBAC relates permissions, roles, and users: permissions are associated with roles, and roles are assigned to users.

Role hierarchy

Role hierarchies define what amounts to an inheritance relation between roles. As an example, role r_1 inherits from role r_2 if all privileges associated with r_2 are also associated with r_1.

Static separation of duty

A static separation of duty (SSD) constraint is characterised by a role set, rs, such that $\# rs \geq 2$, and a natural number, n, such that $2 \leq n \leq \# rs$, and ensures that no user can be authorised for n or more roles in rs.

An example where separation of duty could be necessary is in an accounts department, where there is a requirement that no user can have the ability to both raise and approve purchases. In this case, the roles *purchaseRaiser* and *purchaseApprover* should not be assigned to any individual. To achieve static separation of duty for these roles, it would be necessary to define $rs = \{purchaseRaiser, purchaseApprover\}$ and $n = 2$, which would prevent any user being assigned both roles.

Dynamic separation of duty

A dynamic separation of duty (DSD) constraint differs from a SSD constraint in that it is concerned with *sessions*. Again, given a role set, rs, such that $\# rs \geq 2$, and a natural number, n, such that $2 \leq n \leq \# rs$, a DSD constraint ensures that no user can be associated with n or more roles in rs in a particular session.

Considering the example above in a dynamic separation of duty context, with the same rs and n, no user is permitted to have the both roles activated simultaneously within a session. However, in contrast to the static separation of duty example, a user may be assigned any or all roles from rs. Considering the dynamic case, for the example given above, it is possible for a user to activate the role *purchaseRaiser* during a session, then deactivate the role *purchaseRaiser* and subsequently activate the role *purchaseApprover*.

2.3.4 Role inheritance

In [66], Suggestion 5 (of eight suggestions for improving the ANSI RBAC standard) is stated as "the semantics of role inheritance should be clearly specified and discussed." This section considers different ways role inheritance may work and the different types of limited hierarchies that could be specified.

There are three basic ways of viewing inheritance in a role-based environment:

- *User inheritance* involves a user, u, inheriting all of the roles junior to all of the roles that u is explicitly associated with.
- *Permission inheritance* pertains to the relationship between permissions and roles. With permission inheritance, a role is associated with the permissions of all roles junior to it—as well as its own.
- *Activation inheritance* is only of importance when dealing with sessions. Whenever a role is activated within a session, all junior roles must also be activated.

It is also possible to put additional constraints on a hierarchy to form specialisations, two of which are:

- A *downward limited hierarchy* is a role hierarchy in which each role defined within the hierarchy may be senior to at most one other role, but can be subordinate to many roles.

- An *upward limited hierarchy* is a role hierarchy in which each role defined within the hierarchy may be subordinate to at most one other role, but can be senior to many roles.

2.3.5 Criticisms of the ANSI standard

The first proposal for a standard on RBAC [89] prompted a short rebuttal by Jaeger and Tidswell in [57]. The rebuttal presented three main problems with the proposal. First, the proposed RBAC model consisted of four levels which build upon each other when in fact the concepts are orthogonal. Second, the model limits the expression of hierarchies and constraints which could lead to hierarchy design flaws such as those outlined in [70]. Third, some important concepts are missing from the model such as an administrative role hierarchy. Additional general comments included that several of the concepts were vaguely-specified and open to a large number of interpretations. The final version of the standard [14] contained a number of logical flaws and shortcomings. A comprehensive overview of these issues is given in [66]. The only one of their criticism we are particularly concerned with is: "The core RBAC component includes the notion of sessions, which is not essential to RBAC and does not exist in many important RBAC-based security products." The formal model which is used to underpin the rest of this thesis has opted to move the concept of sessions from the core.

More recently, [37] also included a section highlighting some of the anomalies in the RBAC standard. Some of the anomalies identified are inconsistencies in the names of variables throughout the standard along with poorly specified operations which have definitions which fail to update all the affected state variables.

2.3.6 RBAC alternatives and extensions

In [22] a temporal extension to RBAC, Temporal-RBAC (TRBAC), is proposed. The proposed TRBAC supports additional features such as the periodic enabling and disabling of roles and the ability to specify temporal dependencies between such actions defined as role triggers. In addition, it provides the facility for individual exceptions to be defined, which can tailor the enabling and disabling of roles to individual users.

The authors of [8] suggest that RBAC has several limitations that they propose to overcome with Organization-based access control (ORBAC). ORBAC is a model of security policies that is based on the concept of organisations and how actions, subjects, objects and contexts are related to each other within organisations. For instance, they have the idea that a *subject* is *employed* by an *organisation* in a *role*. The various relations between the different components are combined to give permissions, prohibitions, obligations and recommendations. This work was extended in [34], where an access control language based on ORBAC is presented that permits the definition of network access control policies and [87], which considers network security policies using ORBAC in the attack graph analysis framework.

A model-driven approach to access control is proposed in [17], which uses UML. The authors initially describe the modelling language and an extension to UML, SecureUML, which provides a visualisation component. The adoption of this approach is to encourage the thinking about security requirements when using a model-driven approach to software engineering. The access control policies defined are generalised RBAC policies.

The UCON$_{ABC}$ method of modelling usage control described in [81] is made up of several components that are used to reason about access of objects by subjects. It is an extension of UCON, presented in [80], adding the components: Authorisation (A), which relate subjects rights to objects, Obligations (B), which define mandatory actions a subject must perform before or during usage, and Conditions (C), which are environmental or system-based decision factors. An example of how RBAC could be modelled using UCON$_{ABC}$ is also included in [81], along with example of models of other access control mechanisms.

Another authorisation mechanism that uses a role-based approach is the PrivilEge and Role Management Infrastructure Standards (PERMIS) [30, 31]. In addition to the standard RBAC concepts, PERMIS also adopts concepts from the ISO Attribute Based Access Control (ABAC) Model [52], with support for obtaining attributes from many sources such as X509 certificates, Lightweight Directory Access Protocol (LDAP) directories, and Security Assertion Markup Language (SAML)[76]. PERMIS is then able to adopt a distributed approach to the assignment of roles and attributes via the use of trusted attribute authorities.

2.4 XACML

XACML (eXtensible Access Control Markup Language) [73] is establishing itself as a popular standard for enforcing access control within distributed service-oriented contexts. This section introduces XACML. It starts by providing an overview of the language before considering how access control policies may be described using XACML.

2.4.1 XACML: an overview

XACML is an OASIS (Organisation for the Advancement of Structured Information Standards) standard that allows one to describe—in terms of XML—two languages for a particular application.

The first language—the policy language—is used to describe general access control requirements. The policy language has standard extension points that allow one to define aspects such as new functions, data types, and logics to combine such entities.

The second language—the request/response language—allows one to construct a query to determine whether a particular action should be permitted. Every response contains an answer pertaining to whether the action should be permitted in terms of one of four possible values: *permit*, *deny*, *indeterminate*, or *not applicable*. The first two of these values are self-explanatory. The indeterminate value denotes the fact that a decision cannot be made due to a missing value or the occurrence of an error. The not applicable value denotes the fact that this particular question cannot be answered by this particular service.

The policy language aspect of XACML consists of several core components: *rules*, *policies*, and *policy sets*. Each of these core components has particular elements that allow a variety of access control policies to be defined.

First, each of the core components has a *target*, which is the primary mechanism for deciding if an instance of a component is to be evaluated.

The other elements of the various core components differ:

- in addition to a target, a rule also has a *condition* and an *effect*;
- in addition to a target, a policy also has a *rule-combining algorithm*, a collection of rules, and potentially a collection of *obligations*; and
- in addition to a target, a policy set also has a policy-combining algorithm, a collection of policies and/or policy sets, and (potentially) a collection of obligations.

Typically, an application or user will wish to perform some action on a particular data resource. In order for this action to occur, a request to perform it is made to the entity that *protects* that resource. This *policy enforcement point* (PEP) might be, for example, a file system or a web server. On the basis of several aspects — the requester's attributes, the resource that the requester wishes to perform its action on, what the action is, and so on — the PEP will form a request. This request is then sent to a *policy decision point* (PDP). The PDP analyses the request, together with a policy that applies to the request, and determines the appropriate answer with respect to whether or not the request should be granted. This answer is returned to the PEP. The PEP may then either allow or deny access.

The physical and logical relationships that exists between the PEP and the PDP are not prescribed: they may sit within a single application on the same machine; they may sit within a single application across different machines; or they may exist in different applications on the same machine.

2.4.2 Describing access control in XACML

The data flow of XACML is illustrated in Figure 2.3 [73].

The access requestor sends a request to the PEP via an *access request*. A request is then forwarded to the *context handler*, which includes any additional information relating to the subject, action, resource or environment that is necessary. The context handler then formulates an XACML request and passes it to the PDP via a *request notification*. The *policy access point* (PAP) makes the policies and policy sets available to the PDP. If any additional subject, resource, action or environment attributes are required then the PDP requests them from the context handler. The context handler in turn requests the additional attributes from the *policy information point* (PIP). When the PIP has obtained all the requested attributes, it then returns them to the context handler. The context handler sends the requested attributes to the PDP. The PDP is then able to evaluate the policy. The PDP returns the result of the evaluation via the *response context* to the context handler. The context handler translates the response context to a format appropriate for the PEP. The context handler then returns the response to the PEP. The PEP fulfils any obligations before acting on the response returned. If the response indicates that access is permitted, then the PEP permits access to the resource; otherwise, it denies access.

The following considers the make-up of XACML policies in more detail.

A policy can have any number of *rules* — it is rules that contain the core logic of an XACML policy. At the heart of most rules is a *condition*, which is a Boolean function. If the condition evaluates to true, then the rule's *effect* — which is the intended consequence of a satisfied rule (either 'permit' or 'deny') — is returned. In addition, a rule specifies a *target*, which defines:

- the set of the *subjects* who can access the resource,

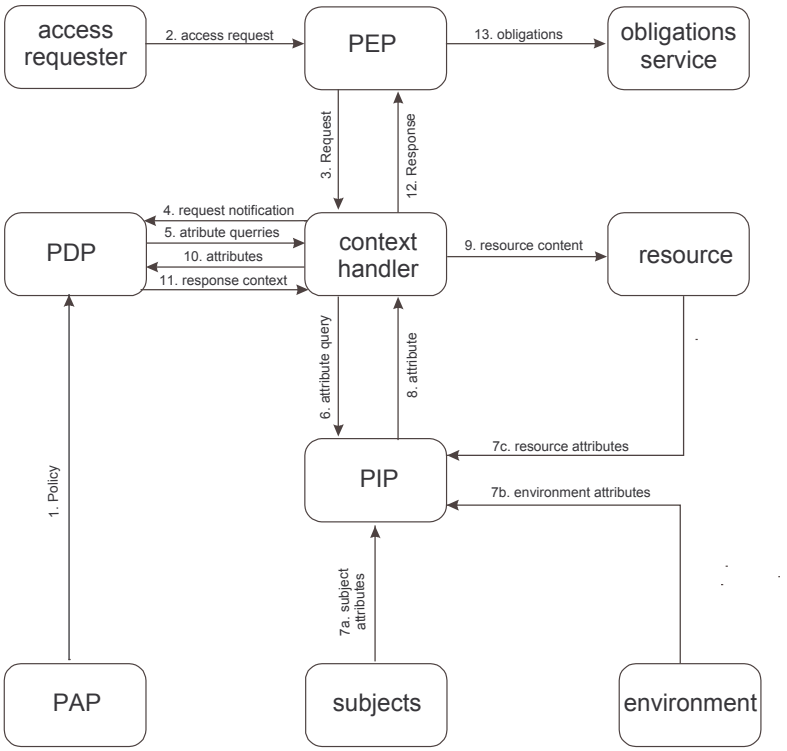

Figure 2.3: XACML data flow [73]. Copyright OASIS Open 2004. All Rights Reserved.

- the *resource* that the subject can access,
- the *action* that the subject can undertake on the resource, and
- the *environmental* attributes that are relevant to an authorisation decision and are independent of a particular subject, resource or action.

Within the request document there exist values of specific attributes that must be compared with the corresponding policy document values in order to determine whether or not access should be allowed. The values of the request attributes are compared with those of the policy document so that a decision can be made with respect to access permission. In the situation where many different policies exist — as opposed to the case in which a single policy document exists — a *policy set document* is defined as a combining set of many policies.

A policy or policy set may contain multiple policies or rules, each of which may evaluate to different access control decisions. In order for a final authorisation decision to be made, combining algorithms are used, with *policy-combining algorithms* being used by policy sets and *rule-combining algorithms* being used by policies. Each algorithm represents a different means of combining multiple decisions into a single authorisation decision. An illustration of different combining algorithms is presented in Figure 2.4.

The authorisation decision in relation to a subject requesting permission for an action on a resource in an environment can take one of four aforementioned values: permit, deny, indeterminate, and not applicable.

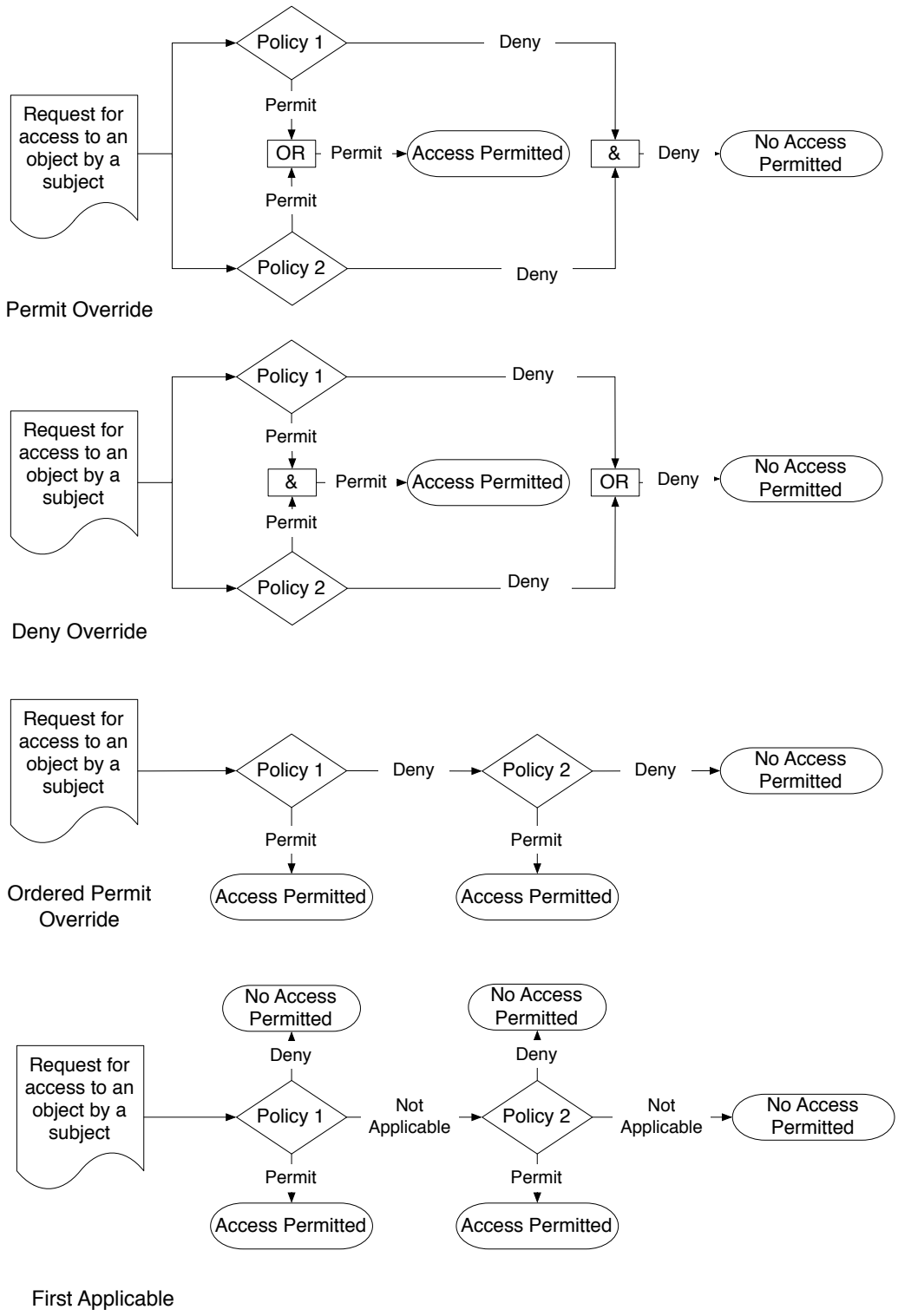

Figure 2.4: Effects of different combining algorithms

```
.
.
.
1   <Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policycs-xacml-schema-policy-01.xsd"
4     PolicyId="read.prescribeDB"
5     RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
6     <Target>
7       <Subjects><AnySubject/></Subjects>
8       <Actions><AnyAction/></Actions>
9       <Resources><AnyResource/></Resources>
10    </Target>
11    <Rule RuleId="read.prescribeDB" Effect="Permit">
12      <Target>
13        <Subjects><AnySubject/></Subjects>
14        <Actions>
15          <Action>
16            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
17             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
18              <ActionAttributeDesignator
19                DataType="http://www.w3.org/2001/XMLSchema#string"
20                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
21            </ActionMatch>
22          </Action>
23        </Actions>
24        <Resources>
25          <Resource>
26            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
27              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">prescribeDB</AttributeValue>
28              <ResourceAttributeDesignator
29                DataType="http://www.w3.org/2001/XMLSchema#string"
30                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
31            </ResourceMatch>
32          </Resource>
33        </Resources>
34      </Target>
35      <Description></Description>
36    </Rule>
37  </Policy>
.
.
.
```

Figure 2.5: A fragment of XACML

When the PDP compares the attribute values contained in the request document with those contained in the policy or policy set document, a response document is generated. The response document includes an answer containing the authorisation decision. This result, together with an optional set of *obligations*, is returned to the PEP by the PDP.

Obligations are sets of operations that must be performed by the PEP in conjunction with an authorisation decision; an obligation may be associated with a positive or negative authorisation decision.

Figure 2.5 is a fragment of an XACML policy file. It illustrates a simple policy which contains a single rule. The policy is identified on line 4 by the `PolicyId` of `read.prescribeDB`. Line 5 indicates that the policy would deal with the results of multiple rules by using the `rule-combining-algorithm:permit-overrides`. Lines 6 to 10 detail the applicability of the policy, and indicate that the policy is universally applicable for all subjects actions and resources. The policy contains a single rule from lines 11 to 36. The content of line 11 identifies the rule as having a `RuleId` of `read.prescribeDB` and an `Effect` of `Permit` if the rule is applicable. The target section of the rule, lines 12 to 34 inclusive, indicates the applicability of the rule. In this case the rule is only applicable if the action is `read` — found on line 17 of the `Action` section

from line 15 to line 22 — and the `Resource` is `prescribeDB` — found on line 27 in the resource section from line 25 to line 32.

A number of extensions to XACML have been developed by OASIS including a profile for describing RBAC policies using XACML [75], which allows RBAC policies to be described but requires the implementation of an external service to resolve the mapping between subjects and roles. An extended version of the XACML RBAC profile has also been proposed in [6], which aims to support additional extensions such as delegation.

In addition, [59] proposed extensions to XACML to provide collaborative access control based on the encoding of an RBAC hierarchy. The concept of collaboration inheritance is introduced which controls the way in which hierarchical permissions are delegated based on privileges and the assessment of right misuse. Other requirements noted are the idea of trust and consideration if the delegation privilege is exposed by the delegated role.

It is worth noting that in [12] the origins of the basic building blocks of XACML are commented on, primarily that XACML uses the abstract model for policy enforcement defined by the IETF [107, 111] and ISO [52], with the XACML PEP being equivalent to the Access Enforcement Point in ISO model.

Some more practical consideration has also been given to XACML. In [68], consideration is given to the use of XACML to provide authorisation in a distributed, decentralised system. The use of various other standards to provide the aspects of a distributed authorisation system that are not specified within the XACML standard are discussed, and a number of possible architectures to meet different needs are presented. If XACML is to be used to provide access control decisions in large systems then the performance of the PDP implementation becomes an important consideration. The evaluation number of XACML PDP implementations is considered in [105], which compares the response times of the tested implementations for varying numbers of policies and rules. In addition to performance, it is also vitally important that an implementation of an XACML PDP generates the correct decision for all requests; this is considered in [67] for several implementations of PDPs.

2.5 Access control and formal methods

The formal modelling of access control paradigms and principles has a rich history spanning many years. One such early attempt at formalising access control can be seen in [102], in which a formal description of an access control system is presented in Z. The general concepts of controlling which subjects are permitted to perform what actions on the resources being protected can be expressed in many different ways, some of which are illustrated in the earlier parts of this chapter.

There are numerous different access control paradigms that can be utilised to describe authorisation policies, with each paradigm potentially offering a completely different method for describing the relationship between subjects, actions and resources. These differing representations potentially have very different cognitive requirements when describing the policy and trying to understanding the meaning of a policy. The manageability of complex access control policies, particularly within safety-critical applications, is considered in [56] with a number of metrics being proposed to calculate the 'authorisation complexity' of a particular configuration and providing a means of balancing any complexity against additional costs that may be incurred when

considering the aggregation or inheriting of permissions.

When RBAC and XACML are considered, there is a clear difference between the cognitive ability necessary to understand and reason about policies written in each of the representations. While RBAC has a limited ability to describe access control policies, the concepts underlying those policies are very easy to follow: essentially a subject is associated with one or more roles, which may or may not inherit from other roles. Each of the roles may grant permissions to perform some action on a given resource. On the other hand, XACML allows for more complex policies to be described. Each policy can combine the rules that make it up using one of many rule-combining algorithms. These policies can then be further combined into policy sets using policy-combining algorithms. Policy sets can themselves be combined within policy sets, with policies or policy sets also being combined by policy-combining algorithms. The complexity of the many possible combining algorithm combinations is then coupled with the applicability of the various rules policies and policy sets. This potentially makes it very difficult to understand the permissions that are being granted or denied to individuals for a given policy set, making it relatively easy to write a policy which has unintended consequences.

From the beginning, RBAC has been closely associated with formal and semi-formal representations, with this association being started by the RBAC standard [14] which provides a semiformal description of RBAC in a Z-like notation. Following on from this initial work, there have been numerous formal models of RBAC proposed. The critique of the standard in [66] is presented in form of a semi-formal model whereas [58] presents a formal model for a subset of the RBAC standard concentrating on core RBAC without hierarchies.

More recently, RBAC and enhancements to it continue to be modelled using formal notation. The B method was utilised in [37], which enabled the identification of several anomalies in the original standard. As another example, RBAC is modelled using Z in [112], which proposes an enhancement to the separation of duty definition to permit validation of the maintenance of separation of duty at the permission level.

Equally, there have been numerous attempts to model XACML formally. Early work in this area includes the modelling of the formal semantics XACML using Haskell in [51], which aimed to provide the community with a mechanism to improve the assurance that an implementation correctly interprets the XACML specification, as well as providing a formal language to discuss and reason about any ambiguities.

We also have [26], which proposes a model of XACML using Communicating Sequential Processes (CSP) [47, 88]. The model presented in [26] excludes the first applicable combining algorithm because "this would prove much more of a challenge to describe in CSP, because the CSP parallel operators are commutative." Other formal models of XACML have been presented using techniques such as: Defeasible Description Logics in [60] and VDM++ in [27].

A description of the Margrave tool is given in [45], along with arguments relating to the soundness of the tool with respect to a subset of XACML. The Margrave tool performs analysis of a subset of XACML policies by constructing a multi-terminal binary decision diagram (MTBDD) representing the XACML policy of interest, before using binary decision diagrams to perform the analysis. A continuation of this work is described in [48], which presents a proposed method of conformance checking XACML policies using Margrave.

In contrast to the reasoning about XACML policies using Margrave, [113] and [114] introduce

a framework for describing and reasoning about access control policies along with a conversion into XACML. The framework uses the Read and Write (RW) formalism, which uses first-order formulas for modelling access control policies. The policy description can then be checked using the model checking tool which evaluates if supplied properties hold for a given model of a policy. The tool can also translate the model into an XACML policy. If the translation generates a policy which has condition tags, a custom extension of XACML is used to evaluate the condition by means of a query against an external database. To facilitate the translation the database must satisfy a number of conditions which are outlines in [114].

In [50] the use of Boolean satisfiability problem (SAT) solvers to verify properties of XACML policies is presented. This is achieved by translating the verification queries about the XACML policies into a representation suitable for utilising a SAT solver. This work extended earlier work described in [49], which utilised Alloy instead of going directly to a SAT solver.

2.6 Alloy

The Alloy language and use of the Alloy Analyzer are described initially in [54] and more fully in [55]. In [54] Jackson describes Alloy as a language for describing structural properties which can be analysed automatically. It supports two type of analysis: *simulation*, in which the validity of an invariant is demonstrated, and *checking*, where the specification is tested by way of trying to find a counter-example.

A key feature of Alloy is that everything is a relation — this includes all data types even sets, tuples, scalars and sequences. The key operator is the *dot join* which fulfils the roles of: relational join, field navigation and function application. The model defined in Alloy is analysed by a SAT solver; an early consideration of the Boolean satisfiability problem can be found in [33].

By means of a short example taken from [69] we will look at the components that make up an Alloy model. The example is a fragment of the birthday book presented in [100] shown in Z along with the Alloy translation. The model is presented interspersed with explanatory comments concerning the Alloy representation.

$$[NAME, DATE]$$

```
sig NAME, DATE {}
```

In Alloy `sig` introduces a new set of objects with the given name. In the above, the line `sig NAME, DATE {}` introduces two sets of objects with the names `NAME` and `DATE`.

BBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\, birthday$$

```
sig BBook {
    known : set NAME,
```

```
    birthday : NAME -> lone DATE
}{
    known = dom[birthday]
}
```

The Alloy `sig BBook` introduces a set of objects with the name BBook which have two fields — `known` and `birthday` — as well as a constraint on the relationship between `known` and the domain of `birthday`. This definition states that `known` is a set of `NAME` and `birthday` is a subset of the set formed by the Cartesian product `NAME` $\times$ `DATE`. As Alloy treats everything as a relation the `lone` is necessary to indicate that the relation `birthday` is from a `NAME` to at most one `DATE`. The content between the second pair of braces is a constraint on the signature. In this case, `known = dom[birthday]` states that the set `known` is exactly the domain of the relation `birthday` .

BBookInit

$$BBook'$$

$$known' = \emptyset$$

```
pred BBookInit [s' : BBook]
{
    no s'.known
}
```

The above, in Z terms, is an initialisation schema. For this, Alloy uses `pred`, which gives a name to a predicate and allows parameters. In this case, `BBookInit` has a single parameter `s'` from the set `BBook`. The body then states that the `BBook` reference must have no members in the set that comprises the field `known`.

Add

$$\Delta BBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin \operatorname{dom} known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

```
pred Add
     [s, s' : BBook,
     name_in : NAME,
     date_in : DATE ]
{
    name_in !in s.known
    s'.birthday = s.birthday + (name_in -> date_in)
```

```
}
```

Finally we look at a simple operation to add a new entry to the `BBook`. This is also defined as a `pred` in Alloy with four parameters: `s` and `s'` which are of type `BBook`, `name_in` which is a `NAME`, and `date_in` which is a `DATE`. Considering the body of the `pred` there is a guard condition `name_in !in s.known` which ensures that the supplied name `name_in` is not already in the set `known`. The final part states that the `birthday` relation in `s'` is the set of pairs from the `birthday` relation in `s` plus the new pair `(name_in -> date_in)`.

In addition to predicates, Alloy also allows the definitions of `facts` — which are constraints that always hold. An example of the use of a `fact` is given below, which ensures that nobody has two birthdays.

```
fact {
    all b : BBook |
        (all n : NAME, d1, d2 : DATE |
           not (b->n->d1 in birthday and
                b->n->d2 in birthday) or
                d1 = d2)
}
```

Alloy also provides functions that are named expressions that take zero or more arguments and return the result of evaluating the associated expression. The `lookup` function defined below returns a `Date` associated with a `NAME` in the `BBook`.

```
fun lookup (b : BBook, n : NAME) : Date {
    n.(b.birthday)
}
```

Although Alloy is less expressive than Z, it provides a useful means of validating ideas presented using the richer notation available in the Z syntax. Utilising Alloy in this way is necessary as there is little tool support for Z over and above tools to type-check Z specifications.

Alloy is able to process a finite state space, meaning that the model is validated only within the scope of the finite parameter space that is considered. Z in contrast can describe the existence of infinitely large sets. This can be a problem when asserting the existence of particular instances of a signature, and occasionally requires the use of generator axioms in an Alloy specification. This issue is covered in [69], which summarises the problem as:

> " ... we choose to represent a Z schema (the set of all bindings that satisfy the constrain of the schema) as an Alloy signature, which denotes just some set of atoms that is not constrained to represent all possible values of the corresponding schema"

The two predicates below are the solution presented in [69]. The first adds a constraint to the Alloy model that no two distinct elements for *BBook* represent the same book, and the second is the generator axiom ensures all possible *BBook* are represented.

```
pred Canonicalisation {
    all disj b1, b2 : BBook | b1.birthday != b2.birthday
}
```

```
pred GeneratorAxiom {
    some b : BBook | no b.birthday
    all b : BBook, n : NAME - b.known, d : DATE |
        some b' : BBook |
            b'.birthday = b.birthday + n -> d
```

Generator axioms are covered more fully in [55], where the author argues that the use of generator axioms causes a state explosion to occur which can make analysis impractical. In addition it is argued that generator axioms are rarely needed as alternative approaches to modelling the problem can frequently be found.

A generator axiom was necessary in the early Alloy model used to validate the translation process presented in [98], but the enhanced Alloy model presented in Chapter 6 no longer requires the use of generator axioms.

Additionally it is worth noting that Alloy does not permit recursive definitions in predicates and functions, and alternative approaches need to be found if the structures being represented are recursive by nature. A potential solution to overcoming this restriction, in the context of the approaches to access control modelled in this thesis, is also given in Chapter 5.

2.7 SOA and web services

2.7.1 Motivation and background

In recent years significant research has involved the use of large, distributed systems. SOA has become a methodology of system design seen in several of these systems, such as [11]. The primary idea behind the SOA paradigm is captured in the OASIS Reference Model for Service Oriented Architecture 1.0 [4], which defines SOA as the following:

> "A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."

SOA is a software system architecture methodology where the goal is to achieve a loose coupling between the interacting services. In this paradigm, a service has a clearly defined interface described in a platform-independent way using XML documents. All communications with or between services are performed using XML documents that conform to a pre-defined schema. Each service performs a well-defined unit of work that, ideally, causes no side-effects and maintains no state.

A web service has the following definition in the W3C Web Services Architecture [3].

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

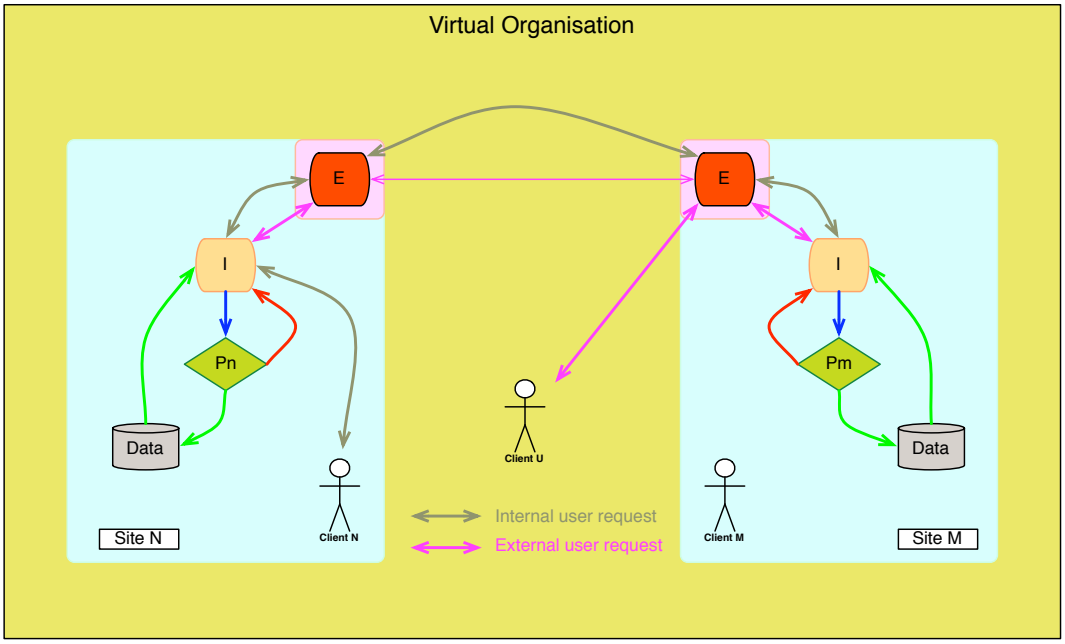

Figure 2.6: A generic virtual organisation

Web services [3] are typically used to implement a system utilising an SOA design, although other methods of implementation are possible. A web service is an Open Standards based web application that can exchange information and interact with other applications. There are a core set of standards that describe the syntax and semantics of software communication: XML provides the common syntax for representing data; the Simple Object Access Protocol (SOAP) [46] provides the semantics for data exchange; and the Web Services Description Language (WSDL) [32] provides a mechanism to describe the capabilities of a web service. Additionally other specifications define additional functionality for web services including security in WS-Security [77] and reliable message passing in WS-ReliableMessaging [78].

2.7.2 Virtual organisations

The above discussion concentrates on the underlying techniques and standards that can be used to develop a data grid. Once data grids have been formed it may be beneficial to share data between them. To facilitate this data sharing it is normal to instigate the creation of VOs.

A example of a large VO is the United Kingdom's National Health Service (NHS). The NHS is made up of a number of independent trusts, who on occasion need to share data. A possible way of achieving the data sharing is to instigate temporary VOs which could then enable distributed healthcare delivery and research to take place. A possible generic term for this type of large scale distributed system is a *healthgrid*.

The diagram in Figure 2.6 is a representation of a generic virtual health organisation. Each hospital has some data, with the data being accessed via an internal interface, I. The permitted access to the data is regulated by the policies (P). The external interface, E, to the organisation

accesses data via the internal interface. The access given to the request for data originating via E is determined by the local policies. The key element of this representation is that each organisation has ultimate control over the access to the data they hold. When sharing data within the context of a VO, this allows a hospital to share only the data they wish to. This leaves the responsibility for defining the policies associated with the access to data to the hospitals that holds the data. It is worth noting that the enforcement of the policies is outside any database management system that an organisation uses. This has the benefit that the same policies can be applied across a number of disparate data sources from different vendors.

The approach outlined above allows an organisation, such as a hospital, to facilitate the sharing of their data within a virtual organisation, while retaining control over who can gain access to the data. Therefore, leaving the responsibility for defining the policies associated with the access to data to the hospitals that holds the data. Some early work towards the realisation of these goals is described in [94].

An example of a concrete implementation of the above architecture was developed in the GIMI project [91, 93]. The main aim of GIMI was to develop a generic, dependable middleware layer capable of supporting secure and ethical data sharing across disparate sources to facilitate healthcare research, delivery, and training. The key to ensuring legal and ethical access to such data is that the mechanism should offer fine-grained and dynamic access control to resources: this was the key driver behind GIMI. The infrastructure was developed so that it was sympathetic to the needs of clinical researchers, commercial organisations involved in the medical domain, healthcare providers, and those concerned with providing training facilities — all of which were concerned with sharing confidential data. The evolution of the middleware which underpinned the GIMI project is described next.

2.7.3 sif

Experience of facilitating access to, and aggregation of, distributed data has been gained while working on and observing a number of projects. The eDiaMoND project [24] was concerned with sharing mammographic images for diagnosis, training and validation of imaging techniques designed to assist diagnosis. The experience gained in this project informed the writing of [84], which outlines the security goals of a secure grid based medical information system. These ideas were subsequently developed in NeuroGrid [44], which was concerned with the sharing of neurological images and data for research. The GIMI project further developed these ideas by providing a generic mechanism for securely accessing data and files. These mechanisms were utilised by the exemplars associated with GIMI to develop applications that demonstrated the benefits of the secure sharing of information. An early version of the middleware developed in GIMI was used in a pilot project for the National Cancer Research Institute Informatics Initiative [99, 83], which was concerned with linking data from different parts of the care pathway for patients suffering from colorectal cancer.

The sif middleware and underlying principles are described in [84, 93, 94, 97, 96, 92]. Fundamentally sif is a mechanism for facilitating the sharing of data in a secure way, a key tenet of which is that the data owner maintains control over who has access to their data. This is achieved by the access control policies relating to each data set being defined by the owner of

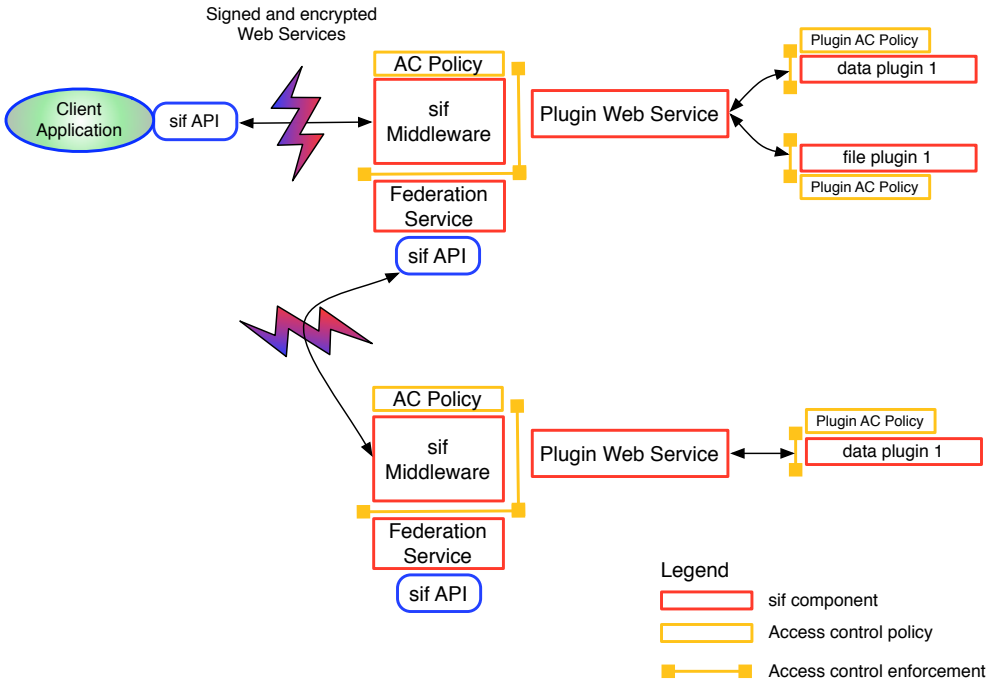

Figure 2.7: Access control architecture in sif

that data set.

The sif middleware also provides a way to add additional functionality via a 'plugin' mechanism, which presents a standard interface to resources accessed by one of the three supported plugin types: data plugins, file plugins and algorithm plugins. Data plugins provide access to data resources via a subset of SQL, with the plugin writer being responsible for any necessary translation between the data sources native interaction and SQL. The file plugin presents a resource to be exposed as if it were a file system. Finally, the algorithm plugin provides a facility for encapsulating and remotely invoking an algorithm. This is achieved by providing a standard way of defining the inputs required and outputs generated by the algorithm. The ability for any user to perform an action requires them to have been permitted by the sif middleware access control mechanism, which is described in Section 2.7.4.

The applications underpinned by the sif middleware that were developed by the exemplar teams of GIMI were used to validate the contribution of this thesis. These applications are covered in more detail in Chapter 3, which provides details about the use cases and requirements that were supplied by the exemplar teams that informed this work.

2.7.4 sif and access control

Figure 2.7 provides a simplified architectural view of the sif middleware, concentrating on where access is controlled within sif. A client application communicates with a sif server by utilising the sif application programming interface (API). All communications with the sif server are via signed and encrypted messages. The web services that provide the externally facing services of sif

are web services that will only respond to messages that are encrypted for themselves and signed by a certificate issued by an approved certificate authority. The sif middleware then examines the request to determine which of the other web services it should be passed to. However, before the request is passed on, an access control evaluation is performed to ensure the user is permitted access to the service requested. This evaluation is performed using the access control policy associated with the sif middleware — represented by the *AC Policy* box adjacent to the *sif middleware* box in the diagram.

If access to a plugin is requested, additional access control evaluations are performed. Each plugin can have an individual access control policy associated with it. This is illustrated in Figure 2.7 by the *Plugin AC Policy* adjacent to the individual plugings, e.g., *data plugin 1*. The ability to provide access control at the individual plugin level allows for simplification of access control policies. Instead of an overarching policy having to be defined and kept current with access control relating to potentially transient plugins, the central access control policy can concentrate on the general access control and those responsible for the individual plugins can define their own policies.

If the request is to perform some sort of federation function using the *Federation Service* then a new request is constructed and passed to the remote server by using the sif API. This request not only is signed and encrypted but also contains information about the route it has taken and the original requester. The sif middleware at the subsequent node will perfom an access control evaluation against the locally defined access control policies before any actions are permitted.

However, if the request is for a simple web service function — not illustrated — once the initial access control evaluation is performed and the user is granted permission, then the function is performed and results returned to the user via a message signed by the server and encrypted for the user.

2.7.5 Service-Oriented Federated Authorization

In the initial version of sif, all the access control policies had to be defined using XACML. The Service-Oriented Federated Authorization (SOFA) project[1] expanded the work of GIMI to add additional functionality to the sif middleware to facilitate the defining of access control policies using access control mechanisms other than XACML. SOFA has added simple RBAC and ACL access control mechanisms to the core functionality. Although this permits the simple implementation of current policies defined in one of those paradigms, there is still a need for translation between access control representations. This need comes from three distinct areas: firstly when the access control being used is not one of the three mechanisms directly supported; secondly from the desire to translate into the more expressive XACML representation to facilitate the capturing of more complex access control policies; and thirdly from the wish to be able to perform some analysis on the combination of policies for different locations when considering distributed access.

[1] See `http://web.comlab.ox.ac.uk/projects/SOFA/index.html`.

2.8 Summary

This chapter has presented a brief overview of access control in general and a detailed account of RBAC and XACML, which are used as example access control mechanisms to drive and validate the contribution of this thesis. Additionally, the use of formal methods in the context of access control is considered before giving a short introduction to Alloy. The Alloy section provides both an introduction to Alloy and some comments on the use of Alloy to model-check transformed Z models.

The final sections concentrate on an overview of the SOA technique used to construct distributed systems, together with a brief description of the sif middleware, which has been used as a test-bed for the research described in this thesis.

Chapter 3

Use cases and requirements

3.1 Introduction

In this chapter the relevant use cases and requirements — which have been elicited from various partners across a number of projects — that inform our work are considered. These use cases and requirements have confirmed the need for, and helped inform the approach taken to develop, the research contribution described in this thesis. Additional to the requirements that have been gathered, experience gained from association with a number of projects have highlighted the need to support heterogeneous access control in a distributed environment as well as providing facilities to data owners to maintain full control over the access control policies associated with their data sources.

The GIMI project has most recently provided the input for this work, with a number of developers working on exemplar applications providing valuable information and allowing a consolidated view of the key requirements to be generated. The rest of this chapter will provide some background to the applications that have informed the use cases and requirements. Following this overview, the key motivating use cases will be considered. The primary requirements that have emerged are then detailed. Finally some of the underlying techniques for providing an arbitrary translation mechanism are examined.

3.2 Context and middleware applications

The use cases and requirements have been derived from work undertaken on numerous projects, including eDiaMoND [24], NeuroGrid [44] and GIMI [91, 93]. The discussion presented here primarily pertains to the exemplar applications associated with the GIMI project; however, it is also informed by work with collaborators associated with the earlier projects. The developers associated with the GIMI project were responsible for creating applications pertaining to healthcare research, healthcare training, and healthcare delivery. A brief description of each of these application areas is provided below.

The overall aim of the first group of application developers was to develop medical image algorithms for application to breast and colorectal cancer (with examples including the work described in [104]). Via the sif middleware, algorithm developers were developing, training and

validating their algorithms via the utilisation of data stored in multiple locations in the UK and Italy. Here, sif was used to provide secure, audited access to legacy data and images from multiple different sources. Each of the local data owners had access control mechanisms in place, with the policies being defined in a way appropriate to that site. Typically the local access control mechanism was different to that used by the sif middleware.

The second application was developed by colleagues at Loughborough University and University College London, and was based on the work of the PERFORMS (Personal Performance in Mammographic Screening) project (the self-assessment programme used by the NHS's Breast Screening Programme) [43]. The application utilised data stored at Oxford, Loughborough, and Unversity College London, with the data comprising demographic information, digitised mammography x-rays, and annotations by experts. The prototype application permitted the end-user to select a particular condition they wished to review or receive additional training on. Once the condition of interest had been selected, the application performed a query via the sif middleware to locate appropriate cases that exhibit that condition. The user could then either review the cases or enter a training session based on the cases. If the user chose to review the cases then the user selected the case they wished to examine; this case was then downloaded and presented to the user as text and images with all the relevant expert comments and annotations. If the training mode was selected then a random case was chosen by the system and it was presented to the user as if they were performing an image reading, which required the user to identify abnormalities and record their observations. The cases and associated data could come from any of the sites and could be stored using a variety of methods including traditional databases and PACS (Picture Archiving and Communications System) — all of which had their own access control systems in place.

The third application area was concerned with the self-management of patients with long-term conditions. To support this, a system for disease management was required that integrated all the relevant information that required monitoring. The team which worked on this application area comprised researchers and developers from the Department of Engineering Science at Oxford University and t+ Medical, an Oxfordshire based SME. This group engaged in a telemedicine trial involving two GP (General Practitioner) surgeries from the local primary care trust—involving delivering blood-glucose summary data from a telemedicine handset to a GP's desktop. In addition to providing the data to the GP, researchers also evaluated predictive algorithms for identifying undesirable trends in the blood glucose readings to allow preventative steps to be taken before dramatic intervention was required. The same team were also responsible for developing a second application which supported a project relating to Asthma. The primary aim of the project was to ascertain if there is any correlation between exacerbation of asthma and the prevailing weather. The application permitted the subjects of the study to record their peak flow measurements at regular intervals. The researches were then able to test the hypothesis by joining the peak flow data with weather data, using the sif middleware.

The case study presented in Chapter 8, is derived from work with the team developing applications for long-term conditions, in particular, an application concerned with blood glucose levels associated with Diabetes.

In addition to the above development work carried out within the GIMI project, additional collaborative work has been undertaken with several other groups with the aim of developing

applications that utilise the sif middleware. A selection of these applications are described below.

In a demonstrator project for the National Cancer Research Institute (NCRI) Informatics Initiative [99], the sif middleware underpinned an application that was developed to facilitate the joining of data relating to different aspects of colorectal cancer. The application demonstrated the ability to join pre-operative MRI data with post-operative pathology data with the aim of demonstrating the utility of the multi-scale, multi-disciplinary approach to enhancing the information that can be derived from data collected within a clinical trial.

The Oxford Project to Investigate Memory and Ageing (OPTIMA),[1] which was established in 1988, aims to improve the understanding of the changes that occur as the brain ages via a longitudinal study. Although a large quantity of data has been collected — and continues to be collected — the data was only accessible to researchers via a data manager. An application was developed to permit researchers to formulate their own queries and gain direct access to a data resource without having to go through the data manager. In addition to regular test data relating to the studies subjects, post-mortem data including images, tissue sample and slides was also collected by a separate group. The second function of the application was to facilitate the federation of the study data and the post-mortem data, particularly any image data relating to changes in the brain.

Two further collaborations were concerned with the joining of local data with centrally held data. In the first, an application was developed to join data held at the Health Information Research Unit (HIRU)[2] with research data pertaining to cancer care and treatment held in a researcher's stand-alone database. The second application was developed to permit the aggregation of centrally held data with outlying data to allow the creation of a unified view of the student life cycle. This application facilitated the answering of questions that were historically difficult to answer because of the distribution of the data and the manual method previously employed to allow joining.

Finally, work with the Oxford Centre for Integrative Systems Biology (OCISB)[3] on data management has led to the development of an applications to support researchers using extensions to the Chaste software [82] to run cell-based simulations. Each group using the cell-based Chaste to run simulations stores the initial parameters used for each simulation they ran along with version information and a portion of the animation data generated. A detailed description of how the sif middleware was used to underpin an application that facilitated the sharing of simulation data between different groups is provided in [95]. The overall goal of the collaboration was to achieve a wider coverage of the possible starting parameter than would have been possible by any individual group.

3.3 Use cases

Based on discussions with the developers of the above applications and work on previous projects (see Appendix B), the primary drivers of the work of this thesis can be characterised in terms of the following general use cases.

[1] See `http://www.medsci.ox.ac.uk/optima`
[2] See `http://www.swan.ac.uk/ils/Research/CHIRAL/Methodologies/HealthInformatics/HealthInformationResearchUnit/`
[3] See `http://www.sysbio.ox.ac.uk`

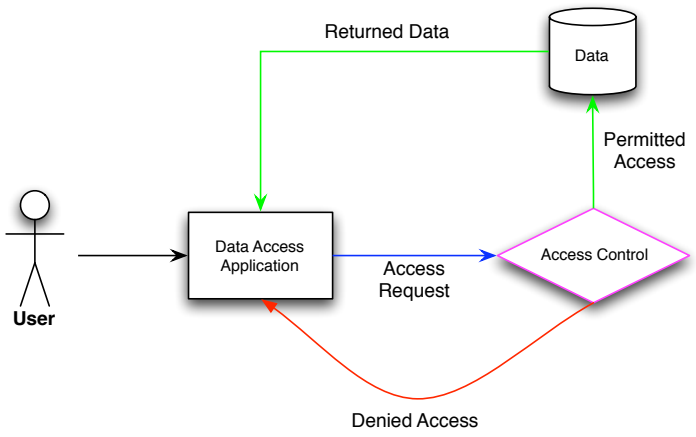

Figure 3.1: Access control with local data access

1. Use existing access control application and leverage new technologies that use a different access control paradigm.
2. In a heterogeneous environment allow different people to use different applications.
3. Facilitate the creation of an overall view of who can do what in a distributed, heterogeneous environment.
4. Allow the validation of policy combinations even when the policies are captured in different access control paradigms.

We might consider all of these in isolation or combination.

The first of these use cases is best illustrated by the move from using one access control paradigm to another for protecting local data. The general preference expressed by the collaborators associated with the GIMI project was for them to continue to use their preferred method of describing access control policies and not having to learn a new approach. Figure 3.1 represents the case where collaborators currently use local applications to access their data with access control policies defined in a representation which they are confident in using. If a new access control paradigm were to be introduced the preference expressed was to have a mechanism by which they could continue to use the original policies. Figure 3.2 illustrates a possible method of achieving this use case by translating from the old access control representation into the new access control representation.

This is equally applicable when considering allowing remote access to be granted to local data via middleware that utilises a different access control paradigm to the local access control. Figure 3.3 illustrates this scenario where the data is shared by middleware which implements local access control rather than central access control.

For the second use case it is important that consistent access control be provided in a heterogeneous environment where multiple applications are utilised to access data. The original application may define policies using a representation that is different to the subsequent applications. In that case to provide a consistent view of the data to individuals utilising different

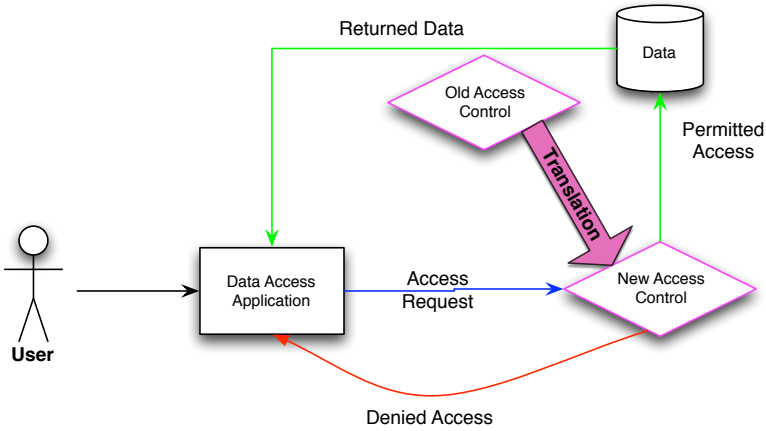

Figure 3.2: Local data access with new access control paradigm

applications the access control policy needs translating into each of the representations used by the different applications.

Considering the third use case, in a heterogeneous environment it could be advantageous to have an overall view of who can do what. The first stage of being able to generate this type of view would be to have formal definitions of the access control systems of interest along with formal definitions of translations between them. In addition a formally defined method of combining the policies would be needed to produce an overall policy that could be evaluated to produce an overall view as to who can perform which actions.

For the fourth use case, when we consider combining policies, especially those originally captured using different access control representations, it is desirable to have some assurance that the overall policy is valid and consistent. Formal descriptions of the translations and combinations of policies would provide a mechanism for validating that the overall policy conforms to the representation in which it is expressed. The formal description could also be utilised for checking the consistency of the overall policy, for example by ensuring that individuals are not permitted and denied the same permission within the policy.

3.4 Requirements

The main requirement that is of interest is the need to translate from one access control representation into another. This is a requirement that arises directly from the first use case. A typical example of where this may be necessary is when data is being exposed by using middleware which uses a different access control paradigm to that of the data source. When considering accessing data from a number of legacy systems via such means, it is essential that the individual access control policies for each of the legacy systems be incorporated into the distributed environment. When considering such cases it is equally important to ensure that the meaning of the access control policies be maintained to provide appropriate protection of the data. That is, the access control only permits those with appropriate authorisation to access the data.

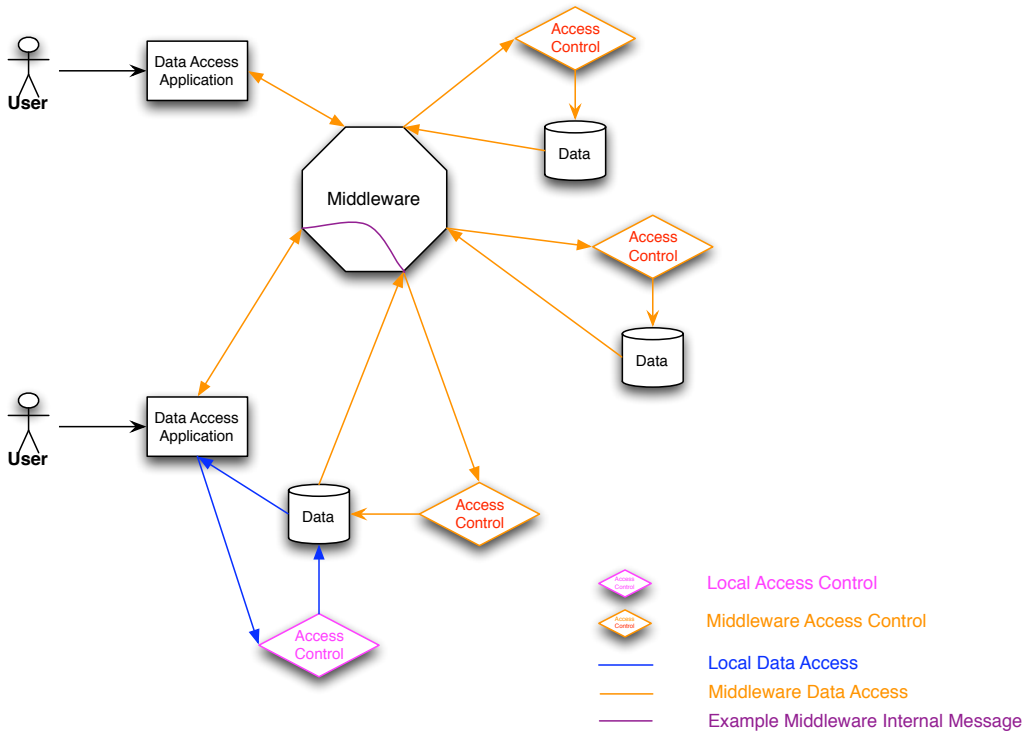

Figure 3.3: Access control with distributed and local data access

Another instance where the translation of access control policies from one representation into another was necessary was during the initial prototyping stages of the Oncology Information Exchange (ONIX) project.[4] Part of the initial feasibility work was to create a bridge between data sources connected to the Cancer Biomedical Informatics Grid (CaBig) initiative[5] and the initial data exposed by the NCRI demonstrator [99]. This work required the manual translation of access control policies, which was time-consuming and potentially error-prone. This experience re-enforced the need for an automated translation mechanism.

A prime driver for a translation mechanism is the fact that the developers and users associated with GIMI and other connected projects all expressed the view that they did not wish to learn another access control paradigm. They stated that they understood their current access control representation and were happy using it. In addition, they felt uncomfortable with the prospect of creating ad-hoc translations of access control policies into a representation they did not fully understand.

When considering the requirement to perform translations between representations, a number of supporting requirements were expressed: the translation has to be accurate, faithfully capturing the meaning of the original policy, and the translation has to be simple to perform.

When accessing data via the middleware as illustrated in Figure 3.3, it is imperative that the local users get the same access to data whether they access it via a local application or the

[4]See `http://www.ncri-onix.org.uk`
[5]See `http://cabig.cancer.gov/`

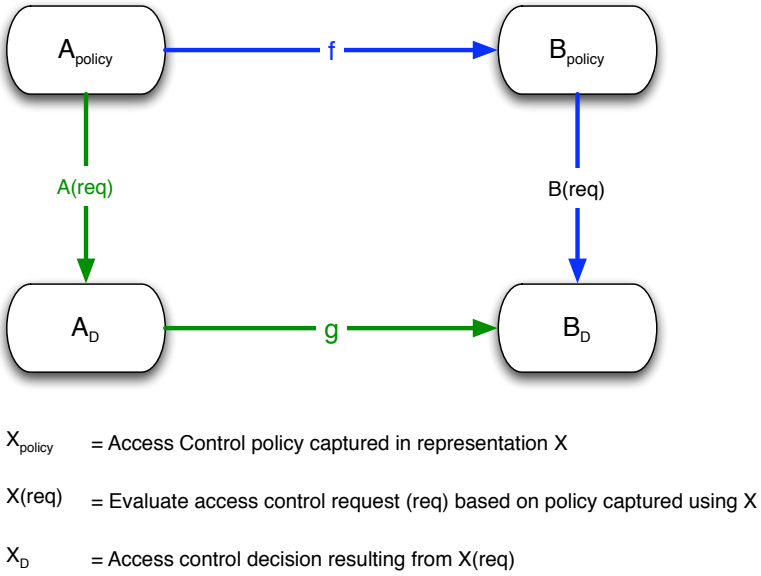

Figure 3.4: Translation properties

middleware. In addition, the access afforded to remote users by a data owner should initially be defined in the access control representation that the data owner is comfortable with using. The access control policies would then require translating into the access control representation utilised by the middleware providing the remote access.

The most important functional requirement is that the policy resulting from the translation has to be functionally indistinguishable from the original policy under the application of an access control request. That is, the access control decision provided by the original and resulting policies have to result in the same access decision for all access control requests.

The translations that are of interest to this work are point in time translations of a static access control policy between representations. The initial work does not intend to deal with dynamic policies or the translation of additional properties between representations. For instance, in general it would not be possible to translate the static separation of duty constraints of an RBAC policy into a policy expressed in an arbitrary access control paradigm. This is primarily because not all access control paradigms consider separation of duty to be of interest and therefore do not provide a facility to capture it.

Figure 3.4 illustrates the properties that need to hold for a translation between two access control paradigms. The two access control paradigms under consideration in Figure 3.4 are A and B. A policy defined using representation A, A_{policy}, is translated into a policy in B, B_{policy}, using the function f — illustrated by the upper horizontal line. The result A_D, is obtained for a request, req, by evaluating $A(req)(A_{policy})$, which is illustrated by the left-hand vertical line. In a similar way the evaluation of $B(req)(B_{policy})$ results in B_D. For the translation to be considered correct, the results given by the different access control paradigms for req must be equivalent, that is, A_D, must map to B_D, under the translation performed by function g. This means that the predicate $\forall\, a : A_{policy};\ req : Request \bullet B(req)(f(a)) = g(A(req)(a))$ must hold for the translation

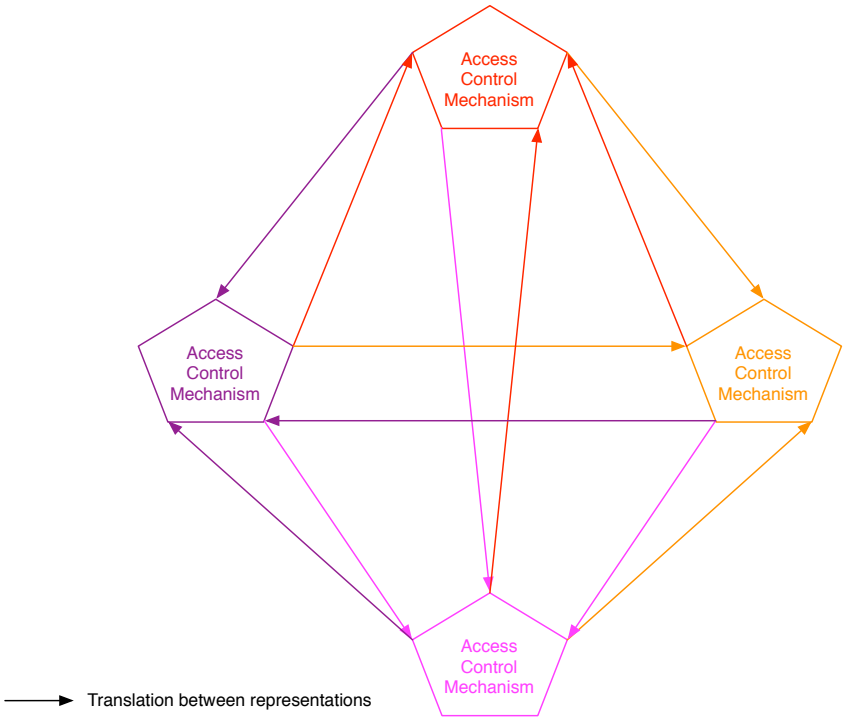

Figure 3.5: Translating directly between different access control representations

to be valid.

3.5 Translation methods

When considering the translation between different representations, there are two distinct ways of performing a translation between any arbitrary pair of representations. One method is direct translation between each pair of representations; the alternative is to translate via an intermediate representation.

Figure 3.5 illustrates the case where a separate translation is defined between every pair of representations. This would require the writing of $n(n-1)$ translators to cover all possible translations, where n is the number of different representations. When a new representation is added to the currently supported s representations it would require $2s$ translators to be written to provide the ability to perform arbitrary translations.

Figure 3.6 illustrates the case where the overall translation is achieved by using a generic representation as an intermediate step. In this case each source/target representation only needs to define a translation to and from the generic representation. Thus, when using the generic representation, we only need to write $n * 2$ translators. In a similar fashion, when support for a new representation is needed only two translators need to be written: translation from the representation to the generic representation and back.

When comparing the two methods, it is clear that the number of translations that need

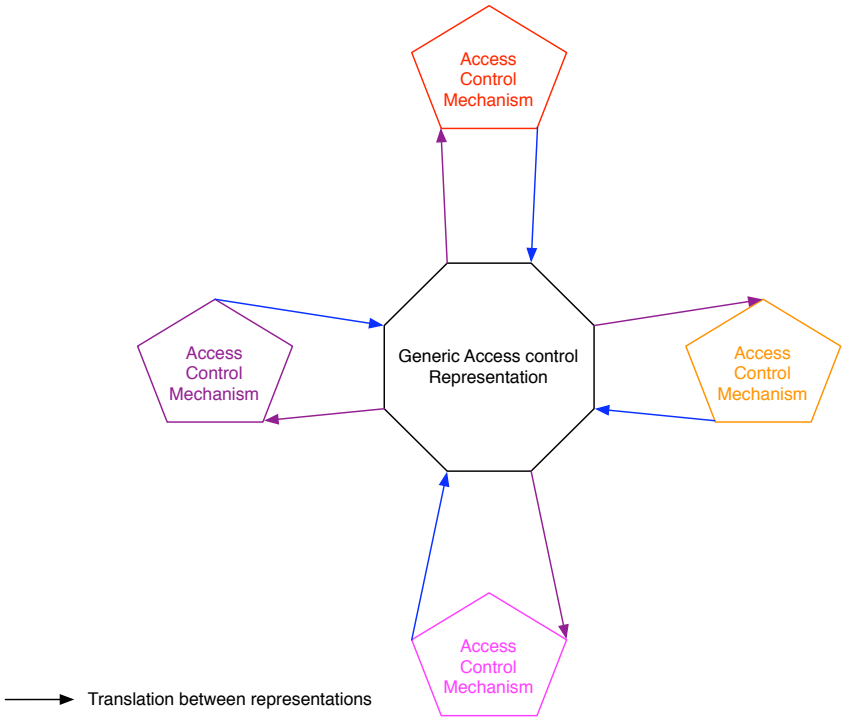

Figure 3.6: Translation via a generic access control representation

defining grows quadratically when a translation between every representation is needed, but the growth is linear when the translation utilises an intermediate generic representation. In the illustrations of Figures 3.5 and 3.6 we can see that when considering four access control representations the number of translations required are 12 and 8 respectively. This illustrates the ideal case where there is only one generic representation; in practice, though, there may be the need for several generic formats to facilitate the translation process.

If this method is used, a generic representation needs to be capable of capturing the key elements of access control policies regardless of their origin. The most important aspects that need to be captured are the relationships between the subjects, actions, resources and permission. This relation is essentially the access control policy. In addition to the relation it is important to be able to indicate if the policy is a default *Permit* or default *Deny*. That is, it is necessary to know what the default action, if any, should be if the request does not generate an explicit result when evaluated in terms of the policy.

3.6 Adopted approach

It would be impractical within the scope of this thesis to develop a generic solution that meets the requirements described in this chapter. This thesis instead will describe an approach which demonstrates how the requirements could be met by using RBAC and XACML as illustrative examples. The following chapters provide formal descriptions of: RBAC (Chapter 4), XACML

(Chapter 5), and the translation between RBAC and XACML (Chapter 6).

3.7 Summary

This chapter has described the use cases and requirements that have driven our research. The collaborators from the GIMI, eDiaMoND and NeuroGrid projects have contributed to the general consensus of requirements described. The key requirement articulated was the need to be able to translate from one access control format to another. Typical cases where translation are necessary include the sharing of data via middleware when the middleware utilises a different access control paradigm to the data source and when there is a need to bridge different systems together to facilitate the sharing of data sources between different frameworks. Additionally, the reluctance of collaborators to learn a new access control paradigm reinforced the need for an automated translation mechanism. However, if such a translation mechanism is to be useful it has to be accurate; that is, the result of the translation has to reflect the intent of the original access control policy. To work towards this goal, a formal approach has been adopted. The techniques described in the rest of this thesis will be mindful of the use cases presented in Section 3.3, which will be utilised as success criteria for the work described in the following chapters.

Chapter 4

Formal representations of role-based access control

To facilitate the creation of a formally defined translation process, in accordance with the goals of Chapter 3, it is first necessary to produce formal descriptions of the access control models that we use as exemplars.

To this end, this chapter presents two formal models of RBAC. The first model has been developed using the Z schema language. This model is derived from the one published in [85], which was developed in collaboration with Simpson and Power. A second model is presented in terms of Alloy and captures core RBAC and hierarchies. This second model is an enhancement of the one described in [98].

4.1 A formal model of RBAC in Z

This section presents a formal description of RBAC in terms of Z. It utilises the schema language of Z; this is a distinction worth making as other approaches (see for example, [14] and [66]) utilise only the mathematical language of Z. The use of schemas allows the model to be defined in a modular fashion, permitting a compositional approach to model development. In the case of RBAC it becomes possible for optional components such as role hierarchies and sessions to be added (or removed) as required without changing the underlying core model. It also becomes possible to reason about multiple RBAC systems without the need for renaming. The following subsection describes a simple running example, which will help to illustrate the formal models defined in this and subsequent chapters.

4.1.1 A running example

The running example is based on a very simple scenario that is motivated and informed by previous work with projects in the healthcare domain.

The scenario concerns the reading and writing of prescription information. The prescription information is held in a database called *PrescribeDB*. For the purpose of this example, all users are either a *doctor* or a *nurse*. A doctor can *read* a prescription from the database, *write*

new prescriptions or modify existing ones (for the sake of simplicity both the write and modify actions are modelled as a write to the database). A nurse, on the other hand, can only read a prescription from the database. The users for this scenario are *Austin*, *Morris*, *Rover*, and *Triumph*. We assume that the users Morris and Rover are doctors and the users Austin and Triumph are nurses.

4.1.2 Types

It is first necessary to introduce the basic types *User*, *Role*, *Action* and *Resource*, and characterise a permission as an action-resource pair identified by *PRMSBase*:

$$[User]$$
$$[Role]$$
$$[Action]$$
$$[Resource]$$
$$PRMSBase == Action \times Resource$$

To illustrate the basic types, it is possible to create definitions which are consistent with our example. Values are defined as being members of the basic types. As such, to remain consistent with the example it is necessary to define: *Austin*, *Morris*, *Rover* and *Triumph* to be members of *User*; *read* and *write* to be elements of *Action*; *Doctor* and *Nurse* to be in the set *Role*; and finally *prescribeDB* to be an element of *Resource*.

$$Austin : User$$
$$Morris : User$$
$$Rover : User$$
$$Triumph : User$$
$$read : Action$$
$$write : Action$$
$$Doctor : Role$$
$$Nurse : Role$$
$$prescribeDB : Resource$$

4.1.3 Core RBAC

The core RBAC system, captured by the schema *Core*, consists of the relations *UA* and *PA*, which represent the relationships that hold between users and roles, and roles and permissions respectively. These relations are restricted to the members of *USERS*, *ROLES* and *PRMS*, which represent the sets of current users, roles and permissions, and form part of the *Core* schema.

Core

$$UA : User \leftrightarrow Role$$
$$PA : Role \leftrightarrow PRMSBase$$
$$USERS : \mathbb{P}\, User$$
$$ROLES : \mathbb{P}\, Role$$
$$PRMS : \mathbb{P}\, PRMSBase$$

$$UA \in USERS \leftrightarrow ROLES$$
$$PA \in ROLES \leftrightarrow PRMS$$

It is now possible to capture our running example in terms of a core RBAC policy, *core*, which is an instance of *Core*. In the instance *core*, the relation *UA* relates each of the defined users with their assigned roles, and the relation *PA* associates each role with its permissions. Finally, the sets *USERS*, *ROLES* and *PRMS* capture the users, roles and permissions associated with *core*.

$$core : Core$$

$$core.UA = \{(Austin, Nurse), (Morris, Doctor), (Rover, Doctor), (Triumph, Nurse)\}$$
$$core.PA = \{Nurse \mapsto (read, prescribeDB), Doctor \mapsto (read, prescribeDB),$$
$$Doctor \mapsto (write, prescribeDB)\}$$
$$core.USERS = \{Austin, Morris, Rover, Triumph\}$$
$$core.ROLES = \{Doctor, Nurse\}$$
$$core.PRMS = \{(read, prescribeDB), (write, prescribeDB)\}$$

4.1.4 General role hierarchies

When hierarchies are to be considered, then the additional relations RH, $\succeq$ and $\succ\!\!\succ$ need to be added to those already present in the core schema, where the relation RH represents the explicitly defined *role dominance* relationship which describes which roles are senior to others. A constraint is placed on the RH relation that it must be irreflexive and acyclic; this is necessary to prevent a role being senior to itself either directly or via transitivity. The next relation, $\succeq$, represents the *derived relationship*, which is the reflexive transitive closure of RH (written RH^*).

It is sometimes useful to know if one role is the immediate predecessor of another: the *immediate successor* relation $\succ\!\!\succ$ captures this relationship. While the relation RH is sufficient to define hierarchies, the relations $\succeq$ and $\succ\!\!\succ$ are useful if additional constraints need to be places on the hierarchies. Such additional constraints are touched upon in Section 4.1.5 when considering limited role hierarchies. The *Hierarchy* schema below captures these relations and constraints.

Figure 4.1 illustrates the relations, RH, $\succeq$ (RH^*) and $\succ\!\!\succ$. In this case, the set *Role* is $\{a...h\}$ and RH is the relation $\{b \mapsto c, b \mapsto d, c \mapsto e, e \mapsto f, e \mapsto h, f \mapsto h\}$, shown on the left-hand side. The central portion illustrates $\succeq$, (RH^*), which adds the mappings indicated by the broken lines. Finally, the right-hand side illustrates the immediate successor relation, $\succ\!\!\succ$, which in this case involves the removal of the mapping $e \mapsto h$ — indicated by the dotted line.

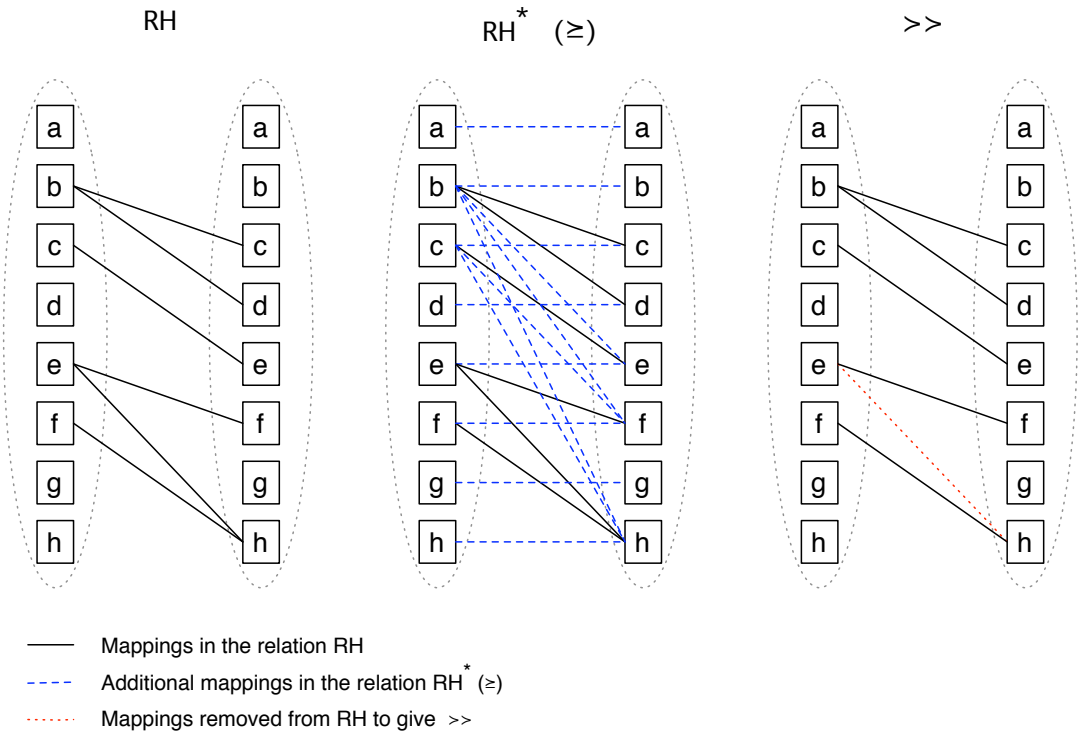

Figure 4.1: RBAC Hierarchy relations

Hierarchy

$$
\begin{array}{l}
Core \\
RH : Role \leftrightarrow Role \\
\_ \succeq \_ : Role \leftrightarrow Role \\
\_ \succ\!\!\succ \_ : Role \leftrightarrow Role \\
\hline
RH \in ROLES \leftrightarrow ROLES \\
\forall\, r : Role \bullet (r, r) \notin RH^{+} \\
\forall\, r_1, r_2 : Role \bullet r_1 \succeq r_2 \Leftrightarrow (r_1, r_2) \in RH^{*} \\
\forall\, r_1, r_2 : Role \bullet r_1 \succ\!\!\succ r_2 \Leftrightarrow r_1 \succeq r_2 \land r_1 \neq r_2 \land \neg\, (\exists\, r_3 : Role \setminus \{r_1, r_2\} \bullet r_1 \succeq r_3 \land r_3 \succeq r_2)
\end{array}
$$

It is possible to express our scenario as an element of *Hierarchy*, *rbach*. In *rbach*, the *RH* relation includes $(Doctor, Nurse)$, which represents the fact that the role *Doctor* inherits the permissions of the role *Nurse*. This inheritance means that, unlike *core* defined above, the role *Doctor* does not need to be directly associated with the permission $(read, prescribeDB)$ in the *PA* relation.

$$
\begin{array}{|l}
rbach : Hierarchy \\
\hline
rbach.UA = \{(Austin, Nurse), (Morris, Doctor), (Rover, Doctor), (Triumph, Nurse)\} \\
rbach.PA = \{Nurse \mapsto (read, prescribeDB), Doctor \mapsto (write, prescribeDB)\} \\
rbach.RH = \{(Doctor, Nurse)\} \\
rbach.USERS = \{Austin, Morris, Rover, Triumph\} \\
rbach.ROLES = \{Doctor, Nurse\} \\
rbach.PRMS = \{(read, prescribeDB), (write, prescribeDB)\}
\end{array}
$$

4.1.5 Limited role hierarchies

It is then possible to model limited role hierarchies via schema inclusion: all of the declarations and constraints of the *Hierarchy* schema are inherited by the *Limited_hierarchy* schema. A limited hierarchy is defined as a hierarchy with an additional constraint that a role can be senior to at most one other role; the *immediate successor* relation, $\succ\!\!\succ$, from the *Hierarchy* schema is used in defining the necessary constraint.

Limited_hierarchy

$$
\begin{array}{|l}
Hierarchy \\
\hline
\forall r_1, r_2, r_3 : Role \bullet r_1 \succ\!\!\succ r_2 \land r_1 \succ\!\!\succ r_3 \Rightarrow r_2 = r_3
\end{array}
$$

4.1.6 Static mutually exclusive roles

The notion of static mutually exclusive roles (SMERs) was first introduced in [65], with the concepts being essentially the same as that of SSDs, described in Section 2.3.3.

First, it is necessary to introduce the basic type *SMER_ID*.

$$[SMER\_ID]$$

The schema *SMER* introduces the relation *SC*, which maps a *SMER_ID* to pairs, with, in each case, the first element of the pair being a set of roles and the second element of the pair being a natural number. A constraint is added to ensure that the natural number is at least 2 and at most the number of elements in the set. Each individual SMER constraint puts a limit on the number of roles any single user may have in a given set, which is captured by the final predicate in the constraints of *SMER*.

SMER

$$
\begin{array}{|l}
Core \\
SC : SMER\_ID \pfun (\mathbb{P}\ Role \times \mathbb{N}) \\
\hline
SC \in SMER\_ID \pfun (\mathbb{P}\ ROLES \times \mathbb{N}) \\
\forall rs : \mathbb{P}\ Role;\ n : \mathbb{N} \mid (rs, n) \in \operatorname{ran} SC \bullet 2 \leq n \land n \leq \#rs \\
\forall rs : \mathbb{P}\ Role;\ n : \mathbb{N} \mid (rs, n) \in \operatorname{ran} SC \bullet (\forall u : User \bullet \#\{r : rs \mid (u, r) \in UA\} < n)
\end{array}
$$

It is possible to create a *SMER*, *smer*, by adding a *SC* constraint to the above defined *core* which prevents any user from holding both the roles *Doctor* and *Nurse*. The *SC* constraint has an

identifier, $smer1$, which is defined as an element of $SMER\_ID$. SC is then defined as associating $smer1$ with the constraint $(\{Doctor, Nurse\}, 2)$.

$$smer1 : SMER\_ID$$

$$\begin{array}{l} smer : SMER \\ \hline smer.UA = \{(Austin, Nurse), (Morris, Doctor), (Rover, Doctor), (Triumph, Nurse)\} \\ smer.PA = \{Nurse \mapsto (read, prescribeDB), Doctor \mapsto (read, prescribeDB), \\ \qquad\qquad Doctor \mapsto (write, prescribeDB)\} \\ smer.USERS = \{Austin, Morris, Rover, Triumph\} \\ smer.ROLES = \{Doctor, Nurse\} \\ smer.PRMS = \{(read, prescribeDB), (write, prescribeDB)\} \\ smer.SC = \{smer1 \mapsto (\{Doctor, Nurse\}, 2)\} \end{array}$$

The schema *HSMER* differs from *SMER* in that it considers the constraints associated with static mutually exclusive roles in an environment in which role hierarchies are present. This is achieved by adding the constraint that checks the roles either directly associated with a user or associated with a user through the role hierarchy in place, by using the $\succeq$ relation.

HSMER

$$\begin{array}{l} Hierarchy \\ SMER \\ \hline \forall\, rs : \mathbb{P}\, Role;\ n : \mathbb{N} \mid (rs, n) \in \operatorname{ran} SC \bullet \\ \qquad \forall\, u : User \bullet (\#\{r : rs \mid \exists\, r' : Role \bullet (u, r') \in UA \land r' \succeq r\} < n) \end{array}$$

Note that it is possible given the above to construct a model of RBAC in which role hierarchies exist *but which are not respected by SMER*. This would be the case if the relation $\succeq$ is not considered when calculating if any user is in breach of a SMER constraint. Such a model might be defined via schema conjunction between *Hierarchy* and *SMER*, shown below, which does not include the additional constraint of *HSMER* which considers inherited roles when assessing the SMER conditions:

$$SMERnotRespectingHierarchy \mathrel{\widehat{=}} Hierarchy \land SMER$$

4.1.7 Evaluation of access requests

To perform an evaluation of a request to act upon a resource, it is first necessary to model a request. In the following schema a request is modelled as consisting of a request identifier, along with sets made up of elements from the types *User*, *Action*, and *Resource*.

$$
\begin{array}{|l}
\textit{Request} \\
\hline
usr : User \\
act : Action \\
res : Resource \\
\hline
\end{array}
$$

This effectively models a user asking to perform an action on a resource.

It is now necessary to define a set of evaluation results and an access control evaluation function. The *Effect* of an evaluation of an RBAC policy with respect to a user making a request to perform an action can be to either *Permit* or *Deny* the request. The evaluation of the request is performed by a function which takes an instance of $RBAC$ and a request as input and returns a member of *Effect* as a result. Two functions to evaluate requests for perfoming actions on resources are defined: the first for requests against *Core* and the second for requests involving *Hierarchy*. The main part of each evaluation checks that the relation generated from a request, req, $req.usr \mapsto (req.act, req.res)$ exists within the set generated from the composition $core.UA \fatsemi core.PA$ when considering an element of *Core*, or $rbach.UA \fatsemi rbach.RH^* \fatsemi rbach.PA$ when considering an element of *Hierarchy*. The existence of the relation in the set reflects the fact that the user has been given permission to perform the requested action on the resource.

$$Effect ::= Permit \mid Deny$$

$$
\begin{array}{|l}
evalCore : Core \times Request \rightarrow Effect \\
\hline
\forall\, req : Request;\ core : Core \bullet \\
\quad evalCore(core, req) = Permit \Leftrightarrow \\
\quad\quad req.usr \mapsto (req.act, req.res) \in core.UA \fatsemi core.PA \\
\end{array}
$$

$$
\begin{array}{|l}
evalHierarchy : Hierarchy \times Request \rightarrow Effect \\
\hline
\forall\, req : Request;\ rbach : Hierarchy \bullet \\
\quad evalHierarchy(rbach, req) = Permit \Leftrightarrow \\
\quad\quad req.usr \mapsto (req.act, req.res) \in rbach.UA \fatsemi rbach.RH^* \fatsemi rbach.PA \\
\end{array}
$$

4.1.8 Sample requests

Before it is possible to perform any evaluations, it is first necessary to define requests, two of which are provided below. The first represents the doctor called Morris making a request to write to *prescribeDB*. The second represents the nurse called Austin making a request to write to *prescribeDB*.

$$
\begin{array}{|l}
reqMorrisWrite : Request \\
\hline
reqMorrisWrite.usr = Morris \\
reqMorrisWrite.act = write \\
reqMorrisWrite.res = prescribeDB \\
\end{array}
$$

$$
\begin{array}{|l}
reqAustinWrite : Request \\
\hline
reqAustinWrite.usr = Austin \\
reqAustinWrite.act = write \\
reqAustinWrite.res = prescribeDB
\end{array}
$$

4.1.9 Evaluations

It is now possible to evaluate the requests in relation to both *core* and *rbach*. Using the function *evalCore* with *core* and the request *reqMorrisWrite* results in the following.

$$evalCore(core, reqMorrisWrite) = Permit$$

To evaluate the application of the function, it is necessary to generate a mapping based on the request and a set of mappings based on the instance of the core. The mapping from the request *reqMorrisWrite* becomes

$$Morris \mapsto (write, prescribeDB)$$

The set of mappings from the composition of *UA* and *PA* from the instance of *Core* supplied is:

$$
\begin{aligned}
&core.UA \fatsemi core.PA = \\
&\quad \{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB), \\
&\quad\ \ Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB), \\
&\quad\ \ Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}
\end{aligned}
$$

For the function to return *Permit*, the mapping from the request must be in the set of mappings from the core, which is captured below.

$$
\begin{aligned}
&Morris \mapsto (write, prescribeDB) \in \\
&\quad \{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB), \\
&\quad\ \ Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB), \\
&\quad\ \ Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}
\end{aligned}
$$

Therefore, evaluating *evalCore* using *core* and *reqMorrisWrite* results in *Permit*.

Next, consider the evaluation of *evalCore* using *core* and *reqAustinWrite*:

$$evalCore(core, reqAustinWrite) = Deny$$

The mapping based on the request becomes:

$$Austin \mapsto (write, prescribeDB)$$

The expansion of $core.UA \fatsemi core.PA$ generates the same set of mappings as above.

This results in the mapping from the request not being a member of the set of mappings from the core, captured below, for which the result of the function application is *Deny*.

$$Austin \mapsto (write, prescribeDB) \notin$$
$$\{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB),$$
$$Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB),$$
$$Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}$$

The same two requests can be evaluated against *rbach* using the *evalHierarchy* function. The mappings for the two requests remain unchanged from the results presented above. The set of mappings generated from *rbach* by $core.UA \fatsemi rbach.RH^* \fatsemi core.PA$ is captured in the following fashion:

$$core.UA \fatsemi rbach.RH^* \fatsemi core.PA =$$
$$\{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB),$$
$$Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB),$$
$$Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}$$

The results of applying the function *evalHierarchy* to the two requests becomes:

$$evalHierarchy(rbach, reqMorrisWrite) = Permit$$

This is due to the fact that

$$Morris \mapsto (write, prescribeDB) \in$$
$$\{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB),$$
$$Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB),$$
$$Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}$$

Furthermore,

$$evalHierarchy(rbach, reqAustinWrite) = Deny$$

because

$$Austin \mapsto (write, prescribeDB) \notin$$
$$\{\ Austin \mapsto (read, prescribeDB), Morris \mapsto (read, prescribeDB),$$
$$Morris \mapsto (write, prescribeDB), Triumph \mapsto (read, prescribeDB),$$
$$Rover \mapsto (read, prescribeDB), Rover \mapsto (write, prescribeDB)\ \}$$

It is worth noting that the set of mappings generated from the instances of *core* and *rbach* are identical, which is as expected as both instances encode the same scenario.

4.2 A formal model of RBAC in Alloy

This section utilises Alloy [55] to formally define a model of core RBAC and hierarchal RBAC. The Alloy model corresponds to the Z model presented in Section 4.1. The SMER constraints of Section 4.1.6 are not included in the Alloy model, as they are not necessary for the translation process of Chapter 6. First, it is necessary to define the basic types that are necessary to create the core RBAC and hierarchal RBAC models.

4.2.1 Types

The `accesscontrol/types` module defines the signatures and facts that are the main primitives used for defining the core RBAC model, and later reused in the XACML model presented in Chapter 5. First, we define `EvalRes` and then define possible values by extension. `EvalRes` is the value returned as a result of evaluating a access request against an access control policy. The `one sig` (for 'signature') decoration on the definition of `Permit`, `Deny` and `NotApplicable` ensures that any reference to `Permit` will be the same instance, and similarly for `Deny` and `NotApplicable`.

```
module accesscontrol/types

abstract sig EvalRes {}
one sig Permit, Deny, NotApplicable extends EvalRes {}
```

Next, it is necessary to define `User`, `Role`, `Action` and `Resource`, with `Action` and `Resource` being used to define the contents of a `PRMSBase` which represents a permission within the RBAC model defined below. An access control request is then defined as a coupling of elements of `User` and `PRMSBase`, which effectively models a user requesting permission to perform an action on a resource.

```
sig User, Role, Action, Resource {}

sig PRMSBase {
    action : Action,
    resource : Resource
}

sig Request {
    u : User,
    p : PRMSBase
}
```

Finally, a fact is added that ensures each `PRMSBase` is unique and no two have the same action/resource pair. It is necessary to ensure unique instances exist to enable matching within Alloy: if this was omitted, Alloy would create unequal instances even if the internal values were identical. It would be possible to define matching functions but would increase the state space used further limiting the size of instance that can be evaluated.

```
fact uniquePRMSBase {
    all disj pb1, pb2 : PRMSBase |
        pb1.action != pb2.action || pb1.resource != pb2.resource
}
```

4.2.2 RBAC Core

The module `accesscontrol/rbac` is the model of RBAC Core in Alloy, utilising the definitions of `accesscontrol/types`. In the definition of `Core` — which is based on the Z model presented earlier in this chapter — `UA` is a mapping between `Users` and `Roles`, `PA` is a mapping between `Roles` and `PRMSBase`, and the sets `USERS`, `ROLES` and `PRMS` are subsets of `User`, `Role` and `PRMSBase` respectively. The relationships are constrained to ensure that the elements of `UA` and `PA` are within `USERS`, `ROLES` and `PRMS`, the set of roles in `UA` matches the set of roles in `PA` (`User.UA = PA.PRMSBase`), and `ROLES` is the set of `Roles` from `PA.PRMSBase`.

```
module accesscontrol/rbac
open accesscontrol/types

sig Core {
    UA : User -> Role,
    PA : Role -> PRMSBase,
    USERS : set User,
    ROLES : set Role,
    PRMS : set PRMSBase
} {
    UA in USERS -> ROLES
    PA in ROLES -> PRMS
    User.UA = PA.PRMSBase
    ROLES = PA.PRMSBase
}
```

To perform analysis involving the evaluations of requests against instances of the RBAC model, it is now necessary to consider an evaluation function. The `evalRBAC` function takes an element of `Core` and an element of `Request` as input, and returns `Permit` if the corresponding `User` and `PRMBase` in the supplied `Request` — `req` — exists in the product of `UA` and `PA` from the supplied instance of `Core` — `rbac` — being used; otherwise it returns `Deny`.

```
fun evalRBAC(rbac : Core, req : Request) : EvalRes {
    ((req.u -> req.p) in ((rbac.UA).(rbac.PA))) => Permit else Deny
}
```

4.2.3 RBAC Hierarchy

The module `accesscontrol/rbach` is the model of RBAC Hierarchy utilising the definitions found in `accesscontrol/types` and `accesscontrol/rbac`. The definition of `Hierarchy` extends `Core`, and is based on the Z model of Section 4.1. `Hierarchy` introduces `RH`, which is a mapping between `Roles` and is constrained to contain only roles found in `ROLES`.

The second constraint defines the elements that make up the set `ROLES`, which consists of the roles found in the relations `UA`, `RH` and `PA`. The final constraint ensures `RH` is irreflexive and acyclic. Here, `^RH` is the transitive closure of `RH`, `iden` is the identity function, and `&` indicates intersection

of sets. This results in `no ^RH & iden` being the constraint that there are no members of the identity function in the transitive closure of RH, which was captured in the Z specification by $\forall r : Role \bullet (r, r) \notin RH^+$.

```
module accesscontrol/rbach

open accesscontrol/types
open accesscontrol/rbac

sig Hierarchy extends Core{
    RH: Role -> Role
} {
    RH in ROLES -> ROLES
    ROLES = User.UA + Role.RH + RH.Role + PA.PRMSBase
    no  ^RH & iden
}
```

Finally, the `evalRBACH` function takes an element of `Hierarchy` and an element of `Request` as input, and returns `Permit` if the corresponding `User` and `PRMBase` in the supplied `Request` — `req` — exists in the product of `UA`, with the reflexive transitive closure of `RH` and `PA` from the supplied instance of `Hierarchy` — `rbac` — being used; otherwise it returns `Deny`.

```
fun evalRBACH(rbac : Hierarchy, req : Request) :  EvalRes {
    ((req.u -> req.p) in ((rbac.UA).(*(rbac.RH)).(rbac.PA))) => Permit
    else Deny
}
```

4.3 An Alloy example

This section uses the running example, presented in terms of Alloy.

4.3.1 Definition of the Alloy instance

Signatures are used to represent the individual elements defined in the scenario, with `one sig` being used to create unique instances of the relevant elements.

```
module accesscontrol/example1
open accesscontrol/rbaccoretoxacml

one sig Morris, Austin, Rover, Triumph extends User {}
one sig Doctor, Nurse extends Role {}
one sig read, modify extends Action {}
one sig PrescribeDB extends Resource {}

one sig writePrescription extends PRMSBase {}
```

```
    {
        action = modify
        resource = PrescribeDB
    }

    one sig readPrescription extends PRMSBase {}
    {
        action = read
        resource = PrescribeDB
    }
```

4.3.2 RBAC Core instance

In turn, these signatures can be used to declare signatures representing the permissions and the core RBAC system conforming to the access control scenario outlined by the running example. The signatures are effectively constrained to concrete values.

```
one sig examplerbac extends Core {}
{
    UA = (Morris -> Doctor) + (Rover -> Doctor) +
             (Austin -> Nurse) +  (Triumph -> Nurse)
    PA = (Doctor -> readPrescription) +
             (Nurse -> readPrescription) + (Doctor -> writePrescription)
    USERS = Morris + Rover + Austin + Triumph
    ROLES = Doctor + Nurse
    PRMS =  readPrescription + writePrescription
}
```

4.3.3 RBAC Core test and results

It is then possible to declare a number of requests that can be evaluated by the RBAC system to determine if the user specified has sufficient permissions to perform the requested action.

```
    one sig reqMorrisWrite extends Request {}
    {
        u = Morris && p = writePrescription
    }

    one sig reqAustinWrite extends Request {}
    {
        u = Austin && p = writePrescription
    }
```

The following assertions are defined to allow checks to be made to establish if the users have the correct permissions in the RBAC representation. That is, we assert that Morris can write a prescription — captured by `assert evalMorrisWrite` — and that Austin cannot — captured by `assert evalAustinWriteF`. The `check evalMorrisWrite` and `check evalAustinWriteF` allow the execution of the assertions.

```
module accesscontrol/example1test

open accesscontrol/example1

assert evalMorrisWrite { evalRBAC[examplerbac, reqMorrisWrite ] = Permit }

check evalMorrisWrite

assert evalAustinWriteF { evalRBAC[examplerbac, reqAustinWrite ] = Deny }

check evalAustinWriteF
```

The command `check evalMorrisWrite` instructs the analyzer to search for counter examples to the assertion, `assert evalMorrisWrite`, within the scope of the instance. The results of running the checks, `evalMorrisWrite` and `evalAustinWriteF`, using the Alloy Analyzer is given below:

```
Executing "Check evalMorrisWrite"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   2075 vars. 185 primary vars. 4746 clauses. 24ms.
   No counterexample found. Assertion may be valid. 1ms.

Executing "Check evalAustinWriteF"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   2075 vars. 185 primary vars. 4745 clauses. 18ms.
   No counterexample found. Assertion may be valid. 1ms.
```

The above means that no counter-examples to the assertions were found within the scope of the instance analysed. The section stating the assertion may be valid is because the Alloy Analyzer only evaluates the assertions within the scope of the instance under consideration.

4.3.4 RBAC Hierarchy

It is possible to define a simple hierarchy that states that the doctor role inherits the permissions of the nurse role. The resulting `prescribeh` differs from the `examplerbac` above by having the relation `(Doctor -> readPrescription)` removed from the `PA` relation and adding the relation `RH` containing `(Doctor -> Nurse)`.

```
one sig prescribeh extends Hierarchy {
}
{
    UA = (Morris -> Doctor) + (Rover -> Doctor) +
            (Austin -> Nurse) +  (Triumph -> Nurse)
    PA = (Nurse -> readPrescription) + (Doctor -> writePrescription)
    RH = (Doctor -> Nurse)
    USERS = Morris + Rover + Austin + Triumph
    ROLES = Doctor + Nurse
    PRMS =  readPrescription + writePrescription
}
```

4.3.5 RBAC Hierarchy test and results

It is possible to use the same requests to test if the hierarchy representation of the access control policy is consistent with the presented scenario. The only changes necessary for the assertions is to use the `EvalRBACH` function instead of the function`EvalRBAC` and to use the `prescribeh` instance of a hierarchy.

```
assert evalMorrisWrite {  evalRBACH[prescribeh, reqMorrisWrite ] = Permit }

check evalMorrisWrite

assert evalAustinWriteF {  evalRBACH[prescribeh, reqAustinWrite ] = Deny }

check evalAustinWriteF
```

In a similar way to that of Section 4.3.3, the assertions are checked using the Alloy Analyzer which results in no counter-examples being found. These results are shown below:

```
Executing "Check evalMorrisWrite"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   5062 vars. 369 primary vars. 10413 clauses. 100ms.
   No counterexample found. Assertion may be valid. 3ms.

Executing "Check evalAustinWriteF"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   5062 vars. 369 primary vars. 10477 clauses. 59ms.
   No counterexample found. Assertion may be valid. 3ms.
```

4.4 Summary

This chapter has presented two formal models of RBAC: one in Z and one in Alloy. The Z model is presented in the schema notation to avoid problems caused by the semi-formal approach that

are inherent in the model of [66]. The Alloy model covers the core and hierarchy aspects of RBAC which are necessary for the analysis of the translation process presented in Chapter 6.

In addition to supporting the translation process, the formal models of RBAC can also be used to reason about policies. An example of this could be to utilise the Alloy representation of a policy to validate that a role hierarchy is consistent and does not contain any cycles.

A small running example has been used as a concrete instance to illustrate the formal models, with the Z model being evaluated by walking through the example manually and the Alloy model being evaluated by the Alloy Analyzer. As the Alloy model was derived from the Z model, the results from the Alloy Analyzer provides some additional confidence in the Z model. The confidence in the model is due to the fact that the Alloy Analyzer exhaustively checks a state space for counter-examples that invalidate the model. The translation between Z and Alloy is further discussed in Chapter 9.

The following chapter presents formal models of XACML in Z and Alloy. These models will be used as the target of the RBAC-to-XACML translation process, which is described formally in Chapter 6.

Chapter 5

Formal representations of XACML

In pursuit of the goals of Chapter 3, this chapter presents two formal models of the XACML standard; the first is in terms of the Z notation and the second is an Alloy version of XACML sufficient to support the translation of Chapter 6. The Z and Alloy models presented are extensions of the models presented in [98].

5.1 A formal model of XACML in Z

This section presents a model of XACML in terms of Z. It first introduces the basic types needed to model XACML in Z. This is followed by a model for a target and models for the basic XACML components — rules, policies and policy sets. Finally, a schema representing XACML is presented, as well as a specialisation which captures the conditions that need to be met to be a valid instance of XACML.

5.1.1 Types

Initially, it is necessary to define the basic and free types needed to underpin the Z descriptions of the various components that make up a XACML policy description as well as permit the definition of a request.

The following types provide identifiers for any defined policy set, policy, rule, target or request respectively.

$$[PolicySetID, PolicyID, RuleID, TargetID, RequestID]$$

For the sake of simplicity, the elements that make up a target and request — namely, subjects, actions, resources, and environments — are modelled as basic types. This results in the abstraction that each member of the type *Subject* represents an entire subject block such as may be found in a request, as well as any subject attributes that may be necessary for supporting a target-matching decision.

It is then possible to treat *Action*, *Resource* and *Environment* in a similar way. An action is simply an operation on a resource. A resource is data, a service or a system component. Finally, the environment can contain other information about the state of the system, such as the current time.

To allow for the translation from RBAC to XACML the *Action* and *Resource* types of Chapter 4 will be used. To maintain consistency with the XACML standard which uses *Subject* rather than *User*, the Chapter 4 type *User* is utilised and the abbreviation below is defined. In addition *Environment* is also defined.

$$Subject == User$$

$$[Environment]$$

Finally *Obligation* is introduced:

$$[Obligation]$$

The type *Obligation* contains those activities that the PEP must undertake before access is permitted. As obligations are optional components, these activities have no relevance on the discourse, so it is appropriate to abstract away from the details.

Next, the free type *PolicyRef* is introduced which deals with the fact that a policy set can reference instances of both policies and policy sets. To this end, the constructor functions take elements of *PolicyID* and *PolicySetID* as arguments.

$$PolicyRef ::= Pol\langle\langle PolicyID\rangle\rangle \mid PolSet\langle\langle PolicySetID\rangle\rangle$$

It is necessary to introduce two free types to handle the evaluation of elements. First, *Effect* contains the possible effects that can be returned from the evaluation of rules, policies and policy sets. Second, *EvalRes* reflects the evaluation of a condition within the context of a specific request, as well as the possible outcomes of a match operation between a request and a target. In addition, rules and policies are associated with particular subsets of *Effect* to take into account different combining algorithms defined within the standard. The *IndeterminateDeny* and *IndeterminatePermit* values are included to support the behaviour of rule combining algorithms defined in the standard. In some cases the potential result that a rule would have given can be taken into account by the combining algorithm even though the rule returned *Indeterminate*. For example,

$$Effect ::= Permit \mid Deny \mid NotApplicable \mid Indeterminate \mid IndeterminateDeny \mid IndeterminatePermit$$

$$EvalRes ::= TRUE \mid FALSE \mid INDETERMINATE$$

$$EffectRule == Effect \setminus \{Indeterminate\}$$

$$EffectPolicy == Effect \setminus \{IndeterminateDeny, IndeterminatePermit\}$$

The final two free types that are considered define the identifiers for the different rule and policy combining algorithms, with the algorithms themselves being defined in Section 5.2.3. Only the basic combining algorithms touched on in [73] have been included. It would however be perfectly possible to define any number of other combining strategies to suit a given situation.

$$RulComAlgID ::= rulPermitOverride \mid rulDenyOverride \mid rulFirstApplicable$$
$$PolComAlgID ::= polPermitOverride \mid polDenyOverride \mid polFirstApplicable$$

5.1.2 Targets

The *Target* schema is an abstraction of the target element that is defined in [73]. The following is a summary of the description of Target in [73], "a Target is a conjunctive sequence of Subjects, Resources, Actions and Environments." The following discussion considers just subjects; the other elements behave similarly. A Subjects element is further described as a disjunctive sequence of Subject elements, with the Subject element being a conjunctive sequence of SubjectMatch elements. The SubjectMatch element defines a matching function and the element in the request context it should be applied to.

This has led to the choice to model the *Subject*, *Action*, *Resource* and *Environment* elements within a Target as functions mapping members of the basic type used within a request to elements of *EvalRes*. Each of the sections of an XACML target — subjects, actions, resources and environments — is represented in the model by a sequence of these functions. For example the functions in the sequence *sub* each represent an XACML subject element. The function is effectively abstracting away the conjunctive sequence of SubjectMatch elements. A similar representation is used for *act*, *res* and *env*. The use of the function is explained in more detail in Section 5.2.1.

Target

$tid : TargetID$
$sub : \mathrm{seq}\,(Subject \rightarrow EvalRes)$
$act : \mathrm{seq}\,(Action \rightarrow EvalRes)$
$res : \mathrm{seq}\,(Resource \rightarrow EvalRes)$
$env : \mathrm{seq}\,(Environment \rightarrow EvalRes)$

It is possible when writing an XACML policy to allow any of the sections pertaining to subjects, resources, actions or environments to be omitted. In such cases, the semantics of XACML requires that any request should match that part of the target.

In order to define a target it is necessary to first define the functions that will comprise the various blocks within the target. As an example, consider a function that would match a subject identified as Morris. If the function was named *matchMorris* it could be defined to return *TRUE* if the value of *Subject* presented as a parameter was *Morris*, and *FALSE* in all other cases, as shown below.

$$
\begin{array}{|l}
Morris : Subject \\
matchMorris : Subject \rightarrow EvalRes \\
\hline
\forall\, r : Subject \bullet r = Morris \Rightarrow matchMorris\, r = TRUE \\
\forall\, r : Subject \bullet r \neq Morris \Rightarrow matchMorris\, r = FALSE
\end{array}
$$

Utilising this function, it is possible to define an instance of *Target* that only matches *Morris*. Such a *Target* could have an identifier of *Tid_IsItMorris* and a subject of $\langle matchMorris \rangle$. The empty sequences assigned to *act*, *res* and *env* indicate that those blocks will match any supplied values as described above. The *Target* identified as *Tid_IsItMorris* is shown below.

$$
\begin{array}{|l}
Tid\_IsItMorris : TargetID \\
T\_IsItMorris : Target \\
\hline
T\_IsItMorris.tid = Tid\_IsItMorris \\
T\_IsItMorris.sub = \langle matchMorris \rangle \\
T\_IsItMorris.act = \langle\rangle \\
T\_IsItMorris.res = \langle\rangle \\
T\_IsItMorris.env = \langle\rangle
\end{array}
$$

It is desirable to define *empty_target*, which will allow the reasoning about rules that have no associated target value. This will allow any request to match the target, meaning any restrictions on the applicability of the rule will have to be via a stricter target at the policy or policy set level. The *empty_target* has a target identifier of *empty_target_id* and each of its component parts is defined as an empty sequence.

$$
\begin{array}{|l}
empty\_target\_id : TargetID \\
empty\_target : Target \\
\hline
empty\_target.tid = empty\_target\_id \\
empty\_target.sub = \langle\rangle \\
empty\_target.act = \langle\rangle \\
empty\_target.res = \langle\rangle \\
empty\_target.env = \langle\rangle
\end{array}
$$

5.1.3 Rules

A rule is the most fundamental part of a policy. A rule is composed of: an identifier, which we denote as *rid*; a target for which the rule is applicable; a condition that is evaluated in relation to a given request; and an effect that is returned if the condition associated with the rule evaluates to true. If no *condition* is included, which is modelled as an empty sequence, then it defaults to *TRUE* and the effect of the rule is returned if the target is matched.

The effect can only have the values *Permit* or *Deny*: if the condition evaluates to *FALSE* or the target is not matched, then the rule would return an effect of *NotApplicable*; if the evaluation were to fail for any reason, then a value of *Indeterminate* would be returned.

Rule

$rid : RuleID$
$target : TargetID$
$condition : \text{seq}\,(Request \rightarrow EvalRes)$
$effect : EffectRule$

$effect \in \{Permit, Deny\}$

It is possible to define a rule that denies everything, that is, regardless of the request it is evaluated against, it returns *Deny*. Such a rule is useful for guaranteeing a response is returned from a policy in response to an access control request. Such a rule, identified as *Rid_DenyAll*, is shown below.

$Rid\_DenyAll : RuleID$
$R\_DenyAll : Rule$

$R\_DenyAll.rid = Rid\_DenyAll$
$R\_DenyAll.target = empty\_target\_id$
$R\_DenyAll.condition = \langle\rangle$
$R\_DenyAll.effect = Deny$

The *target* of *empty_target_id* ensures that the rule is universally applicable, as the *condition* is an empty sequence the *effect* of the rule is returned — in this case, *Deny*. In a similar way, a rule to allow everyone to do everything could be defined as:

$Rid\_PermitAll : RuleID$
$R\_PermitAll : Rule$

$R\_PermitAll.rid = Rid\_PermitAll$
$R\_PermitAll.target = empty\_target\_id$
$R\_PermitAll.condition = \langle\rangle$
$R\_PermitAll.effect = Permit$

5.1.4 Policies

Policies are used by XACML to combine rules to give an overall effect. A *policy* consists of several aspects, namely: an identifier, *pid*, to uniquely identify it; a target for which the policy is applicable; a sequence of *RuleIDs*; and a rule combining algorithm, defining how the effects of the sequence of rules should be combined. Finally, the *Policy* schema also contains a set of obligations — which may be empty.

Policy

$pid : PolicyID$
$target : TargetID$
$rca : RulComAlgID$
$inPol : \text{seq}\, RuleID$
$obli : \mathbb{P}\, Obligation$

Note that it is possible, if undesirable, for *inPol* to be empty as the XACML standard does not preclude this. Further, it is also possible for the same rule identifier to appear more than once within *inPol*.

It is now possible to define two policies: the first of which will always return *Permit* if the subject of the request is *Morris* and the second will always return *Deny* whomever the subject of the request is.

$$
\begin{array}{|l}
Pid\_MorrisCanDoEverything : PolicyID \\
P\_MorrisCanDoEverything : Policy \\
\hline
P\_MorrisCanDoEverything.pid = Pid\_MorrisCanDoEverything \\
P\_MorrisCanDoEverything.target = Tid\_IsItMorris \\
P\_MorrisCanDoEverything.rca = rulPermitOverride \\
P\_MorrisCanDoEverything.inPol = \langle Rid\_PermitAll \rangle \\
P\_MorrisCanDoEverything.obli = \emptyset
\end{array}
$$

In *P_MorrisCanDoEverything*, shown above, the *target* is identified by the *Tid_IsItMorris* which restricts the applicability of the policy to the subject *Morris*. The rule combining algorithm chosen, *rulPermitOverride*, ensures the result of the policy is *Permit* if any rule in *inPol* returns *Permit*; in the case illustrated above, the single rule *Rid_PermitAll* in *inPol* will always return *Permit*, thus ensuring that the policy returns *Permit*.

In contrast the policy shown below, *P_DenyEveryone* will return a *Deny* for any request. This is because the target of the policy is *empty_target_id*, which matches any request. Even though the rule-combining algorithm chosen is *rulPermitOverride*, the policy only has one rule in *inPol* which always returns *Deny* as discussed above. This gives the overall effect that the policy will always return *Deny*.

$$
\begin{array}{|l}
Pid\_DenyEveryone : PolicyID \\
P\_DenyEveryone : Policy \\
\hline
P\_DenyEveryone.pid = Pid\_DenyEveryone \\
P\_DenyEveryone.target = empty\_target\_id \\
P\_DenyEveryone.rca = rulPermitOverride \\
P\_DenyEveryone.inPol = \langle Rid\_DenyAll \rangle \\
P\_DenyEveryone.obli = \emptyset
\end{array}
$$

5.1.5 Policy sets

In XACML policy sets are use to combine policies and other policy sets into another component of an XACML policy.

The *PolicySet* schema consists of five components. First, *psid* acts as a unique identifier for a particular policy set. Second, *target* represents the target for which the policy set is applicable. Next, *pca* identifies the relevant policy-combining algorithm; that is, it determines how the effects of the sequence of policies and (further) policy sets associated with a policy set are combined. The sequence of elements of type *PolicyRef* — which are constructed from elements of both *PolicyID* and *PolicySetID* — allows a particular policy set to effectively contain a sequence of

both policies and policy sets. Finally, a policy set may contain an optional set of obligations.

$$
\begin{array}{|l}
\textit{PolicySet} \\
\hline
psid : \textit{PolicySetID} \\
target : \textit{TargetID} \\
pca : \textit{PolComAlgID} \\
inPolSet : \mathrm{seq}\ \textit{PolicyRef} \\
obli : \mathbb{P}\ \textit{Obligation} \\
\hline
\end{array}
$$

Again, it is possible, if undesirable, for *inPolSet* to be empty. Note that *inPolSet* (and, indeed, *inPol* in *Policy*) is defined as a sequence, rather than a set, to deal with those cases in which the order of rules (or policy references) is important. Such cases include the application of the first applicable combining algorithm.

The policy set *PS_PermitMorris* shown below illustrates how the previously defined policies could be combined to define a policy set which has the overall effect of denying access to everyone apart from *Morris*. The policy set is universally applicable as the *target* is *empty_target_id*. The policy-combining algorithm, *polPermitOverride*, will return *Permit* if any of the members of *inPolSet* returns *Permit*; otherwise it will behave as described in Section 5.2.3. The members of *inPolSet* are $Pol(Pid\_MorrisCanDoEverything)$ and $Pol(Pid\_DenyEveryone)$ which are references to the two policies, *Pid_MorrisCanDoEverything* and *Pid_DenyEveryone*, described above. From the previous descriptions the policy *Pid_DenyEveryone* will always result in *Deny* and *Pid_MorrisCanDoEverything* will only return *Permit* if the request is from *Morris*. This gives the policy set the desired effect. A more detailed description of the evaluation of an XACML policy is given in Section 5.2.2.

$$
\begin{array}{|l}
PSid\_PermitMorris : \textit{PolicySetID} \\
PS\_PermitMorris : \textit{PolicySet} \\
\hline
PS\_PermitMorris.psid = PSid\_PermitMorris \\
PS\_PermitMorris.target = empty\_target\_id \\
PS\_PermitMorris.pca = polPermitOverride \\
PS\_PermitMorris.inPolSet = \langle Pol(Pid\_MorrisCanDoEverything), Pol(Pid\_DenyEveryone)\rangle \\
PS\_PermitMorris.obli = \emptyset \\
\end{array}
$$

Although repetition of elements of both *PolicyID* and *PolicySetID* within a policy set is permissible, repetition of elements of the latter can be problematic. This is due to the fact that such a situation might result in a recursive referencing of policy sets. If such a policy were to be implemented this cycle of references might lead to an infinitely recursing process. Of course, one of the benefits afforded by a formal model is that we have the possibility of checking for the absence of such cycles — as well as other healthiness conditions that we might impose on policies.

A number of optional elements for policies and policy sets are identified in [73], e.g., combiner parameters. These optional elements have been omitted from this description as they add little of value to the narrative. Of course, it would be perfectly feasible to add additional aspects should they be deemed to be necessary in the future.

5.1.6 XACML schema

This section defines the *XACML* schema. It is effectively a collection of identity functions and a set of policy references. The identity functions — *getPolicySet*, *getPolicy*, *getRule* and *getTarget* — map identifiers to instances of policy sets, policies, rules and targets respectively. The restrictions on the functions serve to ensure that each function maps an identifier to a schema containing the same identifier. The set of policy references *rootPol*, defines the top-level policies and policy sets that are to be used to evaluate the overall effect of an XACML policy with respect to an access control request.

$$
\begin{array}{|l}
\hline
XACML \\
\hline
getPolicySet : PolicySetID \rightarrowtail PolicySet \\
getPolicy : PolicyID \rightarrowtail Policy \\
getRule : RuleID \rightarrowtail Rule \\
getTarget : TargetID \rightarrowtail Target \\
rootPol : \mathbb{P}\, PolicyRef \\
\hline
(\forall psi : PolicySetID \mid psi \in \mathrm{dom}\, getPolicySet \bullet (getPolicySet\, psi).psid = psi) \\
(\forall pi : PolicyID \mid pi \in \mathrm{dom}\, getPolicy \bullet (getPolicy\, pi).pid = pi) \\
(\forall ri : RuleID \mid ri \in \mathrm{dom}\, getRule \bullet (getRule\, ri).rid = ri) \\
(\forall ti : TargetID \mid ti \in \mathrm{dom}\, getTarget \bullet (getTarget\, ti).tid = ti) \\
empty\_target\_id \mapsto empty\_target \in getTarget \\
\hline
\end{array}
$$

Each function is injective to guarantee uniqueness of identifiers. Note that this restriction is not enforced by Page 44 of the XACML standard [73]:

> "It is the responsibility of the PAP to ensure that no two policies visible to the PDP have the same identifier. This MAY [original authors' capitals] be achieved by following a predefined URN or URI scheme. If the policy set identifier is in the form of a URL, then it MAY [original author's capitals] be resolvable."

Additionally, it is worth noting that if multiple policies or policy sets referenced in *rootPol* are applicable to a particular access control request, the behaviour is undefined as there is no XACML level combining algorithm defined within the standard.

5.1.7 Valid XACML

This section presents a *ValidXACML* schema, which constrains the XACML schema to display what might be considered valid behaviour.

ValidXACML

XACML

$$
\begin{aligned}
&\forall\, ps : \operatorname{ran} getPolicySet \bullet \forall\, s : \operatorname{ran} ps.inPolSet \bullet \\
&\quad ((s \in \operatorname{ran} PolSet \land (PolSet^{\sim} s) \in (\operatorname{dom} getPolicySet)) \\
&\quad \lor \\
&\quad (s \in \operatorname{ran} Pol \land (Pol^{\sim} s) \in \operatorname{dom} getPolicy)) \\
&\quad \land \\
&\quad ps.target \in \operatorname{dom} getTarget \\
&\forall\, p : \operatorname{ran} getPolicy \bullet \forall\, r : \operatorname{ran} p.inPol \bullet r \in \operatorname{dom} getRule \land p.target \in \operatorname{dom} getTarget \\
&\forall\, r : \operatorname{ran} getRule \bullet r.target \in \operatorname{dom} getTarget \\
&\exists\, pslinks : PolicySetID \leftrightarrow PolicySetID \bullet \\
&\quad pslinks = \bigcup \{ps : \operatorname{ran} getPolicySet \bullet \\
&\qquad\qquad \{s : \operatorname{ran} ps.inPolSet \mid s \in \operatorname{ran} PolSet \bullet ps.psid \mapsto PolSet^{\sim} s\}\} \\
&\quad \land \\
&\quad pslinks^{+} \cap \operatorname{id} PolicySetID = \emptyset \\
&\quad \land \\
&\quad pslinks^{*}(\!|\, PolSet^{\sim}(\!|\, rootPol \,|\!)\,|\!) = \operatorname{dom} getPolicySet \\
&rootPol \neq \emptyset \\
&\forall\, pref : rootPol \bullet pref \in \operatorname{ran} Pol \lor pref \in \operatorname{ran} PolSet \\
&\forall\, pid : \operatorname{dom} getPolicy \bullet Pol\ pid \in rootPol \lor \\
&\quad (\exists\, ps : \operatorname{ran} getPolicySet \bullet Pol\ pid \in \operatorname{ran} ps.inPolSet) \\
&\forall\, rid : \operatorname{dom} getRule \bullet \exists\, p : \operatorname{ran} getPolicy \bullet rid \in \operatorname{ran} p.inPol \\
&\forall\, tid : \operatorname{dom} getTarget \bullet \\
&\quad (\exists\, ps : \operatorname{ran} getPolicySet \bullet ps.target = tid) \\
&\quad \lor \\
&\quad (\exists\, p : \operatorname{ran} getPolicy \bullet p.target = tid) \\
&\quad \lor \\
&\quad (\exists\, r : \operatorname{ran} getRule \bullet r.target = tid) \\
&\quad \lor \\
&\quad tid = empty\_target\_id
\end{aligned}
$$

The consistency of the *ValidXACML* schema are considered in the following.

- The first predicate constrains policy sets returned by the function *getPolicySet* to be only those which have an associated *inPolSet* that contain references to either policy sets from *getPolicySet* or policies from *getPolicy*. Additionally, it ensures that the target attribute references a target from *getTarget*.
- Similarly, the second predicate constrains every policy obtained from *getPolicy* to have only references to rules from *getRule* in their *inPol* and to have a target which is obtainable from *getTarget*.
- Next, a constraint is added that every rule has to have a target obtainable from the function *getTarget*.
- The next predicate defines a set which links policy sets to the policy sets that they reference through *inPolSet*. This set of relations is then used to check there are no cycles, as well as

checking all the defined policy sets are used. The check for defined policy sets is performed by taking the relational image of the reflexive transitive closure of *pslinks* relative to the relational image of the inverse of *PolSet* over the contents of *rootPol*; the result of this is the set of *PolicySet* references that are reachable by inclusion in *rootPol* or are reachable by virtue of being included within an *inPolSet* of a policy set that is included. Further, the constraint insists that these are the only policy sets contained within the function *getPolicySet*.

- $rootPol \neq \emptyset$ ensures that *rootPol* contains at least one policy or policy set.
- The next predicate constrains the members of *rootPol* to be either references to policies or policy sets.
- Every policy obtained from the *getPolicy* function is either a member of *rootPol* or is in an *inPolSet* contained within a policy set.
- Every rule obtainable from *getRule* must be referenced from an *inPol* in a policy.
- Finally a constraint on *getTarget* is defined to so that it contains only those targets referenced by policy sets, policies or rules, or to be the empty target.

5.2 XACML requests and evaluation of policies

5.2.1 Requests

A request is modelled as consisting of a request identifier along with sets made up of elements from the types *Subject*, *Action*, *Resource*, and *Environment*.

Request

$$
\begin{aligned}
&request : RequestID \\
&sub : \mathbb{P}_1\ Subject \\
&act : \mathbb{P}_1\ Action \\
&res : \mathbb{P}_1\ Resource \\
&env : \mathbb{P}\ Environment \\
\hline
&\#env \leq 1 \\
&\#act = 1
\end{aligned}
$$

The request, then, effectively represents a subject asking to perform an action on a resource and also captures any relevant environment information.

The request *morrisWrite*, shown below, captures a request by *Morris* to perform a *write* action on the *prescribeDB*.

$$
\begin{array}{|l}
Rq\_MorrisWrite : RequestID \\
morrisWrite : Request \\
\hline
morrisWrite.request = Rq\_MorrisWrite \\
morrisWrite.sub = \{Morris\} \\
morrisWrite.act = \{write\} \\
morrisWrite.res = \{prescribeDB\} \\
morrisWrite.env = \emptyset
\end{array}
$$

While it is valid to have multiple subject and resource elements, [73] insists that only one action element and one environment element are defined within a request. It should be noted that empty elements, such as `<Environment/>`, are members of their respective type — in this instance, *Environment*.

The generic definition of *Match* defines functions that take a sequence of functions and a set of elements and returns an *EvalRes*: the result is *TRUE* if any of the set of elements when applied to any of the functions return *TRUE*; if no *TRUE* result is found, *INDETERMINATE* is checked for and returned if found; otherwise *FALSE* is returned.

The first clause, $T = \langle\rangle$, states that if an empty sequence is provided it matches any request. The second clause checks if any function is *TRUE* when applied to the elements of the request. As this is modelling a disjunctive sequence, a single match will result in *TRUE* being returned. Next, if there is no *TRUE* result, but one of the function applications returns *INDETERMINATE*, a result of *INDETERMINATE* is returned. This may have occurred because the request does not contain a value matching that required by a function. For example a function needs to check the X.509 certificate Distinguished Name but the request does not contain an element matching this description. Finally if all the results are *FALSE*, then *FALSE* is returned.

$$
\begin{array}{|l}
[X] \\
match : \mathrm{seq}(X \rightarrow EvalRes) \times \mathbb{P}\, X \rightarrow EvalRes \\
\hline
\forall\, T : \mathrm{seq}(X \rightarrow EvalRes);\ R : \mathbb{P}\, X \bullet \\
\quad T = \langle\rangle \Rightarrow match(T, R) = TRUE \\
\quad \land \\
\quad (T \neq \langle\rangle \land (\exists f : \mathrm{ran}\ T;\ r : R \bullet f\ r = TRUE)) \Rightarrow \\
\qquad match(T, R) = TRUE \\
\quad \land \\
\quad (T \neq \langle\rangle \land \neg\, (\exists f : \mathrm{ran}\ T;\ r : R \bullet f\ r = TRUE) \land \\
\quad (\exists f : \mathrm{ran}\ T;\ r : R \bullet f\ r = INDETERMINATE)) \Rightarrow \\
\qquad match(T, R) = INDETERMINATE \\
\quad \land \\
\quad (T \neq \langle\rangle \land \neg\, (\exists f : \mathrm{ran}\ T;\ r : R \bullet f\ r = TRUE) \land \\
\quad \neg\, (\exists f : \mathrm{ran}\ T;\ r : R \bullet f\ r = INDETERMINATE)) \Rightarrow \\
\qquad match(T, R) = FALSE
\end{array}
$$

It is then possible to define *matchTarget* as follows. This function takes a *Target* and a *Request* and returns a result from *EvalRes*. The function *matchTarget* returns a value of *TRUE* if all the component of the target are matched by corresponding components of the request. The

value *INDETERMINATE* is returned if any of the matches between a target component and the corresponding request component return *INDETERMINATE*. Finally the value *FALSE* is returned if there is no match returning *INDETERMINATE* and at least one match fails to return *TRUE*.

$$
\begin{array}{|l}
matchTarget : Target \times Request \pfun EvalRes \\
\hline
\forall\, t : Target;\ r : Request \bullet \\
\quad match(t.sub, r.sub) = TRUE \land \\
\quad match(t.act, r.act) = TRUE \land \\
\quad match(t.res, r.res) = TRUE \land \\
\quad match(t.env, r.env) = TRUE \Rightarrow \\
\qquad matchTarget(t, r) = TRUE \\
\quad \land \\
\quad (match(t.sub, r.sub) = INDETERMINATE \lor \\
\quad match(t.act, r.act) = INDETERMINATE \lor \\
\quad match(t.res, r.res) = INDETERMINATE \lor \\
\quad match(t.env, r.env) = INDETERMINATE) \Rightarrow \\
\qquad matchTarget(t, r) = INDETERMINATE \\
\quad \land \\
\quad (\neg\, (match(t.sub, r.sub) = INDETERMINATE \lor \\
\qquad match(t.act, r.act) = INDETERMINATE \lor \\
\qquad match(t.res, r.res) = INDETERMINATE \lor \\
\qquad match(t.env, r.env) = INDETERMINATE) \\
\qquad \land \\
\qquad \neg\, (match(t.sub, r.sub) = TRUE \land \\
\qquad\qquad match(t.act, r.act) = TRUE \land \\
\qquad match(t.res, r.res) = TRUE \land \\
\qquad match(t.env, r.env) = TRUE)) \Rightarrow \\
\qquad\quad matchTarget(t, r) = FALSE
\end{array}
$$

5.2.2 The evaluation of XACML

To evaluate an XACML policy, each level of component must be evaluated, with the effects being propagated and combined by the relevant combining algorithm.

For any given request, the policies or policy sets referenced by each *PolicyRef* in *rootPol* are tested to see which has a *Target* that matches the *Request*. In [73] it is assumed that a single policy/policy set will be matched. The matching *PolicySet* is then evaluated in relation to the *Request*. This involves evaluating the appropriateness of each element in the sequence of policy references in the *PolicySet*. Each is checked to see if it matches the request and the results combined by the appropriate combining algorithm — examples of possible combining strategies are illustrated in Figure 5.1. The *PolicyRef* is expanded eventually to a sequence of *Rule*s, the effects of which are combined using a combining algorithm. The results propagate back up, to give a single *Effect* of applying a given *Request* to a given *PolicySet*, which ultimately represents the effect of the XACML policy.

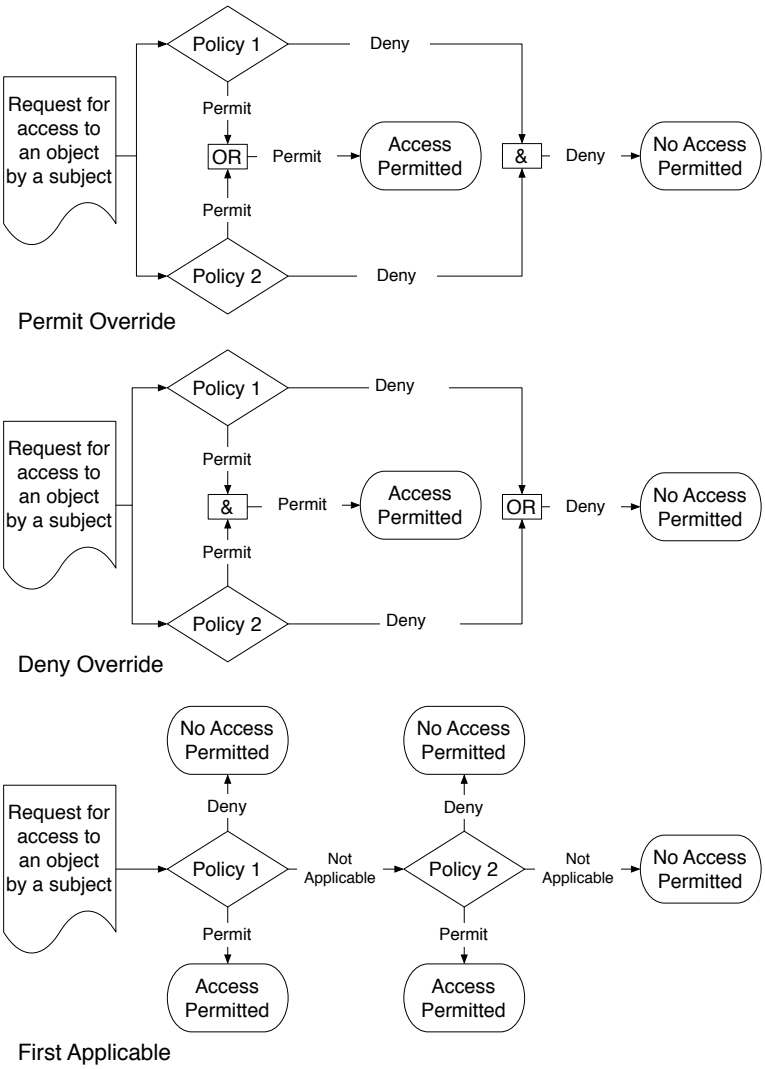

Figure 5.1: Effects of different combining algorithms

The next few subsections describe this process using a Z representation.

5.2.3 Combining algorithms

This subsection introduces the Z definitions of several combining algorithms. Following the definitions from [73] there exist fundamental differences between the rule and policy combining algorithms because the former have to take into account the possible results *IndeterminatePermit* and *IndeterminateDeny* whereas the latter only have a single *Indeterminate* to cope with.

The following defines a selection of combining algorithms as functions operating on a sequence of *Effect*s that return a single *Effect*.

Initially the rule-combining algorithms are considered, the first of which is 'rule permit override'. This algorithm returns *Permit* if any of the effects passed to it are *Permit*. It is worth

noting that if an effect of *IndeterminatePermit* exists it takes precedence over the effect *Deny*.

$$
\begin{array}{|l}
rulPerOvrAlg : \operatorname{seq} EffectRule \rightarrow EffectPolicy \\
\hline
\forall\, s : \operatorname{seq} EffectRule \bullet \\
\quad (\operatorname{ran} s = \emptyset \Rightarrow rulPerOvrAlg\ s = Indeterminate) \land \\
\quad (Permit \in \operatorname{ran} s \Rightarrow rulPerOvrAlg\ s = Permit) \land \\
\quad (Permit \notin \operatorname{ran} s \land IndeterminatePermit \in \operatorname{ran} s \Rightarrow rulPerOvrAlg\ s = Indeterminate) \land \\
\quad (Permit \notin \operatorname{ran} s \land IndeterminatePermit \notin \operatorname{ran} s \land Deny \in \operatorname{ran} s \Rightarrow rulPerOvrAlg\ s = Deny) \land \\
\quad (Permit \notin \operatorname{ran} s \land IndeterminatePermit \notin \operatorname{ran} s \land Deny \notin \operatorname{ran} s \\
\qquad \land IndeterminateDeny \in \operatorname{ran} s \Rightarrow rulPerOvrAlg\ s = Indeterminate) \land \\
\quad (Permit \notin \operatorname{ran} s \land IndeterminatePermit \notin \operatorname{ran} s \land Deny \notin \operatorname{ran} s \\
\qquad \land IndeterminateDeny \notin \operatorname{ran} s \Rightarrow rulPerOvrAlg\ s = NotApplicable)
\end{array}
$$

In a similar way, 'rule deny override' will return *Deny* if any of the effects passed to it contains a *Deny*.

$$
\begin{array}{|l}
rulDenOvrAlg : \operatorname{seq} EffectRule \rightarrow EffectPolicy \\
\hline
\forall\, s : \operatorname{seq} Effect \bullet \\
\quad (\operatorname{ran} s = \emptyset \Rightarrow rulDenOvrAlg\ s = Indeterminate) \land \\
\quad (Deny \in \operatorname{ran} s \Rightarrow rulDenOvrAlg\ s = Deny) \land \\
\quad (Deny \in \operatorname{ran} s \land IndeterminateDeny \in \operatorname{ran} s \Rightarrow rulDenOvrAlg\ s = Indeterminate) \land \\
\quad (Deny \notin \operatorname{ran} s \land IndeterminateDeny \notin \operatorname{ran} s \land Permit \in \operatorname{ran} s \Rightarrow rulDenOvrAlg\ s = Permit) \land \\
\quad (Deny \notin \operatorname{ran} s \land IndeterminateDeny \notin \operatorname{ran} s \land Permit \notin \operatorname{ran} s \\
\qquad \land IndeterminatePermit \in \operatorname{ran} s \Rightarrow rulDenOvrAlg\ s = Indeterminate) \land \\
\quad (Deny \notin \operatorname{ran} s \land IndeterminateDeny \notin \operatorname{ran} s \land Permit \notin \operatorname{ran} s \\
\qquad \land IndeterminatePermit \notin \operatorname{ran} s \Rightarrow rulDenOvrAlg\ s = NotApplicable)
\end{array}
$$

'Rule first applicable' will return the first effect that is not *NotApplicable* — if one exists.

$$
\begin{array}{|l}
rulFirAppAlg : \operatorname{seq} EffectRule \rightarrow EffectPolicy \\
\hline
\forall\, s : \operatorname{seq} EffectRule \bullet \\
\quad (\operatorname{ran} s = \emptyset \Rightarrow rulFirAppAlg\ s = Indeterminate) \land \\
\quad (\operatorname{ran} s = \{NotApplicable\} \Rightarrow rulFirAppAlg\ s = NotApplicable) \land \\
\quad ((\operatorname{ran}\ s \neq \{NotApplicable\}) \land (\operatorname{ran}\ s \neq \emptyset) \Rightarrow \\
\qquad (head(s \upharpoonright \{Permit, Deny, IndeterminateDeny, IndeterminatePermit\}) \in \{Permit, Deny\} \\
\qquad\quad \Rightarrow rulFirAppAlg\ s = head(s \upharpoonright \{Permit, Deny, IndeterminateDeny, IndeterminatePermit\}) \land \\
\qquad (head(s \upharpoonright \{Permit, Deny, IndeterminateDeny, IndeterminatePermit\}) \in \\
\qquad\quad \{IndeterminateDeny, IndeterminatePermit\} \Rightarrow rulFirAppAlg\ s = Indeterminate)))
\end{array}
$$

The following function takes the identifier of a rule-combining algorithm and a sequence of effects, and returns the effect of applying the algorithm to the sequence.

$$getRuleCombAlg : RulComAlgID \rightarrow (\text{seq}\, Effect \rightarrow Effect)$$

$$getRuleCombAlg\,(rulPermitOverride) = rulPerOvrAlg$$
$$getRuleCombAlg\,(rulDenyOverride) = rulDenOvrAlg$$
$$getRuleCombAlg\,(rulFirstApplicable) = rulFirAppAlg$$

In a similar way, the policy-combining algorithms can be defined. The policy-combining algorithms are slightly simpler to define as they do not have to process multiple types if indeterminate as was the case with the rule-combining algorithms.

So, 'policy permit override' will return *Permit* if one exists in the supplied sequence whereas 'policy deny override' will return a *Deny* if one exists. It is also worth noting that 'policy deny override' will return a *Deny* instead of *Indeterminate*, whereas 'policy permit override' would return *Indeterminate*.

$$polPerOvrAlg : \text{seq}\, EffectPolicy \rightarrow EffectPolicy$$

$$\begin{aligned}
&\forall\, s : \text{seq}\, Effect \bullet \\
&\quad \text{ran}\, s = \emptyset \Rightarrow polPerOvrAlg\, s = Indeterminate \land \\
&\quad Permit \in \text{ran}\, s \Rightarrow polPerOvrAlg\, s = Permit \land \\
&\quad Permit \notin \text{ran}\, s \land Deny \in \text{ran}\, s \Rightarrow polPerOvrAlg\, s = Deny \land \\
&\quad Permit \notin \text{ran}\, s \land Deny \notin \text{ran}\, s \land Indeterminate \in \text{ran}\, s \Rightarrow \\
&\qquad polPerOvrAlg\, s = Indeterminate \land \\
&\quad Permit \notin \text{ran}\, s \land Deny \notin \text{ran}\, s \land Indeterminate \notin \text{ran}\, s \Rightarrow \\
&\qquad polPerOvrAlg\, s = NotApplicable
\end{aligned}$$

$$polDenOvrAlg : \text{seq}\, EffectPolicy \rightarrow EffectPolicy$$

$$\begin{aligned}
&\forall\, s : \text{seq}\, Effect \bullet \\
&\quad \text{ran}\, s = \emptyset \Rightarrow polDenOvrAlg\, s = Indeterminate \land \\
&\quad Deny \in \text{ran}\, s \Rightarrow polDenOvrAlg\, s = Deny \land \\
&\quad Deny \notin \text{ran}\, s \land Permit \in \text{ran}\, s \Rightarrow polDenOvrAlg\, s = Permit \land \\
&\quad Deny \notin \text{ran}\, s \land Permit \notin \text{ran}\, s \land Indeterminate \in \text{ran}\, s \Rightarrow \\
&\qquad polDenOvrAlg\, s = Deny \land \\
&\quad Deny \notin \text{ran}\, s \land Permit \notin \text{ran}\, s \land Indeterminate \notin \text{ran}\, s \Rightarrow \\
&\qquad polDenOvrAlg\, s = NotApplicable
\end{aligned}$$

The 'policy first applicable' will return the first effect that is not *NotApplicable* — if one exists.

$$polFirAppAlg : \text{seq}\, EffectPolicy \rightarrow EffectPolicy$$

$$\begin{aligned}
&\forall\, s : \text{seq}\, Effect \bullet \\
&\quad \text{ran}\, s = \emptyset \Rightarrow polFirAppAlg\, s = Indeterminate \land \\
&\quad \text{ran}\, s = \{NotApplicable\} \Rightarrow polFirAppAlg\, s = NotApplicable \land \\
&\quad (\text{ran}\ s \neq \{NotApplicable\}) \land (\text{ran}\ s \neq \emptyset) \Rightarrow \\
&\qquad polFirAppAlg\, s = head(s \upharpoonright \{Permit, Deny, Indeterminate\})
\end{aligned}$$

The following function takes the identifier of a policy-combining algorithm *PolComAlgID* and a sequence of *Effect* and returns the *Effect* of applying the algorithm on the sequence — working in a similar fashion to *getRuleCombAlg*.

$$getPolicyCombAlg : PolComAlgID \rightarrow (\text{seq}\, Effect \rightarrow Effect)$$

$$getPolicyCombAlg\,(polPermitOverride) = rulPerOvrAlg$$
$$getPolicyCombAlg\,(polDenyOverride) = rulDenOvrAlg$$
$$getPolicyCombAlg\,(polFirstApplicable) = rulFirAppAlg$$

5.2.4 The effect of a rule

The function *evalCondition* applies a request to a rule's condition. The condition is represented as a sequence of functions that take an element of *Request* and return an *EvalRes*. This models the condition as a Boolean expression that is evaluated to determine the effect of the rule. The possible values the function can return are *TRUE*, *FALSE* or *INDETERMINATE*, with the *INDETERMINATE* result being used to indicate either that the evaluation could not be performed or that an error occurred.

$$evalCondition : \text{seq}(Request \rightarrow EvalRes) \times Request \rightarrow EvalRes$$

$$\begin{aligned}
&\forall\, T : \text{seq}(Request \rightarrow EvalRes);\ r : Request \bullet \\
&\quad T = \langle\rangle \Rightarrow evalCondition(T, r) = TRUE \land \\
&\quad (T \neq \langle\rangle \land (\forall f : \text{ran}\, T \bullet f\, r = TRUE)) \Rightarrow \\
&\qquad evalCondition(T, r) = TRUE \land \\
&\quad (T \neq \langle\rangle \land \neg\, (\forall f : \text{ran}\, T \bullet f\, r = TRUE) \land \\
&\quad (\exists f : \text{ran}\, T \bullet f\, r = INDETERMINATE)) \Rightarrow \\
&\qquad evalCondition(T, r) = INDETERMINATE \land \\
&\quad (T \neq \langle\rangle \land \neg\, (\forall f : \text{ran}\, T \bullet f\, r = TRUE) \land \\
&\quad \neg\, (\exists f : \text{ran}\, T \bullet f\, r = INDETERMINATE)) \Rightarrow \\
&\qquad evalCondition(T, r) = FALSE
\end{aligned}$$

As an illustration, consider a function called *modifyData* (shown below) which takes an element of *Request* and returns an *EvalRes*: *modifyData* returns *TRUE* if the request has an *act* of *modify* and a *res* of *data*; otherwise, the function returns *FALSE*.

$$modifyData : Request \rightarrow EvalRes$$

$$\forall\, r : Request \bullet modify \in r.act \land data \in r.res \land \#r.res = 1 \Rightarrow modifyData\, r = TRUE$$
$$\forall\, r : Request \bullet modify \notin r.act \lor data \notin r.res \lor \#r.res \neq 1 \Rightarrow modifyData\, r = FALSE$$

If the *modifyData* function were to be included in the sequence that made up a condition in a *Rule*, it would be possible to evaluate that condition against a request. For example, the previously defined *Request*, *Rq_MorrisWrite*. Such an evaluation would result in $evalCondition(\langle modifydata\rangle, Rq\_MorrisWrite)$ returning *FALSE*: the *act* in *Rq_MorrisWrite* is *write* and the *res* is *prescribeDB* which fails to match *modify* and *data*, therefore the result of the evaluation of the function is *FALSE*, resulting in the evaluation of the condition being *False*.

evalRule evaluates the effect of a given rule in relation to a request. If the target of the request and the target of the rule match, then the condition is evaluated with respect to the request.

$$
\begin{array}{l}
evalRule : XACML \times Request \times Rule \rightarrow Effect \\
\hline
\forall\, x : XACML;\ req : Request;\ r : Rule \mid matchTarget(x.getTarget(r.target), req) = TRUE \bullet \\
\quad (evalCondition(r.condition, req) = TRUE \Rightarrow \\
\qquad evalRule(x, req, r) = r.effect \land \\
\quad evalCondition(r.condition, req) = FALSE \Rightarrow \\
\qquad evalRule(x, req, r) = NotApplicable \land \\
\quad evalCondition(r.condition, req) = INDETERMINATE \Rightarrow \\
\qquad ((r.effect = Permit \Rightarrow evalRule(x, req, r) = IndeterminatePermit) \\
\qquad \land (r.effect = Deny \Rightarrow evalRule(x, req, r) = IndeterminateDeny))) \\
\forall\, x : XACML;\ req : Request;\ r : Rule \mid matchTarget(x.getTarget(r.target), req) = FALSE \bullet \\
\quad evalRule(x, req, r) = NotApplicable \\
\forall\, x : XACML;\ req : Request;\ r : Rule \mid \\
\quad matchTarget(x.getTarget(r.target), req) = INDETERMINATE \bullet \\
\qquad (r.effect = Permit \Rightarrow evalRule(x, req, r) = IndeterminatePermit \\
\qquad \land\ r.effect = Deny \Rightarrow evalRule(x, req, r) = IndeterminateDeny)
\end{array}
$$

Consider the XACML policy *xacmlMed*, which is an element of *ValidXACML* and contains a rule *RuleWritePrescribe*, which is an element of *Rule*. If *RuleWritePrescribe* has the *effect* of *Permit* and has a target that matches the request *Rq_MorrisWrite* the evaluation of the rule against the request is captured by $evalRule(xacmlMed, Rq\_MorrisWrite, RuleWritePrescribe)$, which would return the effect *Permit*.

The function *processRuleSeq* is used to obtain a sequence of effects that result from a request being applied to a sequence of *RuleID*s. This is used by the *evalPolicy* function to supply input to the rule-combining algorithm.

$$
\begin{array}{l}
processRuleSeq : XACML \times Request \times \operatorname{seq} RuleID \rightarrow \operatorname{seq} Effect \\
\hline
\forall\, x : XACML;\ req : Request;\ sR : \operatorname{seq}\ RuleID \bullet \\
\quad processRuleSeq(x, req, sR) = \\
\qquad \{ele : \operatorname{dom} sR \bullet ele \mapsto evalRule(x, req, x.getRule(sR(ele)))\}
\end{array}
$$

5.2.5 The effect of a policy

The *evalPolicy* function returns the effect of applying a request to a *PolicyID*. If the request and policy targets match, then the rule-combining algorithm specified by the policy is used to combine the result of applying the request to the sequence of rules making up the policy. If the target matching returns *FALSE* or *INDETERMINATE*, then the returned effect is *NotApplicable* or *Indeterminate* respectively.

$$
\begin{array}{l}
evalPolicy : XACML \times Request \times PolicyID \rightarrow Effect \\
\hline
\forall\, x : XACML;\ req : Request;\ pid : PolicyID;\ pol : Policy \mid pol = x.getPolicy(pid) \bullet \\
\quad (matchTarget(x.getTarget(pol.target), req) = TRUE \Rightarrow \\
\quad\quad evalPolicy(x, req, pid) = \\
\quad\quad\quad getRuleCombAlg(pol.rca)(processRuleSeq(x, req, pol.inPol))) \land \\
\quad (matchTarget(x.getTarget(pol.target), req) = FALSE \Rightarrow \\
\quad\quad evalPolicy(x, req, pid) = NotApplicable) \land \\
\quad (matchTarget(x.getTarget(pol.target), req) = INDETERMINATE \Rightarrow \\
\quad\quad evalPolicy(x, req, pid) = Indeterminate)
\end{array}
$$

5.2.6 The effect of a policy set

The *evalPolicySet* function returns the effect of the application of a request to a *PolicySetID*. In doing this, the function traverses the sequence of policy references which may contain elements constructed from *PolicySetID*. Each *PolicyRef* is evaluated with respect to the request and the combined effect is returned, with the effects being combined using the policy-combining algorithm defined in the policy set.

$$
\begin{array}{l}
evalPolicySet : XACML \times Request \times PolicySetID \rightarrow Effect \\
\hline
\forall\, x : XACML;\ req : Request;\ psid : PolicySetID;\ pset : PolicySet \mid \\
\quad pset = x.getPolicySet(psid) \bullet \\
\quad\quad matchTarget((x.getTarget(pset.target), req)) = TRUE \Rightarrow \\
\quad\quad\quad evalPolicySet(x, req, psid) = \\
\quad\quad\quad\quad (getPolicyCombAlg(pset.pca) \\
\quad\quad\quad\quad\quad (\{n : \mathrm{dom}\, pset.inPolSet;\ ef : Effect \mid \\
\quad\quad\quad\quad\quad\quad (pset.inPolSet(n) \in \mathrm{ran}\, Pol \land \\
\quad\quad\quad\quad\quad\quad ef = evalPolicy(x, req, ((Pol^{\sim})(pset.inPolSet(n))))) \\
\quad\quad\quad\quad\quad\quad \lor \\
\quad\quad\quad\quad\quad\quad (pset.inPolSet(n) \in \mathrm{ran}\, PolSet \land \\
\quad\quad\quad\quad\quad\quad ef = evalPolicySet(x, req, ((PolSet^{\sim})(pset.inPolSet(n))))) \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ n \mapsto ef\})) \\
\quad\quad \land \\
\quad\quad matchTarget((x.getTarget(pset.target), req)) = FALSE \Rightarrow \\
\quad\quad\quad evalPolicySet(x, req, psid) = NotApplicable \\
\quad\quad \land \\
\quad\quad matchTarget((x.getTarget(pset.target), req)) = INDETERMINATE \Rightarrow \\
\quad\quad\quad evalPolicySet(x, req, psid) = Indeterminate
\end{array}
$$

5.2.7 The effect of an XACML policy

The *evalXACML* function returns the effect of the application of a request to an instance of *XACML*. If no member of the set *rootPol* matches the target then *NotApplicable* is returned. If multiple members of *rootPol* match the target then the behaviour is not defined and *Indeterminate* is returned. If a single member matches then the request is applied to that policy or policy set and the result of that evaluation is returned.

$$
\begin{array}{|l}
evalXACML : XACML \times Request \rightarrow Effect \\
\hline
\forall\, x : XACML;\ req : Request;\ prefid : \mathbb{P}\, PolicyRef \bullet \\
\quad prefid = \{prid : x.rootPol \mid \\
\qquad (prid \in \text{ran } Pol \wedge \\
\qquad\quad matchTarget((x.getTarget((x.getPolicy((Pol^{\sim})prid)).target)), req) = TRUE) \vee \\
\qquad (prid \in \text{ran } PolSet \wedge \\
\qquad\quad matchTarget((x.getTarget((x.getPolicySet((PolSet^{\sim})prid)).target)), req) = TRUE)\} \wedge \\
\quad \#prefid = 0 \Rightarrow evalXACML(x, req) = NotApplicable \wedge \\
\quad \#prefid > 1 \Rightarrow evalXACML(x, req) = Indeterminate \wedge \\
\quad \#prefid = 1 \Rightarrow (\forall\, p : prefid \bullet \\
\qquad (p \in \text{ran } Pol \Rightarrow evalXACML(x, req) = evalPolicy(x, req, (Pol^{\sim})p)) \wedge \\
\qquad (p \in \text{ran } PolSet \Rightarrow evalXACML(x, req) = evalPolicySet(x, req, (PolSet^{\sim})p)))
\end{array}
$$

5.3 An example

This section uses the same scenario as was defined in Section 4.1.1 and demonstrates how the formal model can be used to capture the described permissions. As well as demonstrating the defining of policies, sample requests are also shown. Alongside the formal representation, fragments of XACML are included for illustration; the full XACML representation of the policies can be found in Appendix A.1.

It is first necessary to define *Austin*, *Morris*, *Rover* and *Triumph* to be members of *Subject*, *read* and *write* to be elements of *Action*, and finally *prescribeDB* to be a *Resource*.

$$
\begin{array}{|l}
Austin : Subject \\
Morris : Subject \\
Rover : Subject \\
Triumph : Subject \\
read : Action \\
write : Action \\
prescribeDB : Resource
\end{array}
$$

It is then necessary to declare a number of functions. These functions will form the component parts of the *Target* schemas. They will take the request components as input and return an *EvalRes*. For instance the first function declared below is used to match a resource. It returns *TRUE* if the resource being requested is the *prescribeDB*, and *FALSE* otherwise.

$$
\begin{array}{|l}
matchPrescribeDB : Resource \rightarrow EvalRes \\
\hline
\forall\, r : Resource \bullet r = prescribeDB \Rightarrow matchPrescribeDB\ r = TRUE \\
\forall\, r : Resource \bullet r \neq prescribeDB \Rightarrow matchPrescribeDB\ r = FALSE
\end{array}
$$

The function *matchPrescribeDB* defined above is equivalent to the following XACML fragment. This fragment forms part of a *Target* section within the full XACML policy set.

```
<Resources>
    <Resource>
        <ResourceMatch
```

```
        MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">prescribeDB
          </AttributeValue>
          <ResourceAttributeDesignator
            AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </ResourceMatch>
    </Resource>
</Resources>
```

The remainder of the functions that are used for the various *Subject* and *Action* matches that need to be performed are defined in a similar fashion. The functions defined perform the following matches against subjects and actions. The function *matchNurse* matches the subject, passed as a parameter, against either*Austin* or *Triumph*, and *matchDoctor* matches against *Morris* or *Rover*. The *matchRead* function matches the action passed as a parameter against *read* and *matchWrite* matches against *write*.

$$matchNurse : Subject \rightarrow EvalRes$$

$$\forall r : Subject \bullet r = Austin \lor r = Triumph \Rightarrow matchNurse\, r = TRUE$$
$$\forall r : Subject \bullet r \neq Austin \land r \neq Triumph \Rightarrow matchNurse\, r = FALSE$$

$$matchDoctor : Subject \rightarrow EvalRes$$

$$\forall r : Subject \bullet r = Morris \lor r = Rover \Rightarrow matchDoctor\, r = TRUE$$
$$\forall r : Subject \bullet r \neq Morris \land r \neq Rover \Rightarrow matchDoctor\, r = FALSE$$

$$matchRead : Action \rightarrow EvalRes$$

$$\forall r : Action \bullet r = read \Rightarrow matchRead\, r = TRUE$$
$$\forall r : Action \bullet r \neq read \Rightarrow matchRead\, r = FALSE$$

$$matchWrite : Action \rightarrow EvalRes$$

$$\forall r : Action \bullet r = write \Rightarrow matchWrite\, r = TRUE$$
$$\forall r : Action \bullet r \neq write \Rightarrow matchWrite\, r = FALSE$$

It is necessary to define an instance of XACML which will represent the overall policy. *ValidXACML* has been chosen for this as it places the additional healthiness constraints on the policy.

$$xacml : ValidXACML$$

With the above functions defined, it is now possible to define a number of targets that will be used by policy sets, policies or rules. As stated in the description of XACML, the target section is the part used to select applicable policy sets, policies or rules. The first definition below describes a target that is applicable if there is a successful match of the subject against either a doctor or

nurse as well as a successful match of the resource against *prescribeDB*.

$$
\begin{array}{|l}
T\_PrescribeDB : TargetID \\
\hline
(xacml.getTarget(T\_PrescribeDB)).tid = T\_PrescribeDB \\
(xacml.getTarget(T\_PrescribeDB)).sub = \langle matchDoctor, matchNurse \rangle \\
(xacml.getTarget(T\_PrescribeDB)).act = \langle\rangle \\
(xacml.getTarget(T\_PrescribeDB)).res = \langle matchPrescribeDB \rangle \\
(xacml.getTarget(T\_PrescribeDB)).env = \langle\rangle
\end{array}
$$

The following fragment of XACML, which shows a target block, is equivalent to $T\_PrescribeDB$, shown above.

```
<Target>
  <Subjects>
    <Subject>
      <SubjectMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">Morris
        </AttributeValue>
        <SubjectAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
    </Subject>
    <Subject>
      <SubjectMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">Rover
        </AttributeValue>
        <SubjectAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
    </Subject>
    <Subject>
      <SubjectMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">Austin
        </AttributeValue>
        <SubjectAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
    </Subject>
    <Subject>
      <SubjectMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">Triumph
        </AttributeValue>
        <SubjectAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
```

```
  <Resources>
    <Resource>
      <ResourceMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">prescribeDB
        </AttributeValue>
        <ResourceAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </ResourceMatch>
    </Resource>
  </Resources>
</Target>
```

The remaining required targets are defined in a similar way. The targets defined as T_Doctor and T_Nurse match subjects who are either doctors or nurses respectively. In a similar fashion the targets T_Read and T_Write match the actions of *read* and *write* respectively.

$$
\begin{array}{|l}
T\_Doctor : TargetID \\
\hline
(xacml.getTarget(T\_Doctor)).tid = T\_Doctor \\
(xacml.getTarget(T\_Doctor)).sub = \langle matchDoctor \rangle \\
(xacml.getTarget(T\_Doctor)).act = \langle\rangle \\
(xacml.getTarget(T\_Doctor)).res = \langle\rangle \\
(xacml.getTarget(T\_Doctor)).env = \langle\rangle
\end{array}
$$

$$
\begin{array}{|l}
T\_Nurse : TargetID \\
\hline
(xacml.getTarget(T\_Nurse)).tid = T\_Nurse \\
(xacml.getTarget(T\_Nurse)).sub = \langle matchNurse \rangle \\
(xacml.getTarget(T\_Nurse)).act = \langle\rangle \\
(xacml.getTarget(T\_Nurse)).res = \langle\rangle \\
(xacml.getTarget(T\_Nurse)).env = \langle\rangle
\end{array}
$$

$$
\begin{array}{|l}
T\_Read : TargetID \\
\hline
(xacml.getTarget(T\_Read)).tid = T\_Read \\
(xacml.getTarget(T\_Read)).sub = \langle\rangle \\
(xacml.getTarget(T\_Read)).act = \langle matchRead \rangle \\
(xacml.getTarget(T\_Read)).res = \langle\rangle \\
(xacml.getTarget(T\_Read)).env = \langle\rangle
\end{array}
$$

$$
\begin{array}{|l}
T\_Write : TargetID \\
\hline
(xacml.getTarget(T\_Write)).tid = T\_Write \\
(xacml.getTarget(T\_Write)).sub = \langle\rangle \\
(xacml.getTarget(T\_Write)).act = \langle matchWrite \rangle \\
(xacml.getTarget(T\_Write)).res = \langle\rangle \\
(xacml.getTarget(T\_Write)).env = \langle\rangle
\end{array}
$$

It is now possible to consider the rules that are required. The first rule returns a *Permit* if the request contains the action *read* — this is matched to the target *T_Read*. The empty condition always evaluates to true as defined in *evalCondition*. The subject and resource parts of the target will effectively be inherited from the policy and policy set that the rule appears in.

$$
\begin{array}{|l}
R\_ReadPres : RuleID \\
\hline
(xacml.getRule(R\_ReadPres)).rid = R\_ReadPres \\
(xacml.getRule(R\_ReadPres)).target = T\_Read \\
(xacml.getRule(R\_ReadPres)).effect = Permit \\
(xacml.getRule(R\_ReadPres)).condition = \langle\rangle
\end{array}
$$

In a similar way, a rule to apply on a write prescription request can be defined.

$$
\begin{array}{|l}
R\_WritePres : RuleID \\
\hline
(xacml.getRule(R\_WritePres)).rid = R\_WritePres \\
(xacml.getRule(R\_WritePres)).target = T\_Write \\
(xacml.getRule(R\_WritePres)).effect = Permit \\
(xacml.getRule(R\_WritePres)).condition = \langle\rangle
\end{array}
$$

Now that the targets and basic rules are defined, it is possible to define policies, with the first being active only if a subject who is a doctor (in this example, either *Morris* or *Rover*), is present in the request to match the target. If this is the case, the rules *R_ReadPres* and *R_WritePres* are evaluated. The results of the rule evaluation are then combined using the permit override rule-combining algorithm.

$$
\begin{array}{|l}
P\_Dr : PolicyID \\
\hline
(xacml.getPolicy(P\_Dr)).pid = P\_Dr \\
(xacml.getPolicy(P\_Dr)).target = T\_Doctor \\
(xacml.getPolicy(P\_Dr)).rca = rulPermitOverride \\
(xacml.getPolicy(P\_Dr)).inPol = \langle R\_ReadPres, R\_WritePres\rangle \\
(xacml.getPolicy(P\_Dr)).obli = \emptyset
\end{array}
$$

In a similar way a policy which is evaluated in the circumstance of the subject being a nurse (either *Austin* or *Triumph*) can be defined.

$$
\begin{array}{|l}
P\_Nurse : PolicyID \\
\hline
(xacml.getPolicy(P\_Nurse)).pid = P\_Nurse \\
(xacml.getPolicy(P\_Nurse)).target = T\_Nurse \\
(xacml.getPolicy(P\_Nurse)).rca = rulPermitOverride \\
(xacml.getPolicy(P\_Nurse)).inPol = \langle R\_ReadPres\rangle \\
(xacml.getPolicy(P\_Nurse)).obli = \emptyset
\end{array}
$$

The full policy set can then be constructed using the above components, with the target of the policy set being *T_PrescribeDB*. If the target is matched, the policy-combining algorithm

polPermitOverride is used to combine the results of the policies *p_Dr* and *P_Nurse*.

$$
\begin{array}{|l}
PS\_PrescribeDB : PolicySetID \\
\hline
(xacml.getPolicySet(PS\_PrescribeDB)).psid = PS\_PrescribeDB \\
(xacml.getPolicySet(PS\_PrescribeDB)).target = T\_PrescribeDB \\
(xacml.getPolicySet(PS\_PrescribeDB)).pca = polPermitOverride \\
(xacml.getPolicySet(PS\_PrescribeDB)).inPolSet = \langle Pol(P\_Dr), Pol(P\_Nurse)\rangle \\
(xacml.getPolicySet(PS\_PrescribeDB)).obli = \emptyset
\end{array}
$$

The following is a possible request that could be evaluated against the policy set defined above — with the request being from *drX* to write a prescription using the resource *prescribeDB*. To do this, we first need to define that *Rq_MorrisWrite* is an element of *RequestID*.

$$
\begin{array}{|l}
Rq\_MorrisWrite : RequestID
\end{array}
$$

We can then define the request as follows.

$$
\begin{array}{|l}
morrisWrite : Request \\
\hline
morrisWrite.request = Rq\_MorrisWrite \\
morrisWrite.sub = \{Morris\} \\
morrisWrite.act = \{write\} \\
morrisWrite.res = \{prescribeDB\}
\end{array}
$$

A full XACML representation of the above request can be found in Appendix A.3 — the request is asking if *Morris* is allowed to *write* to the database *prescribeDB*.

Evaluating the request against the policy will result in a *permit*. The policy set target is matched against the subject *drX* and resource *prescribeDB*. With this being a match, the policy targets are then checked and the policy matching *drX* is selected. Each of the rules in the policy are checked and the rule with the target action of *writePrescription* is evaluated. The *Permit* that results is propagated and returned as the result.

$$
\begin{array}{|l}
res : Effect \\
\hline
res = evalPolicySet(xacml, morrisWrite, PS\_PrescribeDB)
\end{array}
$$

In Appendix A.1, the policy is represented in terms of XACML. XACML provides us with the benefits of expressibility, but this is coupled with verbosity. Our Z representation of the same policy is given in Appendix A.2: here we also have expressibility — but with comprehensibility. Similarly, an XACML representation of a request is given in Appendix A.3, with our Z equivalent being given in Appendix A.4.

5.4 An Alloy model of XACML

This section presents an Alloy model of XACML based on the Z model of XACML presented above. The module `accesscontrol/xacml` is the XACML model in Alloy, which also utilises the

definitions of `accesscontrol/types`, presented in Chapter 4. The model is presented by defining a target and how a target is tested for applicability against a request. Next, the core components of XACML are defined, followed by a definition of XACML and functions to evaluate a request against an instance of XACML.

5.4.1 Alloy definition of Target

First a signature `Target` is defined as a set of `Users`, `Actions` and `Resources`. `Target` is used to decide the applicability of an XACML component in relation to a request.

```
module accesscontrol/xacml
open accesscontrol/types

sig Target {
    sub : set User,
    act : set Action,
    res : set Resource
}
```

`EmptyTarget` is then defined as an extension of `Target`; this will match any request and has all the constituent parts set to none. In addition, the fact `uniqueTarget` constrains each target to be unique.

```
one sig EmptyTarget extends Target {}
{
    sub = none
    act = none
    res = none
}

fact uniqueTarget {
    all disj t1, t2 : Target |
       t1.sub != t2.sub || t1.act != t2.act || t1.res != t2.res
}
```

The predicate `matchTarget` is defined to evaluate if a target is applicable to a request: if the target is the empty target or every element of the request is found within a non-empty element of the target, then the target is applicable to the request.

```
pred matchTarget (t : Target, req : Request) {
    (req.u in t.sub  || t.sub == none) &&
    (req.p.action in t.act || t.act  == none) &&
    (req.p.resource in t.res || t.res  == none)
}
```

5.5 Alloy definition of XACML components

It is now possible to define the main components of XACML, which are `Rule`, `Policy` and `PolicySet`. These are defined as extensions to a couple of base signatures, which contain the common elements. The base signature underpinning all three is the abstract signature `XACMLelementBase`, which contains an element relating to the target and defines the relation `eval` between elements of `Request` and elements of `EvalRes`. It is worth noting that it is necessary to define `eval` within the base type to overcome the restriction in Alloy that it does not permit recursive predicate or function definitions. This base type is extended into a `Rule` by adding an element `eff` which represents the effect of the rule (constrained to be either `Permit` or `Deny`), along with a second constraint which defines the behaviour `eval`. Although the XACML standard also includes a Condition in the definition of Rule, we have chosen ignore this aspect as it defaults to true if omitted; furthermore, we do not require a condition when translating from RBAC.

```
abstract sig XACMLelementBase {
    t : Target,
    eval : Request -> one EvalRes
}

sig Rule extends XACMLelementBase { eff : EvalRes }
{
    eff = Permit || eff = Deny
    all r : Request |
        matchTarget[t, r] => eval[r] = eff else eval[r] = NotApplicable
}
```

Another abstract signature, `PolBase`, which extends `XACMLelementBase` is defined. `PolBase` is used as a base for `Policy` and `PolicySet`.

`Policy` extends `PolBase` by adding a constraint to `eval`, along with an element `inPol`, which is the set of rule contained within the policy. In this case the constraint `eval` models the rule-combining algorithm `rulePermitOverride`.

```
abstract sig PolBase extends XACMLelementBase {}

sig Policy  extends PolBase { inPol : set Rule }
{
    all r : Request |
        matchTarget[t,r] =>
            (
               Permit in (inPol.@eval)[r]  => eval[r] = Permit
               else
               Deny in (inPol.@eval)[r] => eval[r] = Deny
               else
               eval[r] = NotApplicable
```

```
            )
        else eval[r] =  NotApplicable
}
```

In a similar fashion, `PolicySet` extends `PolBase` by adding a constraint on `eval` and an element `inPolSet` which is a set of `PolBase`. In a similar way to above, the constraint on `eval` models the policy-combining algorithm

```
sig PolicySet extends PolBase { inPolSet :  set PolBase }
{
    this !in inPolSet
    all r : Request |
        matchTarget[t,r] =>
            (
               Permit in (inPolSet.@eval)[r]  => eval[r] = Permit
               else
               Deny in (inPolSet.@eval)[r] => eval[r] = Deny
               else
               eval[r] = NotApplicable
            )
        else eval[r] =  NotApplicable
}
```

5.5.1 XACML in Alloy

Next, the signature `XACML` is defined, which contains an element of type `PolBase`, which is the root policy or policyset of the particular XACML instance.

```
sig XACML { rootPolSet : PolBase }
```

Finally, a function `evalXACML` is defined to evaluate the result of appluing a request to an instance of XACML.

```
fun evalXACML(x: XACML , req : Request) : EvalRes {
    x.rootPolSet.eval[req]
}
```

5.5.2 Comments on the Alloy model

It is worth noting some of the differences between the Z specification and the Alloy specification. It is not possible to use recursion within an Alloy specification; as the evaluation of a request recursively traverses the policies and policies sets to evaluate the request, a different approach was necessary for the Alloy representation of XACML. This resulted in the addition of `eval` to the `XACMLelementBase` signature. The meaning of `eval` is defined in different ways in `Rule`,

`Policy` and `PolicySet`. The model presented only defines the `PermitOverride` combining algorithm, with this combining algorithm being chosen as it is the one utilised by the translation process described in Chapter 6. Of course, it would be possible to extend the model to capture `DenyOverride` and `FirstApplicable`. However, the implementation of `FirstApplicable` requires the use of sequences which are not handled efficiently by the Alloy Analyzer.

5.6 Example using XACML in Alloy

This section presents a possible policy that conforms to the running example using the Alloy representation of XACML. In the same way as for the RBAC representation of Chapter 4, the base types `User`, `Action` and `Resource` are extended to reflect the scenario under consideration.

```
module accesscontrol/xacmlexampletest

open accesscontrol/xacml

one sig Morris, Austin, Rover, Triumph extends User {}
one sig read, write extends Action {}
one sig PrescribeDB extends Resource {}
```

Two extensions to `PRMSBase` are then needed to represent the permissions `writePrescription` and `readPrescription`, which are represented by the action-resource pairs of (`write, prescribeDB`) and (`read, prescribeDB`) respectively.

```
one sig writePrescription extends PRMSBase {}
{
    action = write
    resource = PrescribeDB
}

one sig readPrescription extends PRMSBase {}
{
    action =  read
    resource =  PrescribeDB
}
```

Next, it is necessary to define a number of targets that will be used by the rules, policies and policy sets that will make up the XACML policy.

```
one sig readPresTarget extends Target{}
{
    sub = none
```

```
    act = read
    res = PrescribeDB
}

one sig writePresTarget extends Target{}
{
    sub = none
    act = write
    res = PrescribeDB
}

one sig nurseTarget extends Target{}
{
    sub = Austin + Triumph
    act = none
    res = none
}

one sig doctorTarget extends Target{}
{
    sub = Morris + Rover
    act = none
    res = none
}
```

Two rules are defined that permit the action `write` on the resource `prescribeDB` and the action `read` on the resource `prescribeDB`.

```
one sig readPresRule extends Rule {}
{
    t = readPresTarget
    eff = Permit
}

one sig writePresRule extends Rule {}
{
    t = writePresTarget
    eff = Permit
}
```

Two policies are defined that 'wrap' the rules defined above.

```
one sig readPresPolicy extends Policy {}
{
```

```
    t = EmptyTarget
    inPol = readPresRule
}

one sig writePresPolicy extends Policy {}
{
    t = EmptyTarget
    inPol = writePresRule
}
```

Next, two policies sets are defined: one matches requests from users that are doctors — that is either Morris or Rover; the other matches if the user is a nurse — either Austin or Triumph.

```
one sig doctorPolicySet extends PolicySet {}
{
    t = doctorTarget
    inPolSet = writePresPolicy + readPresPolicy
}

one sig nursePolicySet extends PolicySet {}
{
    t = nurseTarget
    inPolSet = readPresPolicy
}
```

Finally, the two policy sets are combined into a single universally applicable policy set which is then set to be the root policy set of the XACML instance.

```
one sig masterPolicySet extends PolicySet {}
{
    t = EmptyTarget
    inPolSet = nursePolicySet + doctorPolicySet
}

one sig exXACML extends XACML {}
{
    rootPolSet = masterPolicySet
}
```

A number of requests are then defined so that the XACML instance can be tested for conformance with the scenario.

```
one sig reqMorrisWrite extends Request {}
{
   u = Morris
   p = writePrescription
```

```
}

one sig reqAustinWrite extends Request {}
{
   u = Austin
   p = writePrescription
}

one sig reqMorrisRead extends Request {}
{
   u = Morris
   p = readPrescription
}

one sig reqAustinRead extends Request {}
{
   u = Austin
   p = readPrescription
}
```

The following defines a number of tests, in the form of assertions, that use the previously defined requests. The first test `assert evalMorrisWrite` makes the assertion that the function `evalXACML` supplied with the parameters `exXACML` and the request `reqMorrisWrite` should return `Permit`. The other tests are defined in a similar way. It is worth noting that the result of `NotApplicable` is treated in the same way as `Deny` in this example. This is because, in general, it would be undesirable to permit someone to perform an action on a resource when no policies returned a result for the request.

```
module accesscontrol/xacmlexampletestex

open accesscontrol/xacmlexampletest

assert evalMorrisWrite {
    evalXACML[exXACML, reqMorrisWrite ] = Permit
}

check evalMorrisWrite

assert evalAustinWriteF {
    evalXACML[exXACML, reqAustinWrite ] = Deny  ||
        evalXACML[exXACML, reqAustinWrite ] = NotApplicable
}
```

```
check evalAustinWriteF

assert evalMorrisRead {
    evalXACML[exXACML, reqMorrisRead ] = Permit
}

check evalMorrisRead

assert evalAustinRead  {
    evalXACML[exXACML, reqAustinRead ] = Permit
}

check evalAustinRead
```

Finally, the results generated by executing the tests are given below.

```
Executing "Check evalMorrisWrite"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   3053 vars. 264 primary vars. 6642 clauses. 23ms.
   No counterexample found. Assertion may be valid. 2ms.

Executing "Check evalAustinWriteF"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   3056 vars. 264 primary vars. 6667 clauses. 26ms.
   No counterexample found. Assertion may be valid. 2ms.

Executing "Check evalMorrisRead"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   3053 vars. 264 primary vars. 6642 clauses. 32ms.
   No counterexample found. Assertion may be valid. 1ms.

Executing "Check evalAustinRead"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   3053 vars. 264 primary vars. 6642 clauses. 25ms.
   No counterexample found. Assertion may be valid. 2ms.

4 commands were executed. The results are:
   #1: No counterexample found. evalMorrisWrite may be valid.
   #2: No counterexample found. evalAustinWriteF may be valid.
   #3: No counterexample found. evalMorrisRead may be valid.
   #4: No counterexample found. evalAustinRead may be valid.
```

The fact that no counter-examples were found provides confidence in the model for at least the scope of the instance in the same way as noted in Chapter 4.

5.7 Summary

This chapter has presented two formal models of XACML. In addition, the formal models have been used to represent an access control policy based on the running example defined in Chapter 4. The first model presented in terms of Z, captures both the XACML component parts along with the combining algorithms and how a request to an XACML policy could be evaluated. The Alloy model presented is a slight simplification of the Z model. This is mainly because of an inability to define recursive functions in Alloy. However, the Alloy model has permitted the evaluation of requests and given some assurance that the basic model is sound.

The following chapter will use the formal models presented in this chapter and the previous chapter as a basis for defining a formal translation between RBAC and XACML. In addition to the use of the formal models in the translation process, the ability to represent policies in a formal way opens up the possibility of performing analysis of the policies to check that they are consistent, moving towards achieving the goals of Chapter 3.

Chapter 6

Translation of RBAC to XACML

This chapter presents formal models defining the translation from RBAC to XACML in terms of both Z and Alloy. The translation utilises both the formal model of RBAC of Chapter 4 and the formal model of XACML of Chapter 5. The first model detailing the translation process is presented in terms of Z, and is followed by a walked-through example of its application to our running example. The second model is presented in terms of Alloy and is used to validate that the technique works within the scope explored by executing tests within the Alloy Analyzer.

When considering how to model the translation between RBAC and XACML it was first important to consider the philosophy behind RBAC access control policies, with the most important aspect being that an RBAC access control system utilises a default deny access control mechanism. This essentially means that every action a user is permitted to perform on a resource has to be explicitly encoded within the RBAC access control policy. This is achieved by the user being explicitly given the roles with the permission in question, or the user being given a role which inherits, either directly or indirectly, from another role with the appropriate permission. Thus, an RBAC policy is what is termed "closed".

In contrast XACML is more complicated and allows policies to be written to give default responses from the following: *Permit*, *Deny* or *NotApplicable*. This can be achieved by using rules that return either *Permit* or *Deny* and the careful choice of rule and policy combining algorithms.

When the translation from RBAC to XACML is considered, the XACML policy should return a *Permit* for every explicitly defined permission in the RBAC policy and either *Deny* or *NotApplicable* in all other cases. The latter response has been chosen for this translation as this gives the best correspondence between the RBAC and XACML policies. To facilitate this the XACML representation of the RBAC policy requires rules that, when matched, return *Permit*, and that each rule corresponds to an RBAC permission. The rules then need to be combined into policies and policy sets which are combined with the *rulPermitOverride* and *polPermitOverride* combining algorithms respectively. Following this pattern, it is possible to define a policy that when evaluated in the presence of a request returns one of two values (in the absence of errors). If the action on the resource is permitted for the user, a *Permit* is returned; if the user did not explicitly have permission to perform the action on the resource, the XACML response is *NotApplicable*.

The following section describes the correspondence between the RBAC and XACML components, which is then used in the subsequent sections to model the translation in terms of Z and Alloy.

6.1 Translation outline

This section provides an overview of the relationship between RBAC and XACML, which will form the basis of the translation process, as well as an outline of the translation process itself. The overall translation process is shown in Figure 6.1, which illustrates the mappings between a policy captured in RBAC and an equivalent policy presented in XACML.

To support the translation process, the following correspondences between the component parts of RBAC and XACML are used. An RBAC permission, which is represented by an element of *PRMSBase*, is mapped to an XACML rule as shown on the right-hand side of Figure 6.1. Next, when roles are considered, each role may be part of several different relations that make up the overall RBAC policy. This is captured in Figure 6.1, where from right to left the figure shows that a role may be used in the *PA* relation, the *RH* relation or the *UA* relation, as well as illustrating how each of these relations are represented as XACML components. More specifically, each role that is a member of the domain of *PA* is mapped to an XACML policy. Each role that is a member of the domain of *RH* is mapped to a policy set, as is each role from the range of *UA*. The rules, policies and policy sets that are created are combined to form the overall policy, which is itself captured in an overarching root policy set. The rest of this section provides details about the contents of XACML rules, policies and policy sets, in particular with respect to how they relate to the contents of an RBAC policy.

Each *PRMSBase* element contained in an RBAC policy represents a single permission detailing an action that can be performed on a resource. The translation process will perform a one-to-one mapping between elements of *PRMSBase* and an XACML rule, with each rule capturing the permission by means of the target and having an overall effect of *Permit*. The rules target captures the permission by setting the *act* and *res* sequences to contain the action and resource of the permission it is to represent. This is illustrated in Figure 6.1 with the RBAC *PRMSBase* of $(write, prescribeDB)$ mapping to an XACML rule with an id of $f_r(write, prescribeDB)$, the target consisting of *act* being $\langle t\,(write)\rangle$, *res* being $\langle t\,(prescribeDB)\rangle$ and the *effect* being *Permit*. Here, f_r is a function that generates an identifier that can be used to reference the rule. The function t, in the sequences *act* and *res*, is a generic generator function, which, when given an element, will generate a function that returns *true* when the supplied argument has the same value as the originally supplied element, and *false* otherwise. The empty sequences corresponding to *sub* and *env* have been omitted from the diagram for clarity, as they would match any subject or environment respectively and therefore have no effect on the requests that the target would match.

The three different mappings from an RBAC role to corresponding XACML components are now explored. Each of the policies and policy sets defined combine to give an overall effect that corresponds to the original RBAC policy.

The first set of roles considered are those which appear in the domain of the *PA* relation. Each of these roles is individually mapped into an XACML policy with the following properties.

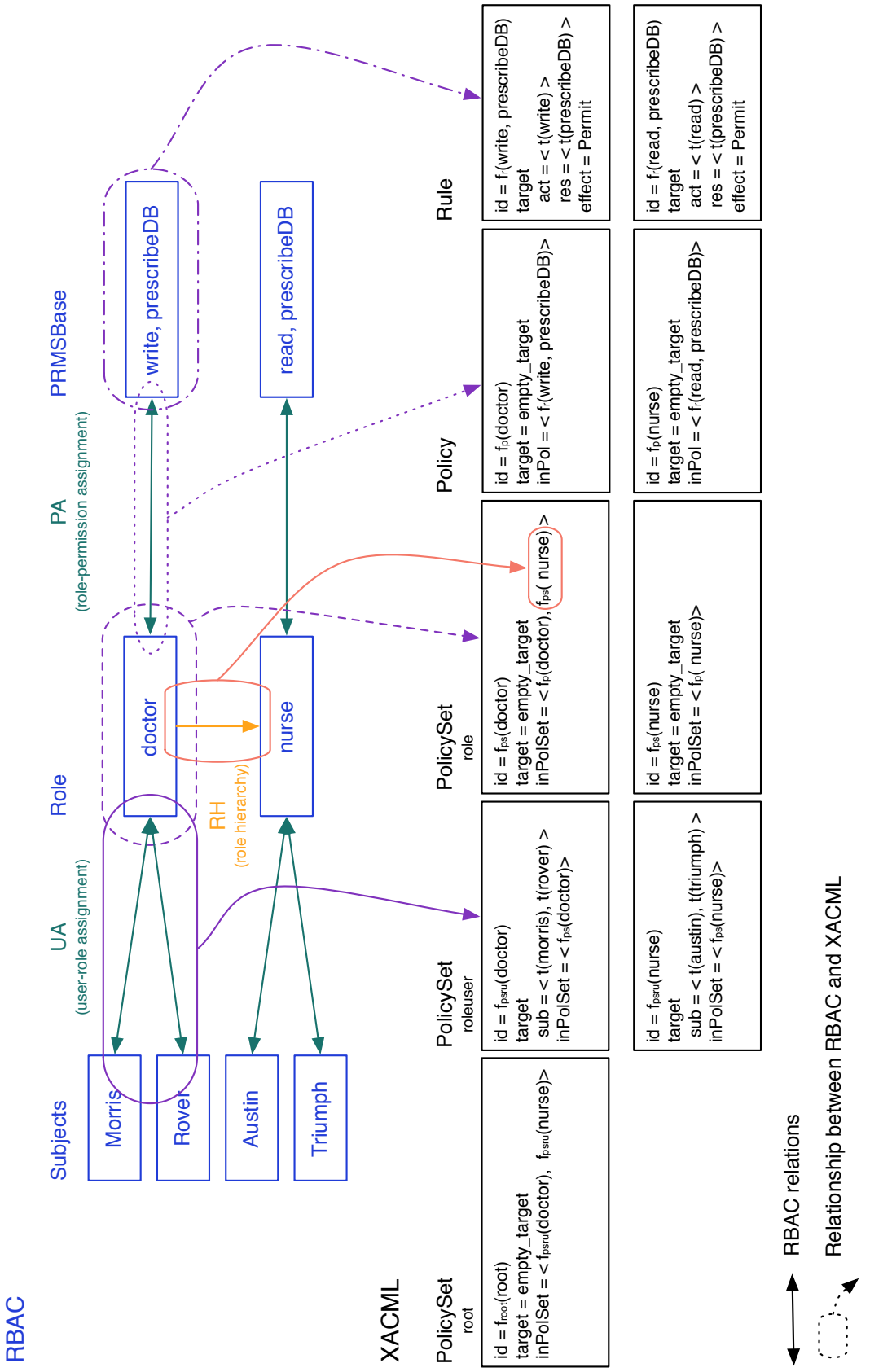

Figure 6.1: RBAC to XACML translation

The target is set to be the empty target which means the policy is applicable for any combination of user, action and resource. The value of *inPol* is set to be the sequence of rule identifiers that correspond to the permissions that are associated with the role, that is, there is an entry in *inPol* for each member of the set $PA(\!|\{role\}|\!)$. This is illustrated in Figure 6.1 by the dotted oval around the line representing the element of *PA* which associates the *doctor* role with the *PRMSBase* of $(write, prescribeDB)$ mapping to the policy with the following characteristics:

- the policies identifier, *id*, is set to be $f_p(doctor)$,
- the target is set to be the *empty_target*, and
- *inPol* is set to be the sequence $\langle f_r(write, prescribeDB)\rangle$, which is the identity of the rule that corresponds to the permission that is associated with the doctor role via the *PA* relation.

Next, the roles that make up dom RH^* are considered, which, for the purposes of translation, could be represented as ran $UA \cup$ ran $RH \cup$ dom $RH \cup$ dom *PA*. Each of these roles is represented in XACML as a policy set consisting of the following properties. The target is set to be the empty target which means the policy set is applicable for any combination of user, action and resource. The sequence that makes up *inPolSet* is made up of two distinct parts. The first part references the policy associated with the role under consideration. The second part references all the policy sets that are associated with the roles which the role under consideration inherits. This is also illustrated in Figure 6.1 by the dashed line around doctor and the line highlighting the *RH* element representing the fact that the doctor role inherits the permissions of the nurse role. The dashed line points at the overall policy set, which has the following values. The policy sets identifier, *id*, is set to be $f_{ps}(doctor)$, the target is set to be the *empty_target*, and *inPolSet* is set to contain two elements. The first element references the XACML policy which represents the RBAC role *doctor*, identified by $f_p(doctor)$. The second element references the policy set associated with the role that is directly inherited, indicated by the line from the *RH* element, which is $f_{ps}(nurse)$. This gives an overall *inPolSet* of $\langle f_p(doctor), f_{ps}(nurse)\rangle$.

The final set of roles considered are the members of ran *UA*, which will each yield a policy set for each role with the following properties. The target has *sub* set to be a sequence of users with a domain of $UA(\!|\{role\}|\!)$. The *inPolSet* will have a single entry which references the policy set for the role described above. This is illustrated in Figure 6.1 by the solid line around the elements of the *UA* relation pointing at a policy set with the following values. The identifier, *id*, is set to be $\langle f_{psru}(doctor)\rangle$. The *sub* part of target is set to be $\langle t\,(morris), t\,(rover)\rangle$, which corresponds to *t* applied to the members of $UA(\!|\{doctor\}|\!)$. Finally, *inPolSet* is set to $\langle f_{ps}(doctor)\rangle$, which references the policy set related to the role doctor. Again the target elements that are empty sequences are omitted for clarity.

Finally, a root policy set is defined which will have an empty target and an *inPolSet* that will have an entry to reference each role in ran *UA*. This is shown in Figure 6.1 as the policy set on the left-hand side with the following values. The identity, *id*, is set to be $f_{root}(root)$, the target is set to *empty_target* to make it universally applicable, and *inPolSet* is set to $\langle f_{psru}(doctor), f_{psru}(nurse)\rangle$.

6.2 Formal translation from RBAC to XACML using Z

This section presents the helper functions and translation schemas that are necessary for supporting the translation from RBAC to XACML. The helper functions provide basic utilities that are required by the translation process such as identifier generation and function definition. The schemas defined are a collection of initialisation schemas, which are utilised to produce a formal description of the translation process.

6.2.1 Helper functions

It is assumed that a number of functions exist which are capable of generating various identifiers that will be required during the translation process. The three functions that will generate an element of *TargetID* (*genTidR*, *genTidPSrr* and *genTidPSru*) are constrained to produce disjoint sets of identifiers; this is also the case for the functions that produce an element of *PolicySetID*. Each of the generator functions outlined below are both total and injective, which ensures that for every valid input there is a unique output.

$$
\begin{array}{|l}
genTidR : PRMSBase \rightarrowtail TargetID \\
genTidPSrr : Role \times \mathbb{P}\, Role \rightarrowtail TargetID \\
genTidPSru : Role \times \mathbb{P}\, User \rightarrowtail TargetID \\
genRid : PRMSBase \rightarrowtail RuleID \\
genPid : Role \rightarrowtail PolicyID \\
genPSidrr : Role \rightarrowtail PolicySetID \\
genPSidru : Role \times \mathbb{P}\, User \rightarrowtail PolicySetID \\
genPSidA : Hierarchy \rightarrowtail PolicySetID \\
\hline
\operatorname{ran} genTidR \cap \operatorname{ran} genTidPSrr = \emptyset \\
\operatorname{ran} genTidR \cap \operatorname{ran} genTidPSru = \emptyset \\
\operatorname{ran} genTidPSrr \cap \operatorname{ran} genTidPSru = \emptyset \\
\operatorname{ran} genPSidrr \cap \operatorname{ran} genPSidru = \emptyset \\
\operatorname{ran} genPSidrr \cap \operatorname{ran} genPSidA = \emptyset \\
\operatorname{ran} genPSidru \cap \operatorname{ran} genPSidA = \emptyset
\end{array}
$$

The next helper function is a generic function that converts a set to a sequence[1] ; in particular, it is used for creating the sequences that make up *inPol* and *inPolSet*.

$$
\begin{array}{|l}
[X] \\
\hline
sequence : \mathbb{P}\, X \rightarrow \operatorname{seq} X \\
\hline
\forall\, xs : \mathbb{P}\, X \bullet \#(sequence(xs)) = \#xs \land \operatorname{ran}(sequence(xs)) = xs
\end{array}
$$

[1] In Spivey [100], there is a proof obligation that a generic definition is uniquely defined for all possible instantiations. However, the Z Standard [53] relaxes this requirement, permitting the use of loose generic "definitions" like this one which actually admits a number of possible values. But had we chosen to abide by [100], then an alternative non-generic axiomatic description in the context of Subject could be:

$$
\begin{array}{|l}
sequence : \mathbb{P}\, Subject \rightarrow \operatorname{seq} Subject \\
\hline
\forall\, subs : \mathbb{P}\, Subjects \bullet \#(sequence(subs)) = \#subs \land \operatorname{ran}(sequence(subs)) = subs
\end{array}
$$

Next, the generic function *targetElementEquals* is defined, which is used to define the functions that will be used in *Target*. The function *targetElementEquals* returns a function which has the property that it returns *FALSE* for all input values — other than for the input x, for which it returns *TRUE*.

$$[X]$$

$$targetElementEqual : X \rightarrow X \rightarrow EvalRes$$

$$\forall x : X \bullet targetElementEqual\, x = \{y : X \setminus \{x\} \bullet y \mapsto FALSE\} \cup \{x \mapsto TRUE\}$$

6.2.2 Translation schemas

The translation from *Hierarchy* to *ValidXACML* is achieved via the following (initialisation) schemas.

TargetInit will generate a binding compatible with the *Target* schema. For each of the input sequences $sub?$, $act?$, $res?$ and $env?$ the resulting binding will have a sub', act', res' and env' which will contain a *targetElementEquals* function that corresponds to each element of the input sequence.

TargetInit

$$Target'$$
$$tid? : TargetID$$
$$sub? : \text{seq}\ Subject$$
$$act? : \text{seq}\ Action$$
$$res? : \text{seq}\ Resource$$
$$env? : \text{seq}\ Environment$$

$$tid' = tid?$$
$$\#sub? = \#sub'$$
$$\forall i : 1 \mathinner{..} \#sub? \bullet sub'\, i = targetElementEqual\, (sub?\, i)$$
$$\#act? = \#act'$$
$$\forall i : 1 \mathinner{..} \#act? \bullet act'\, i = targetElementEqual\, (act?\, i)$$
$$\#res? = \#res'$$
$$\forall i : 1 \mathinner{..} \#res? \bullet res'\, i = targetElementEqual\, (res?\, i)$$
$$\#env? = \#env'$$
$$\forall i : 1 \mathinner{..} \#env? \bullet env'\, i = targetElementEqual\, (env?\, i)$$

The next schema considered is *RuleInit*, which will generate a rule that has a target that matches the action and resource of the *PRMSBase*, which is the input component $prm?$, and the effect of *Permit*. The effect is *Permit* as each element of *PRMSBase* represents the permission to perform an *Action* on a *Resource*. As there are no additional conditions that have to be met, there is no conditional part to the rule which is represented by the value of *condition* being an empty sequence.

RuleInit

$Rule'$

$prm? : PRMSBase$

$rid' = genRid\ prm?$

$target' = genTidR\ prm?$

$effect' = Permit$

$condition' = \langle\rangle$

The next initialisation schema captures the mapping represented by the relation *PA* between an element of *Role* and all the permissions associated with that *Role*, captured by a set containing elements of *PRMSBase*. This is achieved by defining *RolePolicyInit*, which creates a *Policy* binding that has *inPol* which contains the identifiers of all the rules which are associated with the set of *PRMSBase* elements supplied as the input parameter *prms*?. The rule combining algorithm is set to be *rulPermitOverride* as this captures the fact that an RBAC policy essentially defines what is permitted.

RolePolicyInit

$Policy'$

$r? : Role$

$prms? : \mathbb{P}\ PRMSBase$

$pid' = genPid\ r?$

$target' = empty\_target\_id$

$\forall\ prm : prms? \bullet genRid\ prm \in \operatorname{ran} inPol'$

$\#inPol' = \#prms?$

$rca' = rulPermitOverride$

$obli' = \emptyset$

The relationship between elements of *Role*, which is captured by the *RH* relation, is considered next. The *RH* relation explicitly defines which elements of *Role* directly inherit from other elements of *Role*. The *RoleRolePolicySetInit* schema creates a *PolicySet* binding for a role with the *inPolSet* made up of a reference to the policy associated with the role (if one exists), along with a reference to each policy set associated with each role inherited. If no policy representing the role exists it indicates that the role only exists within the role hierarchy and has no explicit permissions associated with it via the *PA* relation. The set of directly inherited roles associated with the supplied role, $r?$, is calculated by $h?.RH(\!|\ \{r?\}\ |\!)$. The use of relational image on the $h.RH$ relation returns the set of roles that are explicitly associated with $r?$ in the *RH* relation of $h?$, the supplied *Hierarchy*. The permit override combining algorithm *polPermitOverride* is selected for the same reason as outlined above.

RoleRolePolicySetInit

$PolicySet'$
$h? : Hierarchy$
$r? : Role$

$psid' = genPSidrr\,(r?)$
$target' = empty\_target\_id$
$pca' = polPermitOverride$
$\forall\, ro : h?.RH(\!|\, \{r?\} \,|\!) \bullet PolSet\,(genPSidrr\,(ro)) \in \operatorname{ran} inPolSet'$
$Pol\,(genPid\,(r?)) \in \operatorname{ran} inPolSet'$
$\#inPolSet' = \#(h?.RH(\!|\, \{r?\} \,|\!)) + 1$
$obli' = \emptyset$

Now a *PolicySet* binding that associates users with the roles they are explicitly assigned can be defined via the following schema. A policy set is created for a role and the target is set to be the users who have that role directly associated with them, which is obtained from $(h?.UA^{\sim})(\!|\, \{r?\} \,|\!)$; the relation UA is inverted to obtain all the users associated with role $r?$ by the use of relational image. The value of *inPolSet* contains a single reference to the policy set that is associated with the role $r?$. Again the permit override combining algorithm *polPermitOverride* is used.

It is now possible to consider the UA relation and associate a role with all the subjects that are directly assigned that role via the UA relation. The element of *Target* referenced will capture the subjects associated with the role in the sequence *sub*.

RoleUserPolicySetInit

$PolicySet'$
$h? : Hierarchy$
$r? : Role$

$psid' = genPSidru\,(r?, (h?.UA^{\sim})(\!|\, \{r?\} \,|\!))$
$target' = genTidPSru\,(r?, (h?.UA^{\sim})(\!|\, \{r?\} \,|\!))$
$pca' = polPermitOverride$
$PolSet\,(genPSidrr(r?)) \in \operatorname{ran}\ inPolSet'$
$\#inPolSet' = 1$
$obli' = \emptyset$

It is then necessary to define a policy set initialisation schema that will represent the root policy set which collects together all of the role-user policy sets. This is achieved by the *RootPolicySetInit* schema which generates a *PolicySet* binding that has an *inPolSet* which consists of references to role-user policy sets, for each of the roles in ran $h?.UA$.

RootPolicySetInit

$PolicySet'$

$h? : Hierarchy$

$psid' = genPSidA\, h?$

$target' = empty\_target\_id$

$pca' = polPermitOverride$

$\forall ro : \operatorname{ran} h?.UA \bullet PolSet(genPSidru(ro, (h?.UA^{\sim})(\!| \{ro\} |\!))) \in \operatorname{ran} inPolSet'$

$\#inPolSet' = \#(\operatorname{ran}(h?.UA))$

$obli' = \emptyset$

Finally, it is possible to define *XACMLfromRBACInit*, which pulls together all the previously defined schemas and initialises them with appropriate values from the hierarchy supplied to give a *ValidXACML* binding.

XACMLfromRBACInit

$ValidXACML'$

$h? : Hierarchy$

$\vdots$

The predicate part of *XACMLfromRBACInit* consists of the following seven constraints.

1. The first part states that a policy set exists which is the binding created by *RootPolicySetInit*, and the mapping from $psid'$ to that policy set binding is a member of the function *getPolicySet*:

$$\exists\, PolicySet' \bullet RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

2. The second predicate makes a statement about every role which is in the range of the relation *UA* in the hierarchy h under consideration. For each of these roles there is a policy set binding which is the result of *RoleUserPolicySetInit* as well as a mapping from the $psid'$ to the binding being a member of the function *getPolicySet*:

$$\forall r : \operatorname{ran}(h?.UA) \bullet \\ \exists\, PolicySet' \bullet RoleUserPolicySetInit\,[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

3. The third predicate ensures that there is a policy set binding associated with every role from the hierarchy under consideration. The policy set binding is initialised via *RoleRolePolicySetInit* and *getPolicySet* includes the mapping from $psid'$ to the binding:

$$\forall r : h?.ROLES \bullet \\ \exists\, PolicySet' \bullet RoleRolePolicySetInit\,[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

4. The fourth predicate ensures that there is a policy binding for every role that is either associated with a user via the *UA* relation, a permission via the *PA* relation or that appears

on either side of the RH relation. Each of the roles is associated with a policy binding initialised by $RolePolicyInit$, with $prms?$ being the set of permissions associated with the role via the PA relation. The mapping from the pid' to the policy binding becomes a member of the function $getPolicy$:

$$\begin{array}{l} \forall\, r : \bigcup\{\mathrm{ran}\,(h?.UA), \mathrm{ran}(h?.RH), \mathrm{dom}\,(h?.RH), \mathrm{dom}(h?.PA)\} \bullet \\ \quad \exists\, Policy';\ prms : \mathbb{P}\, PRMSBase \bullet \\ \qquad prms = h?.PA(\!|\ \{r\}\ |\!) \land \\ \qquad RolePolicyInit[r/r?, prms/prms?] \land \\ \qquad pid' \mapsto \theta Policy' \in getPolicy' \end{array}$$

5. The fifth predicate associates every permission from the range of the PA relation with a rule binding via $RuleInit$. In addition, the mapping from the identifier, rid, of the binding to the binding itself is a member of the function $getRule$.

$$\begin{array}{l} \forall\, prm : \mathrm{ran}(h?.PA) \bullet \\ \quad \exists\, Rule' \bullet RuleInit\,[prm/prm?] \land rid' \mapsto \theta Rule' \in getRule' \end{array}$$

6. The sixth predicate relates to the target bindings that are referenced by the policy set bindings associated with second predicate. These target bindings are designed to match a group of users to control the applicability of the policy sets. A target binding exists for each role, r, in the range of the relation UA. The identifier of the target binding, tid, is constrained to be the result of the function $genTidPSru$ with the parameters r and $(UA^{\sim})(\!|\ \{r\}\ |\!)$, where the first parameter is the role and the second parameter is the set of users associated with the role via the UA relation. The target being designed to match based on users means that the values of act, res and env are constrained to be the empty sequence, and therefore not considered when matching the target with a request. Finally, sub is constrained to be the sequence containing $(UA^{\sim})(\!|\ \{r\}\ |\!)$. This will mean the target binding will match a request which has a subject who is a member of $(UA^{\sim})(\!|\ \{r\}\ |\!)$:

$$\begin{array}{l} \forall\, r : \mathrm{ran}(h?.UA) \bullet \\ \quad \exists\, Target';\ tid : TargetID;\ sub : \mathrm{seq}\, Subject;\ act : \mathrm{seq}\, Action; \\ \qquad res : \mathrm{seq}\, Resource;\ env : \mathrm{seq}\, Environment \bullet \\ \qquad\quad tid = genTidPSru(r, (h?.UA^{\sim})(\!|\ \{r\}\ |\!)) \land \\ \qquad\quad sub = sequence((h?.UA^{\sim})(\!|\ \{r\}\ |\!)) \land \\ \qquad\quad act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\ \qquad\quad TargetInit\,[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\ \qquad\quad tid' \mapsto \theta Target' \in getTarget' \end{array}$$

7. The final predicate relates to the targets that are associated with the rules from the fifth predicate. A target binding is associated with every permission from the range of the PA relation. Each of these target bindings is initialised by $TargetInit$ and conforms to the following constraints. The identifier, tid, is the value given by the function $genTidR$ with the parameter of prm. The values of sub and env are both constrained to be empty

sequences. In addition, to facilitate the matching of a request involving the permission, *prm*, the value of *act* is constrained to be the sequence containing the action part of *prm* and *res* is constrained to be the sequence containing the resource part of *prm*.

$$
\begin{array}{l}
\forall\, prm : \mathrm{ran}(h?.PA) \bullet \\
\quad \exists\, Target'; \; tid : TargetID; \; sub : \mathrm{seq}\, Subject; \; act : \mathrm{seq}\, Action; \\
\quad\quad res : \mathrm{seq}\, Resource; \; env : \mathrm{seq}\, Environment \bullet \\
\quad\quad\quad tid = genTidR\, prm \land \\
\quad\quad\quad sub = \langle\rangle \land \\
\quad\quad\quad act = \langle first\, prm \rangle \land \\
\quad\quad\quad res = \langle second\, prm \rangle \land \\
\quad\quad\quad env = \langle\rangle \land \\
\quad\quad\quad TargetInit\, [tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\quad\quad\quad tid' \mapsto \theta\, Target' \in getTarget'
\end{array}
$$

The entirety of the *XACMLfromRBACInit* schema is shown below.

XACMLfromRBACInit

$ValidXACML'$

$h? : Hierarchy$

$\exists\, PolicySet' \bullet RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet'$

$\forall\, r : \mathrm{ran}(h?.UA) \bullet$
$\quad \exists\, PolicySet' \bullet RoleUserPolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$

$\forall\, r : h?.ROLES \bullet$
$\quad \exists\, PolicySet' \bullet RoleRolePolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$

$\forall\, r : \bigcup\{\mathrm{ran}\,(h?.UA), \mathrm{ran}(h?.RH), \mathrm{dom}\,(h?.RH), \mathrm{dom}(h?.PA)\} \bullet$
$\quad \exists\, Policy';\ prms : \mathbb{P}\, PRMSBase \bullet$
$\quad\quad prms = h?.PA(\!|\ \{r\}\ |\!) \land$
$\quad\quad RolePolicyInit[r/r?, prms/prms?] \land$
$\quad\quad pid' \mapsto \theta Policy' \in getPolicy'$

$\forall\, prm : \mathrm{ran}(h?.PA) \bullet \exists\, Rule' \bullet RuleInit[prm/prm?] \land rid' \mapsto \theta Rule' \in getRule'$

$\forall\, r : \mathrm{ran}(h?.UA) \bullet$
$\quad \exists\, Target';\ tid : TargetID;\ sub : \mathrm{seq}\, Subject;\ act : \mathrm{seq}\, Action;$
$\quad\quad res : \mathrm{seq}\, Resource;\ env : \mathrm{seq}\, Environment \bullet$
$\quad\quad\quad tid = genTidPSru(r, (h?.UA^{\sim})(\!|\ \{r\}\ |\!)) \land$
$\quad\quad\quad sub = sequence((h?.UA^{\sim})(\!|\ \{r\}\ |\!)) \land$
$\quad\quad\quad act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land$
$\quad\quad\quad TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land$
$\quad\quad\quad tid' \mapsto \theta Target' \in getTarget'$

$\forall\, prm : \mathrm{ran}(h?.PA) \bullet$
$\quad \exists\, Target';\ tid : TargetID;\ sub : \mathrm{seq}\, Subject;\ act : \mathrm{seq}\, Action;$
$\quad\quad res : \mathrm{seq}\, Resource;\ env : \mathrm{seq}\, Environment \bullet$
$\quad\quad\quad tid = genTidR\, prm \land sub = \langle\rangle \land act = \langle first\, prm\rangle \land res = \langle second\, prm\rangle \land$
$\quad\quad\quad env = \langle\rangle \land TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land$
$\quad\quad\quad tid' \mapsto \theta Target' \in getTarget'$

6.3 A walked-through example translation

This section provides a walk-through of the translation of the previous section being applied to the example hierarchy of our running example. The application of *XACMLfromRBACInit* will translate the hierarchy into an equivalent XACML representation.

For convenience, the definitions of the users, actions, resources and roles are repeated here, along with the hierarchy *rbach*.

$Austin, Morris, Rover, Triumph : User$
$read, write : Action$
$Doctor, Nurse : Role$
$prescribeDB : Resource$

$$
\begin{array}{|l}
rbach : Hierarchy \\
\hline
rbach.UA = \{(Austin, Nurse), (Morris, Doctor), (Rover, Doctor), (Triumph, Nurse)\} \\
rbach.PA = \{Nurse \mapsto (read, prescribeDB), Doctor \mapsto (write, prescribeDB)\} \\
rbach.RH = \{(Doctor, Nurse)\} \\
rbach.USERS = \{Austin, Morris, Rover, Triumph\} \\
rbach.ROLES = \{Doctor, Nurse\} \\
rbach.PRMS = \{(read, prescribeDB), (write, prescribeDB)\}
\end{array}
$$

As a starting point, *ANS* is defined as the set of *ValidXACML* bindings that result from the evaluation of the initialisation schema *XACMLfromRBACInit* with the input of $h?$ being set to the value of example hierarchy *rbach*.

$$ANS \mathrel{\widehat=} [XACMLfromRBACInit \mid h? = rbach] \setminus (h?)$$

XACMLfromRBACInit can then be expanded and have the one-point rule (see Chapter 4 of [109]) applied to $h?$ to give the following result:

$$
\begin{array}{|l}
\hline
ValidXACML' \\
\hline
\exists\, PolicySet' \bullet RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet' \\
\\
\forall\, r : \mathrm{ran}(rbach.UA) \bullet \\
\quad \exists\, PolicySet' \bullet RoleUserPolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet' \\
\\
\forall\, r : rbach.ROLES \bullet \\
\quad \exists\, PolicySet' \bullet RoleRolePolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet' \\
\\
\forall\, r : \mathrm{ran}(rbach.UA) \cup \mathrm{ran}(rbach.RH) \cup \mathrm{dom}(rbach.RH) \cup \mathrm{dom}(rbach.PA) \bullet \\
\quad \exists\, Policy';\ prms : \mathbb{P}\, PRMSBase \bullet \\
\qquad prms = rbach.PA(\!|\ \{r\}\ |\!) \land \\
\qquad RolePolicyInit[r/r?, prms/prms?] \land \\
\qquad pid' \mapsto \theta Policy' \in getPolicy' \\
\\
\forall\, prm : \mathrm{ran}(rbach.PA) \bullet \\
\quad \exists\, Rule' \bullet RuleInit[prm/prm?] \land rid' \mapsto \theta Rule' \in getRule' \\
\\
\forall\, r : \mathrm{ran}(rbach.UA) \bullet \\
\quad \exists\, Target';\ tid : TargetID;\ sub : \mathrm{seq}\, Subject;\ act : \mathrm{seq}\, Action; \\
\qquad res : \mathrm{seq}\, Resource;\ env : \mathrm{seq}\, Environment \bullet \\
\qquad\quad tid = genTidPSru(r, (rbach.UA^{\sim})(\!|\ \{r\}\ |\!)) \land \\
\qquad\quad sub = sequence((rbach.UA^{\sim})(\!|\ \{r\}\ |\!)) \land act = \langle\rangle \land res = \langle\rangle \land \\
\qquad\quad env = \langle\rangle \land tid' \mapsto \theta Target' \in getTarget' \land \\
\qquad\quad TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \\
\\
\forall\, prm : \mathrm{ran}(rbach.PA) \bullet \\
\quad \exists\, Target';\ tid : TargetID;\ sub : \mathrm{seq}\, Subject;\ act : \mathrm{seq}\, Action; \\
\qquad res : \mathrm{seq}\, Resource;\ env : \mathrm{seq}\, Environment \bullet \\
\qquad\quad tid = genTidR\ prm \land sub = \langle\rangle \land \\
\qquad\quad act = \langle first\ prm\rangle \land res = \langle second\ prm\rangle \land env = \langle\rangle \land \\
\qquad\quad TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\quad tid' \mapsto \theta Target' \in getTarget' \\
\hline
\end{array}
$$

The first part of the above schema is now considered in detail, namely

$$\exists\, PolicySet' \bullet RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

By expanding $PolicySet'$ this becomes:

$$
\begin{array}{l}
\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\, PolicyRef;\ obli' : \mathbb{P}\, Obligation \bullet \\
\qquad RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet'
\end{array}
$$

When $RootPolicySetInit$ is expanded this becomes:

$$
\begin{array}{l}
\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\, PolicyRef;\ obli' : \mathbb{P}\, Obligation \bullet \\
\qquad psid' = genPsidA\,(rbach) \land \\
\qquad target' = empty\_target\_id \land \\
\qquad pca' = polPermitOverride \land \\
\qquad \#inPolSet' = \#\,\mathrm{ran}(rbach.UA) \land \\
\qquad (\forall\, ro : \mathrm{ran}(rbach.UA) \bullet \\
\qquad\quad PolSet(genPSidu(ro, (rbach.UA^{\sim})(\!|\, \{ro\} \,|\!)) \in \mathrm{ran}\, inPolSet') \land \\
\qquad obli' = \emptyset \land \\
\qquad psid' \mapsto \theta PolicySet' \in getPolicySet'
\end{array}
$$

The function application $genPsidA\,(rbach)$ will be left without being evaluated to a concrete value, as it is the application of a total injective function which will give a single value for a given input. Although the exact value that results from the function application is unimportant, the fact that it is a total injective function makes it useful for creating unique identifiers based on inputs provided. In a similar way, each of the other identifier functions will be left as function applications.

It is now possible to evaluate the above to get $\mathrm{ran}(rbach.UA) = \{Doctor, Nurse\}$, which, in turn, results in $\#\,\mathrm{ran}(rbach.UA) = 2$, which can then be substitute into the above to give:

$$
\begin{array}{l}
\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\, PolicyRef;\ obli' : \mathbb{P}\, Obligation \bullet \\
\qquad psid' = genPsidA\,(rbach) \land \\
\qquad target' = empty\_target\_id \land \\
\qquad pca' = polPermitOverride \land \#inPolSet' = 2 \land \\
\qquad (\forall\, ro : \{Doctor, Nurse\} \bullet \\
\qquad\quad PolSet(genPSidu(ro, (rbach.UA^{\sim})(\!|\, \{ro\} \,|\!)) \in \mathrm{ran}\, inPolSet') \land \\
\qquad obli' = \emptyset \land \\
\qquad psid' \mapsto \theta PolicySet' \in getPolicySet'
\end{array}
$$

The $\forall$ statement can then be expanded to give:

$$
\begin{array}{l}
\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\, PolicyRef;\ obli' : \mathbb{P}\, Obligation \bullet \\
\qquad psid' = genPsidA\,(rbach) \land \\
\qquad target' = empty\_target\_id \land \\
\qquad pca' = polPermitOverride \land \#inPolSet' = 2 \land \\
\qquad PolSet(genPSidu(Doctor, (rbach.UA^{\sim})(\!|\, \{Doctor\} \,|\!))) \in \mathrm{ran}\, inPolSet' \land \\
\qquad PolSet(genPSidu(Nurse, (rbach.UA^{\sim})(\!|\, \{Nurse\} \,|\!))) \in \mathrm{ran}\, inPolSet' \land \\
\qquad obli' = \emptyset \land \\
\qquad psid' \mapsto \theta PolicySet' \in getPolicySet'
\end{array}
$$

The next step is to evaluate $(rbach.UA^{\sim})(\!|\, \{Doctor\} \,|\!)$ and $(rbach.UA^{\sim})(\!|\, \{Nurse\} \,|\!)$ to give:

$$
\begin{array}{l}
\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\, PolicyRef;\ obli' : \mathbb{P}\, Obligation \bullet \\
\qquad psid' = genPsidA\,(rbach) \land \\
\qquad target' = empty\_target\_id \land \\
\qquad pca' = polPermitOverride \land \#inPolSet' = 2 \land \\
\qquad PolSet(genPSidu(Doctor, \{Morris, Rover\})) \in \mathrm{ran}\, inPolSet' \land \\
\qquad PolSet(genPSidu(Nurse, \{Austin, Triumph\})) \in \mathrm{ran}\, inPolSet' \land \\
\qquad obli' = \emptyset \land \\
\qquad psid' \mapsto \theta PolicySet' \in getPolicySet'
\end{array}
$$

The result of replacing $PolicySet'$ with a binding and applying the one-point rule, gives:

$$
\begin{array}{l}
\exists\, inPolSet' : \mathrm{seq}\, PolicyRef \bullet \\
\quad \#inPolSet' = 2 \land \\
\quad PolSet(genPSidu(Doctor, \{Morris, Rover\}) \in \mathrm{ran}\, inPolSet' \land \\
\quad PolSet(genPSidu(Nurse, \{Austin, Triumph\}) \in \mathrm{ran}\, inPolSet' \land \\
\quad genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow inPolSet', \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow \emptyset |\!) \in getPolicySet'
\end{array}
$$

Rearranging and utilising the constant $\#inPolSet' = 2$ gives two possibilities for *inPolSet*. This is because *inPolSet* is a sequence, which means the two values in *inPolSet* can appear with either value as the head of the sequence. This is reflected in the following:

$$
\begin{array}{l}
\exists\, inPolSet' : \mathrm{seq}\, PolicyRef \bullet \\
\quad (inPolSet' = \langle\ PolSet(genPSidu(Doctor, \{Morris, Rover\})), \\
\qquad\qquad PolSet(genPSidu(Nurse, \{Austin, Triumph\}))\ \rangle \lor \\
\quad inPolSet' = \langle\ PolSet(genPSidu(Nurse, \{Austin, Triumph\})), \\
\qquad\qquad PolSet(genPSidu(Doctor, \{Morris, Rover\}))\ \rangle) \land \\
\quad genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow inPolSet', \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow\ emptyset |\!) \in getPolicySet'
\end{array}
$$

It is then possible to distribute the $\land$ across the $\lor$ and then the $\exists$ quantifier over the $\lor$ which results in:

$$
\begin{array}{l}
(\exists\, inPolSet' : \mathrm{seq}\, PolicyRef \bullet \\
\quad inPolSet' = \langle\ PolSet(genPSidu(Doctor, \{Morris, Rover\})), \\
\qquad\qquad PolSet(genPSidu(Nurse, \{Austin, Triumph\}))\ \rangle \\
\quad \wedge \\
\quad genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow inPolSet', \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow \emptyset |) \in getPolicySet'\ ) \\
\vee \\
(\exists\, inPolSet' : \mathrm{seq}\, PolicyRef \bullet \\
\quad inPolSet' = \langle\ PolSet(genPSidu(Nurse, \{Austin, Triumph\})), \\
\qquad\qquad\qquad\qquad PolSet(genPSidu(Doctor, \{Morris, Rover\}))\ \rangle \\
\quad \wedge \\
\quad genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow inPolSet', \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow \emptyset |) \in getPolicySet'\ )
\end{array}
$$

Then applying one-point rule to each quantifier, gives.

$$
\begin{array}{l}
(genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow \langle\ PolSet(genPSidu(Doctor, \{Morris, Rover\}), \\
\qquad\qquad\qquad\qquad\qquad\qquad PolSet(genPSidu(Nurse, \{Austin, Triumph\}))\rangle, \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow \emptyset |) \in getPolicySet') \\
\vee \\
(genPsidA\,(rbach) \mapsto (\!|\ psid \rightsquigarrow genPsidA\,(rbach), \\
\qquad\qquad\qquad\qquad target \rightsquigarrow empty\_target\_id, \\
\qquad\qquad\qquad\qquad pca \rightsquigarrow polPermitOverride, \\
\qquad\qquad\qquad\qquad inPolSet \rightsquigarrow \langle\ PolSet(genPSidu(Nurse, \{Austin, Triumph\}), \\
\qquad\qquad\qquad\qquad\qquad\qquad PolSet(genPSidu(Doctor, \{Morris, Rover\}))\rangle, \\
\qquad\qquad\qquad\qquad obli \rightsquigarrow \emptyset |) \in getPolicySet')
\end{array}
$$

As in this instance the permit override combining algorithm is being used, the order of the elements in the *inPolSet* sequence is unimportant and as such either instance above would result in the same evaluation. However, it should be noted that in the general case the order may be important — For example, the case where a first applicable combining algorithm is being used. In the case of this example, the first instance will be chosen.

Next, it is convenient to introduce names for the possible values of the bindings to allow reuse and save space. Giving the name *rootps* to the root policy set binding selected from above permits

$$\exists\, PolicySet' \bullet RootPolicySetInit \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

to become

$$genPsidA\,(rbach) \mapsto rootps \in getPolicySet'$$

where the selected value of *rootps* is:

$$\begin{aligned} rootps == \langle\!| \;& psid \rightsquigarrow genPsidA\,(rbach),\\ & target \rightsquigarrow empty\_target\_id,\\ & pca \rightsquigarrow polPermitOverride,\\ & inPolSet \rightsquigarrow \langle\; PolSet(genPSidu(Doctor, \{Morris, Rover\})),\\ & \qquad\qquad PolSet(genPSidu(Nurse, \{Austin, Triumph\}))\rangle,\\ & obli \rightsquigarrow \emptyset |\!\rangle \end{aligned}$$

The remaining sections of *XACMLfromRBACInit* can be evaluated in a similar way, the results of which are detailed below. Firstly, the predicate which generates the role-user policy sets is considered:

$$\forall\, r : \mathrm{ran}(rbach.UA) \bullet \exists\, PolicySet' \bullet RoleUserPolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

$$genPSidru\,(Doctor, \{Morris, Rover\}) \mapsto rupsDoctor \in getPolicySet'$$
$$genPSidru\,(Nurse, \{Austin, Triumph\}) \mapsto rupsNurse \in getPolicySet'$$

$$\begin{aligned} rupsDoctor == \langle\!| \;& psid \rightsquigarrow genPSidru\,(Doctor, \{Morris, Rover\}),\\ & targetid \rightsquigarrow genTidPSru\,(Doctor, \{Morris, Rover\}),\\ & pca \rightsquigarrow polPermitOverride,\\ & inPolSet \rightsquigarrow \langle PolSet\,(genPSidrr\,(Doctor))\rangle,\\ & obli \rightsquigarrow \emptyset |\!\rangle \end{aligned}$$

$$\begin{aligned} rupsNurse == \langle\!| \;& psid \rightsquigarrow genPSidru\,(Nurse, \{Austin, Triumph\}),\\ & targetid \rightsquigarrow genTidPSru\,(Nurse, \{Austin, Triumph\}),\\ & pca \rightsquigarrow polPermitOverride,\\ & inPolSet \rightsquigarrow \langle PolSet\,(genPSidrr\,(Nurse))\rangle,\\ & obli \rightsquigarrow \emptyset |\!\rangle \end{aligned}$$

Each of the above policy sets captures part of the *UA* mapping from the original RBAC policy. The policy set represented by *rupsDoctor* captures all the users that map directly to the role doctor in the RBAC policy. The applicability of the policy set is controlled by the target referenced by $targetid \rightsquigarrow genTidPSru\,(Doctor, \{Morris, Rover\})$. The sequence that make up the *inPolSet* contains a single entry, the policy set reference $PolSet\,(genPSidrr\,(Doctor))$, which is the policy set that represents the role doctor from the RBAC policy.

The next predicate considered generates the role-role policy sets:

$$\forall\, r : rbach.ROLES \bullet \exists\, PolicySet' \bullet RoleRolePolicySetInit[r/r?] \land psid' \mapsto \theta PolicySet' \in getPolicySet'$$

$$genPSidrr\,(Doctor) \mapsto rrpsDoctor \in getPolicySet'$$
$$genPSidrr\,(Nurse) \mapsto rrpsNurse \in getPolicySet'$$

$$rrpsDoctor == (\!| \; psid \rightsquigarrow genPSidrr\,(Doctor), target \rightsquigarrow empty\_target\_id, \\ pca \rightsquigarrow polPermitOverride, \\ inPolSet \rightsquigarrow \langle PolSet\,(genPSidrr\,(Nurse)), Pol\,(genPid\,(Doctor))\rangle, \\ obli \rightsquigarrow \emptyset \; |\!)$$

$$rrpsNurse == (\!| \; psid \rightsquigarrow genPSidrr\,(Nurse), target \rightsquigarrow empty\_target\_id, \\ pca \rightsquigarrow polPermitOverride, inPolSet \rightsquigarrow \langle Pol\,(genPid\,(Nurse))\rangle, \\ obli \rightsquigarrow \emptyset \; |\!)$$

The following predicate is concerned with generating an XACML policy for each role in the original RBAC.

$$\forall\, r : \bigcup\{\mathrm{ran}\,(rbach.UA), \mathrm{ran}(rbach.RH), \mathrm{dom}\,(rbach.RH), \mathrm{dom}(rbach.PA)\} \bullet \\ \exists\, Policy'; \; prms : \mathbb{P}\, PRMSBase \bullet \\ prms = rbach.PA(\!|\, \{r\} \,|\!) \wedge \\ RolePolicyInit[r/r?, prms/prms?] \wedge \\ pid' \mapsto \theta Policy' \in getPolicy'$$

$$genPid\,(Doctor) \mapsto policyDoctor \in getPolicy' \\ genPid\,(Nurse) \mapsto policyNurse \in getPolicy'$$

$$policyDoctor == (\!| \; pid \rightsquigarrow genPid\,(Doctor), tid \rightsquigarrow empty\_target\_id, \\ inPol \rightsquigarrow \langle genRid\,(write, prescribeDB)\rangle, rca = rulPermitOverride, obli = \emptyset \; |\!)$$

$$policyNurse == (\!| \; pid \rightsquigarrow genPid\,(Nurse), tid \rightsquigarrow empty\_target\_id, \\ inPol \rightsquigarrow \langle genRid\,(read, PrescribeDB)\rangle, rca = rulPermitOverride, obli = \emptyset \; |\!)$$

In the above, each entry in an *inPol* sequence references a rule which represents a permission by way of a *PRMSBase*.

The following predicate ensures that each *PRMSBase* from the RBAC policy is encoded in an XACML rule.

$$\forall\, prm : \mathrm{ran}(rbach.PA) \bullet \exists\, Rule' \bullet RuleInit[prm/prm?] \wedge rid' \mapsto \theta Rule' \in getRule'$$

$$genRid\,(write, prescribeDB) \mapsto ruleWrite \in getRule' \\ genRid\,(read, prescribeDB) \mapsto ruleRead \in getRule'$$

$$ruleWrite == (\!| \; rid \rightsquigarrow genRid\,(write, prescribeDB), targetid \rightsquigarrow genTidR(write, prescribeDB), \\ condition \rightsquigarrow \langle\rangle, effect \rightsquigarrow Permit \; |\!)$$

$$ruleRead == (\!| \; rid \rightsquigarrow genRid\,(read, prescribeDB), targetid \rightsquigarrow genTidR(read, prescribeDB), \\ condition \rightsquigarrow \langle\rangle, effect \rightsquigarrow Permit \; |\!)$$

Next, the predicates associated with target generation are considered. First, the initialisation

of targets associated with the elements of UA:

$$
\begin{array}{l}
\forall\, r : \mathrm{ran}(rbach.UA) \bullet \\
\quad \exists\, Target'; \; tid : TargetID; \; sub : \mathrm{seq}\, Subject; \; act : \mathrm{seq}\, Action; \\
\quad\quad res : \mathrm{seq}\, Resource; \; env : \mathrm{seq}\, Environment \bullet \\
\quad\quad\quad tid = genTidPSru(r, (h?.UA^{\sim})(\!|\, \{r\} \,|\!)) \land \\
\quad\quad\quad sub = sequence((h?.UA^{\sim})(\!|\, \{r\} \,|\!)) \land \\
\quad\quad\quad act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\
\quad\quad\quad TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\quad\quad\quad tid' \mapsto \theta\, Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genTidPSru\,(Doctor, \{Morris, Rover\}) \mapsto tpsruDoctor \in getTarget' \\
genTidPSru\,(Nurse, \{Austin, Triumph\}) \mapsto tpsruNurse \in getTarget'
\end{array}
$$

There are two possible values for *sub* in each of the targets defined as *sub* is modelled as a sequence. The evaluation of a *Target* that is used only checks if a particular subject is in the sequence; the order in which the subjects occur is not a concern as either order will result in the same evaluation. The following chooses one of the possible sequences.

$$
\begin{array}{l}
tpsruDoctor == (\!|\; tid \rightsquigarrow genTidPSru\,(Doctor, \{Morris, Rover\}), \\
\qquad\qquad\quad sub \rightsquigarrow \langle targetElementEquals\ Morris, targetElementEquals\ Rover\rangle, \\
\qquad\qquad\quad act \rightsquigarrow \langle\rangle, res \rightsquigarrow \langle\rangle, env \rightsquigarrow \langle\rangle \,|\!) \\
tpsruNurse == (\!|\; tid \rightsquigarrow genTidPSru\,(Nurse, \{Austin, Triumph\}), \\
\qquad\qquad\quad sub \rightsquigarrow \langle targetElementEquals\ Austin, targetElementEquals\ Triumph\rangle, \\
\qquad\qquad\quad act \rightsquigarrow \langle\rangle, res \rightsquigarrow \langle\rangle, env \rightsquigarrow \langle\rangle \,|\!)
\end{array}
$$

Finally, the targets associated with the rules which encode each *PRMSBase* are condsidered.

$$
\begin{array}{l}
\forall\, prm : \mathrm{ran}(rbach.PA) \bullet \\
\quad \exists\, Target'; \; tid : TargetID; \; sub : \mathrm{seq}\, Subject; \; act : \mathrm{seq}\, Action; \\
\quad\quad res : \mathrm{seq}\, Resource; \; env : \mathrm{seq}\, Environment \bullet \\
\quad\quad\quad tid = genTidR\, prm \land sub = \langle\rangle \land \\
\quad\quad\quad act = \langle first\ prm\rangle \land res = \langle second\ prm\rangle \land env = \langle\rangle \land \\
\quad\quad\quad TargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\quad\quad\quad tid' \mapsto \theta\, Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genTidR\,(write, prescribeDB) \mapsto trWrite \in getTarget' \\
genTidR\,(read, prescribeDB) \mapsto trRead \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
trWrite == (\!|\; tid \rightsquigarrow genTidR\,(write, prescribeDB), sub \rightsquigarrow \langle\rangle, \\
\qquad\qquad act \rightsquigarrow \langle targetElementEquals\ write\rangle, \\
\qquad\qquad res \rightsquigarrow \langle targetElementEquals\ prescribeDB\rangle, env \rightsquigarrow \langle\rangle \,|\!) \\
trRead == (\!|\; tid \rightsquigarrow genTidR\,(read, prescribeDB), sub \rightsquigarrow \langle\rangle, \\
\qquad\qquad act \rightsquigarrow \langle targetElementEquals\ read\rangle, \\
\qquad\qquad res \rightsquigarrow \langle targetElementEquals\ prescribeDB\rangle, env \rightsquigarrow \langle\rangle \,|\!)
\end{array}
$$

Finally, it is possible to combine all the above sections together and give the name *exXACML* to the XACML generated from the application of *XACMLfromRBACInit*.

$$exXACML \in \{XACMLfromRBACInit \mid h? = rbach \bullet \theta ValidXACML'\}$$

This gives the following mappings as being members of the various functions.

$$genPsidA\,(rbach) \mapsto rootps \in getPolicySet'$$
$$genPSidru\,(Doctor, \{Morris, Rover\}) \mapsto rupsDoctor \in getPolicySet'$$
$$genPSidru\,(Nurse, \{Austin, Triumph\}) \mapsto rupsNurse \in getPolicySet'$$
$$genPSidrr\,(Doctor) \mapsto rrpsDoctor \in getPolicySet'$$
$$genPSidrr\,(Nurse) \mapsto rrpsNurse \in getPolicySet'$$
$$genPid\,(Doctor) \mapsto policyDoctor \in getPolicy'$$
$$genPid\,(Nurse) \mapsto policyNurse \in getPolicy'$$
$$genRid\,(write, prescribeDB) \mapsto ruleWrite \in getRule'$$
$$genRid\,(read, prescribeDB) \mapsto ruleRead \in getRule'$$
$$genTidPSru\,(Doctor, \{Morris, Rover\}) \mapsto tpsruDoctor \in getTarget'$$
$$genTidPSru\,(Nurse, \{Austin, Triumph\}) \mapsto tpsruNurse \in getTarget'$$
$$genTidR\,(write, prescribeDB) \mapsto trWrite \in getTarget'$$
$$genTidR\,(read, prescribeDB) \mapsto trRead \in getTarget'$$
$$empty\_target\_id \mapsto empty\_target \in getTarget'$$

Although theoretically there could be additional members of the functions this is prevented by the constraints in the initialisation schemas and the *ValidXACML* schema. This gives us:

$$\begin{aligned} exXACML == \langle\!| \; & getPoliySet \rightsquigarrow \{ \; genPsidA\,(rbach) \mapsto rootps, \\ & \qquad genPSidru\,(Doctor, \{Morris, Rover\}) \mapsto rupsDoctor, \\ & \qquad genPSidru\,(Nurse, \{Austin, Triumph\}) \mapsto rupsNurse, \\ & \qquad genPSidrr\,(Doctor) \mapsto rrpsDoctor, \\ & \qquad genPSidrr\,(Nurse) \mapsto rrpsNurse\} \\ & getPolicy \rightsquigarrow \{genPid\,(Doctor) \mapsto policyDoctor, genPid\,(Nurse) \mapsto policyNurse\} \\ & getRule \rightsquigarrow \{genRid\,(write, prescribeDB) \mapsto ruleWrite, \\ & genRid\,(read, prescribeDB) \mapsto ruleRead\} \\ & getTarget \rightsquigarrow \{genTidR\,(write, prescribeDB) \mapsto trWrite, \\ & genTidR\,(read, prescribeDB) \mapsto trRead, \\ & genTidPSru\,(Doctor, \{Morris, Rover\}) \mapsto tpsruDoctor, \\ & genTidPSru\,(Nurse, \{Austin, Triumph\}) \mapsto tpsruNurse\} \\ & rootPol \rightsquigarrow \{PolSet(rootps.psid)\} |\!\rangle \end{aligned}$$

When considering the overall policy, *exXACML*, the existence of sequences — which were generated from sets — in the policy sets allows for there to be several equally valid bindings based on the order of the elements in the sequences. This does not pose a problem for the translation as the order of the sequence has no bearing on the overall evaluation of the policy set — it is only the range of the sequence that is important. This is only the case because the translation relies upon the permit override algorithm for combining the various elements. It is therefore valid to choose the above binding for *exXACML* which contains the mapping $genPsidA\,(rbach) \mapsto rootps$,

this mapping being the same as the one chosen above during the walk-through.

The output generated by the application of the formal translation process has been used to manually populate an XACML policy. The XACML policy generated was then used to evaluate a number of access control requests and the results compared against the results obtained by evaluating the requests against the original RBAC. This process verified that the XACML policy that resulted from the translation was consistent with the original XACML policy. A prototype application, described in Chapter 7, was developed to permit the testing of much larger access control examples.

6.4 Alloy

6.4.1 RBAC to XACML

In this section we define the operations necessary to convert policies captured in the RBAC model into corresponding policies in the XACML model; in addition, a predicate characterising validity is also defined. The validity assertion states that for any request presented to any RBAC instance, the resulting permission will be the same as that obtained by applying the same request to the XACML instance returned by applying the translation to the RBAC instance. When considering the returned permission, an XACML `NotApplicable` is considered to be equivalent to a `Deny`.

6.4.2 A formal translation

To facilitate the translation from RBAC to XACML, it is necessary to define a number of signatures that specialise the `Policy` and `PolicySet` signatures, this is achieved by adding constraints which are based on the values of the new fields defined within the signature. We initially define a `rolePolicy`, which has an extra field, `prms` — a set of elements of type PRMSBase. The constraints of the resulting signature ensure that the target, `t`, of the policy is the `EmpyTarget` and that the rules in `inPol` correspond to one of the entries in `prms` such that the rule has a target with `act` and `res` set to be the action and resource of the corresponding entry in `prms`. Finally, the effect, `eff`, is constrained to be `Permit`.

```
module accesscontrol/rbactoxacml

open accesscontrol/rbach
open accesscontrol/xacml

sig rolePolicy extends Policy{
    prms : set PRMSBase
}
{
    t = EmptyTarget
    all prm : prms |
        some ru : Rule |
            ru in inPol &&
```

```
            ru.@t.sub = none &&
            ru.@t.act = prm.action &&
            ru.@t.res = prm.resource &&
            ru.eff = Permit

    # inPol = # prms

}
```

The `rolerolePolicySet` signature extends `PolicySet` by including the fields `r` of type `Role` and `h` of type `Hierarchy`. The constraints result in a policy set that captures all the permissions either directly associated with the role `r` or inherited by `r` via the role hierarchy, `RH` in `h`. The constraint part of the signature makes the policy set universally applicable by stating the target `t` is the `EmptyTarget`. The next part of the constraint ensures a policy relating to the role `r` is included in the `inPolSet` only if permissions are associated with the role in the `h.PA` relation, with the guard `#r.(h.PA) > 0` performing this check. The following part of the constraint ensures a policy set is included in `inPolSet` for each of the roles that role `r` inherits from directly in the `h.RH` relation. Finally, the number of elements in `inPolSet` is constrained to be the number of roles associated with role `r` in `h.RH` plus one if there are permissions directly associated with `r` in `h.PA`.

```
sig rolerolePolicySet extends PolicySet {
    r : Role,
    h : Hierarchy
}
{
    t = EmptyTarget
    (#r.(h.PA) > 0 =>
        one rp : rolePolicy | rp.prms = r.(h.PA) && rp in inPolSet)

    (#(r.(h.RH)) > 0 =>
        all ro : r.(h.RH) |
            one rrps : rolerolePolicySet |
                rrps.@r = ro && rrps.@h = h && rrps in inPolSet)

    #r.(h.PA) > 0 => #inPolSet =  1 + #(r.(h.RH))
        else  #inPolSet = #(r.(h.RH))
}
```

Next, `roleusersPolicySet` is defined, which extends `PolicySet` and has the additional elements `r` (of type `Role`) and `h` (of type `Hierarchy`). The first part of the constraints restrict the target to being the target which has a `sub` that contains the users within the hierarchy `h` which are directly associated with the role `r` via `h.UA`. The `act` and `res` of the target are set to `none`, so they can match any value. The final part constrains `inPolSet` to contain a single `rolerolePolicyset` policy set that has the role set to `r` and hierarchy set to `h`.

```
sig roleusersPolicySet extends PolicySet {
    r : Role,
    h : Hierarchy
}
{
    t.sub = (h.UA).r
    t.act = none
    t.res = none

    some rrps : rolerolePolicySet | rrps in inPolSet && rrps.@r = r && rrps.@h = h
    one inPolSet
}
```

The final signature, `rootPolicySet`, also extends `PolicySet` by the addition of an element `h` (of type `Hierarchy`), along with some constraints. The first constraint requires the target `t` to be the empty target. The following constraint populates `inPolSet` with a `roleusersPolicySet` for each unique role in the `UA` relation from `h`. Finally, the number of elements in `inPolSet` is constrained to be the number of unique role in the `h.UA`, given by `# (User.(h.UA))`.

```
sig rootPolicySet extends PolicySet {
    h : Hierarchy
}
{
    t = EmptyTarget

    all r1 : User.(h.UA) |
        some rup : roleusersPolicySet |
            rup in inPolSet && rup.@r = r1 && rup.@h = h

    # inPolSet = # (User.(h.UA))

}
```

Finally it is possible to define a predicate, `validXACML`, which, for an element of `Hierarchy` and an element of `XACML`, states that the XACML `rootPolicy` set is the `rootPolicySet` that has `h` set to be the hierarchy supplied.

```
pred validXACML (h: Hierarchy, x :  XACML ) {
    some rps : rootPolicySet | x.rootPolSet = rps && rps.@h = h
}
```

A final predicate, `convertEQ`, is defined, which takes an element of `Hierarchy`, `h`, and an element of `Request`, `req`, and checks if the result of evaluating `h` in relation to `req` is the same as evaluating a `validXACML` conversion of the hierarchy in relation to the request. In the matching

of results, a `Deny` from an element of `Hierarchy` can match either a `Deny` or a `NotApplicable` from an element of `XACML`.

```
pred convertEQ (h : Hierarchy, req : Request){
    all x : XACML |  validXACML[h, x] =>
        (
            ( evalRBACH[h,req] = Permit && evalXACML[x,req] = Permit ) ||
            ( evalRBACH[h,req] = Deny && evalXACML[x,req] = Deny ) ||
            ( evalRBACH[h,req] = Deny && evalXACML[x,req] = NotApplicable )
        )
}
```

In addition a second predicate is defined which is designed to fail, as the case where XACML returns `NotApplicable` is ignored; this is included as a simple test that the model is being checked fully.

```
pred convertEQfail (h : Hierarchy, req : Request){
    all x : XACML |  validXACML[h, x] =>
        (
            ( evalRBACH[h,req] = Permit && evalXACML[x,req] = Permit ) ||
            ( evalRBACH[h,req] = Deny && evalXACML[x,req] = Deny )
        )
}
```

6.4.3 Validation assertions

It is now necessary to define assertions to permit the testing of the translations to provide assurance that the resulting XACML retains the same properties as the original RBAC hierarchy. To this end, an assertion is defined, which, for every (`Hierarchy`, `Request`) pairing, the result of the request applied to the hierarchy is the same as the result of the request applied to the conversion utilising the `convertEQ` predicate. The check statement has constraints put on it to allow the execution of the assertion looking for counter-examples to occur in a reasonable period. A second assertion is also defined that should fail, `eqFail`, which uses the `convertEQfail` predicate above. This assertion should generate a counter-example in which the RBAC evaluation returns `Deny` and the XACML evaluation returns `NotApplicable`.

```
module accesscontrol/rbaccoretoxacmltest
open accesscontrol/rbaccoretoxacml

assert eq { all rbac : Hierarchy, req : Request | convertEQ [rbac, req] }

check eq for 9 but 6 int, 1 Hierarchy, 1 XACML, 3 Action, 3 Resource,
    3 User, 1 Core
```

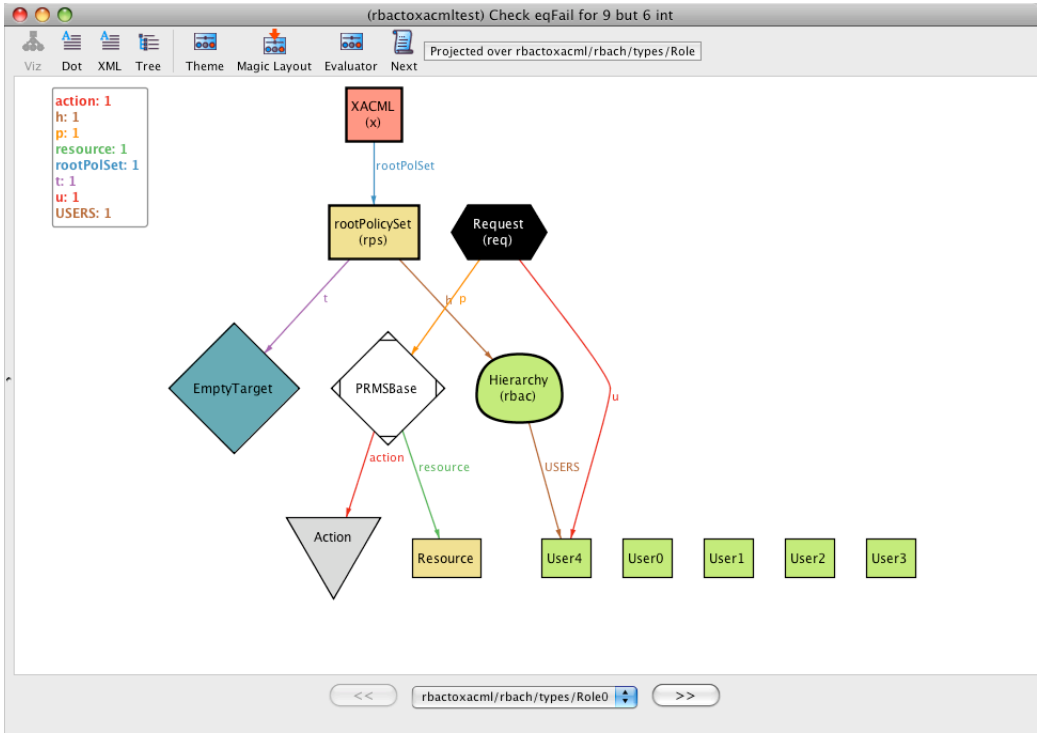

Figure 6.2: Testing the conversion for a failure by ignoring `NotApplicable`

```
assert eqFail { all rbac : Hierarchy, req : Request | convertEQfail [rbac, req] }

check eqFail for 9 but 6 int
```

The following is the result of executing the `check eq` which shows that for the scope under consideration no counter-examples were found.

```
Executing "Check eq for 9 but 6 int, 1 Hierarchy, 1 XACML, 3 Action, 3 Resource,
    3 User, 1 Core"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=9 SkolemDepth=1 Symmetry=20
   52058 vars. 1456 primary vars. 130079 clauses. 1555ms.
   No counterexample found. Assertion may be valid. 14721867ms.
```

In addition, the failure case has also been checked with the result shown below and a counter-example illustrated in Figure 6.2. In the counter-example found, the root policy set has no policies or rules and so will always return `NotApplicable` regardless of the request — which, for the purpose of finding a counter-example, has been distinguished from `Deny`.

```
Executing "Check eqFail for 9 but 6 int"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=9 SkolemDepth=1 Symmetry=20
   138754 vars. 4346 primary vars. 355345 clauses. 957ms.
   Counterexample found. Assertion is invalid. 137ms.
```

6.5 Alloy translation of the running example

Using the same Alloy instance of *Hierarchy* described in Chapter 4, it is possible to run a number of tests with sample requests to validate that the results of applying the request to an RBAC instance is functionally equivalent to the result of applying the request to the XACML instance generated by the translation.

```
-- Check all access control decisions are equivelant
assert evalTransAllEquivelant{
  all x: XACML | validXACML[prescribeh, x] =>
    all req : allRequests |  (
      ( evalRBACH[prescribeh,req] = Permit && evalXACML[x,req] = Permit ) ||
      ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = Deny ) ||
      ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = NotApplicable ) )
}

check evalTransAllEquivelant for 8 but 14 XACMLelementBase , 6 int , 1 XACML ,
      1 Hierarchy, 2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

Executing "Check evalTransAllEquivelant for 8 but 6 int, 14 XACMLelementBase,
       1 XACML, 1 Hierarchy, 2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=8 SkolemDepth=1 Symmetry=20
   70337 vars. 2072 primary vars. 181780 clauses. 581ms.
   No counterexample found. Assertion may be valid, as expected. 39519ms.
```

The predicates and assertions used to further test the Alloy version of the translation are included in Appendix C.2, with the results given in Appendix C.3. The tests are designed to exercise the model and demonstrate the validity of the approach. The initial set of tests check that the various component parts of the translation mechanism create the expected instances. The remainder of the tests are concerned with the correctness of the evaluations with regard to requests. A number of requests are defined, with each being evaluated by the original RBAC policy and the XACML policy which resulted from the translation. An assertion is made that the results of the evaluations are equivalent, and this assertion is checked. In addition a number of false assertions are included to validate that everything is executing correctly.

6.6 XACML to RBAC

For the sake of completeness, this section considers the restrictions that would need to be imposed on an XACML policy set which would be sufficient to permit a translation from XACML to RBAC to occur. It is recognised that, in the general case, it would not be possible to convert XACML into RBAC. The primary reason for the inability to produce a generalised translation is the difference in expressiveness between XACML and RBAC. This expressiveness results in XACML being capable of capturing far more complex access control requirements than would be possible using simple RBAC. For example, an XACML policy can:

- contain negative rules that if applicable return `Deny`;
- combine rules, policies and policy sets in a number of different ways including permit override, deny overrides and first applicable;
- have complex target matching defining the applicability of rule, policies and policy sets; and
- have additional conditions applied to rules that further constrain the result returned by a rule.

For a subset of XACML policies it would be possible to provide a simple translation into RBAC. However, for an XACML policy to be suitable for translation into RBAC, it would need to conform to the following restrictions:

1. No rule can have a condition statement which references external or dynamic state; it would, however, be permissible to have a condition involving a complex relationship between the user, action and resource from the request.
2. A definitive set of all users for which the RBAC policy will provide access control.
3. A definitive set of actions and resources for which access control is to be provided.

This would result in the following sets.

$$[AllUsers, AllActions, AllResources]$$

These sets could be used to generate an exhaustive list of all possible permissions by taking the cross-product of *AllActions* and *AllResources*.

$$AllPossiblePermissions : AllActions \times AllResources$$

Once all the possible permissions are defined the translation could generate a role associated with each permission. It is then possible to calculate which users should be associated with each role-based on the effective permissions a user has. A possible method for calculating the relationship between users and roles would be to cycle through every possible request by taking a user from *AllUsers* and a permission from *AllPossiblePermissions* and applying it to the XACML policy. If this evaluation returns *Permit* than the user has the role associated with the permission in the equivalent RBAC. This will generate a core RBAC policy; if, however, a hierarchical RBAC policy is preferred, the core RBAC could be translated into a hierarchy using techniques described in [85].

Although an XACML profile for RBAC exists (see [75]) it does not necessarily make the translation to RBAC any more straightforward. There are several possible approaches that could be taken when performing such a translation, although the approach outlined above would still be valid providing it is possible to generate and evaluate the appropriate requests. The evaluation of all the necessary requests may be problematic depending how the system being used provides the user-to-role mappings. In [75], a comment is made that either the *Request* must contain the direct roles held by the user or the PDP's context handler must be able to discover them. If the

user-role mappings can be obtained and separate requests used to infer the role-to-permission mappings, it would be possible to combine the two mappings into a flat RBAC description, which could be given a hierarchy in the way suggested previously. It is, however, not possible to make more specific comments as the XACML RBAC profile leaves a number of details undefined and state they are an implementation concern.

For a particular implementation, it should be possible to produce a formal definition of the policy and subsequently define a translation that maintains the full role hierarchy defined within the XACML representation.

6.7 Summary

This chapter has presented a Z and an Alloy version of a translation process that will generate XACML policies from RBAC policies. The Z model was presented first, followed by a walked-through translation of our running example. The Alloy version of the translation process was validated by a number of assertions and predicates that checked the consistency of the results given to access control requests by the original RBAC captured in Alloy and the XACML resulting from the application of the translation.

The formal definition of the translation process, along with the confidence gained in the process by virtue of the Alloy representation, successfully meets the requirement of providing consistent translation between access control paradigms — at least for these two examples — discussed in Chapter 3.

The model defined within this chapter is subsequently used in Chapter 7 to inform the implementation of an application that permits the capture of an access control policy in terms of RBAC concepts and then permits the translation of the RBAC policy into an equivalent XACML policy. In addition, the application also permits certain properties of the RBAC policy to be checked prior to the translation being performed, as well as allowing the comparison of the results of access control requests posed to both the RBAC and XACML versions of the policy.

This chapter has also commented on the possibility of translating from XACML into RBAC and noted that it would only be possible for a subset of XACML policies that meet certain criteria.

Chapter 7

Access control policy tool

This chapter describes a prototype application that has been written to capture access control policies using the RBAC representation of Chapter 4, and convert them to an equivalent policy in the XACML representation of Chapter 5. The application is only concerned (currently) with RBAC and XACML as these are the models considered for the proof-of-concept approach of this thesis. In addition to the translation between RBAC and XACML, the prototype application also allows some properties of the RBAC policy to be checked, as well as facilitating the evaluation of access control requests against the different representations of the policy.

7.1 Prototype RBAC policy translation application

The prototype RBAC policy translation application is a Java application that allows the editing and validation of RBAC policies, as well as the translation of these into XACML. The overall work flow of the application is illustrated in Figure 7.1. The application permits the reading of an existing RBAC policy or the development of a new one. The policy can then be edited to reflect the access control requirements. A simple validation facility is also provided that allows the policy to be checked against a number of user-defined criteria to increase the confidence that the writers have in the policy. The validation facility utilises Alloy to check the policy against defined predicates to highlight any inconsistencies. The resulting policy can then be saved as an RBAC policy as well as being translated into XACML and written out to an XACML file. The resulting file can then be used by an access control engine to make decisions pertaining to access control requests. The resulting policies can then be uploaded to the sif middleware of Section 2.7.3, with an example of this process being described in the case study presented in Chapter 8. The next section provides a more detailed description of the prototype application.

7.2 Translation application interface

The interface of the application is divided into a number of panes as can be seen in Figure 7.2. Each of the individual panes is designed to perform a particular task. Some of the panes capture information pertaining to a single aspect of an RBAC policy; others facilitate the validation of a

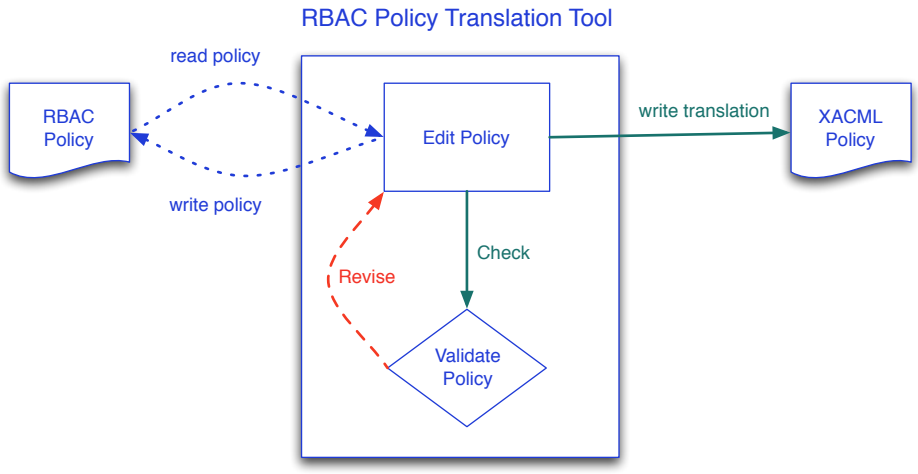

Figure 7.1: RBAC policy translation tool

policy; the final one permits the translation of the RBAC policy into XACML and allows testing to be performed on the resulting policy. Each of the panes is now covered in greater detail.

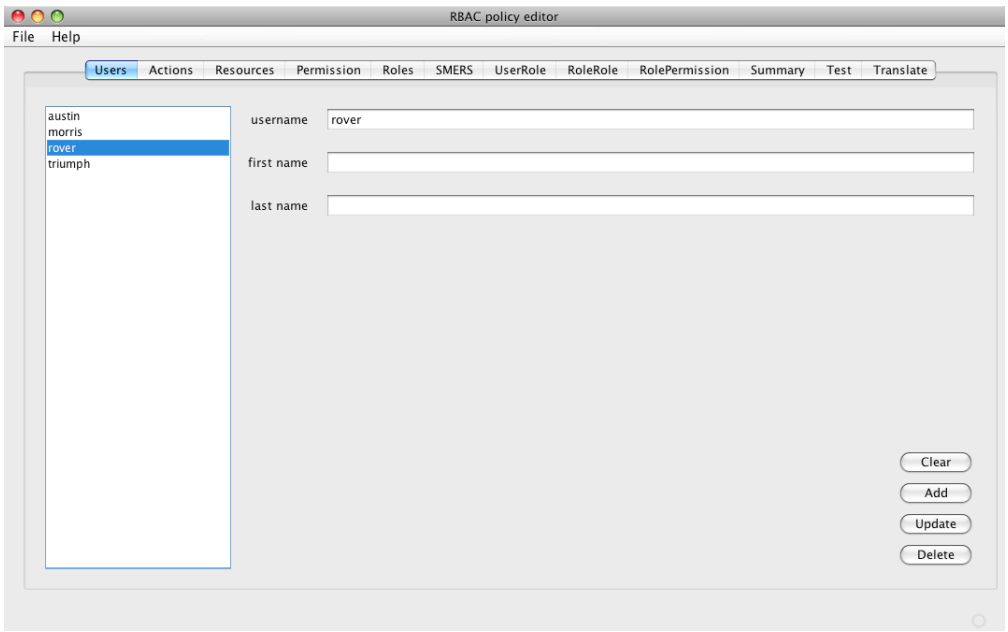

Figure 7.2: RBAC policy translation tool: user panel

In the following description of the policy tool, the capturing of the RBAC version of the running example will be used for illustration. The first three panes facilitate the collection of basic information about users, actions and resources. Figure 7.2 illustrates the adding of a new user *rover* via the user pane. The entering of the action *write* is illustrated in Figure 7.3 with the resource pane in Figure 7.4 capturing the resource *PrescribeDB*.

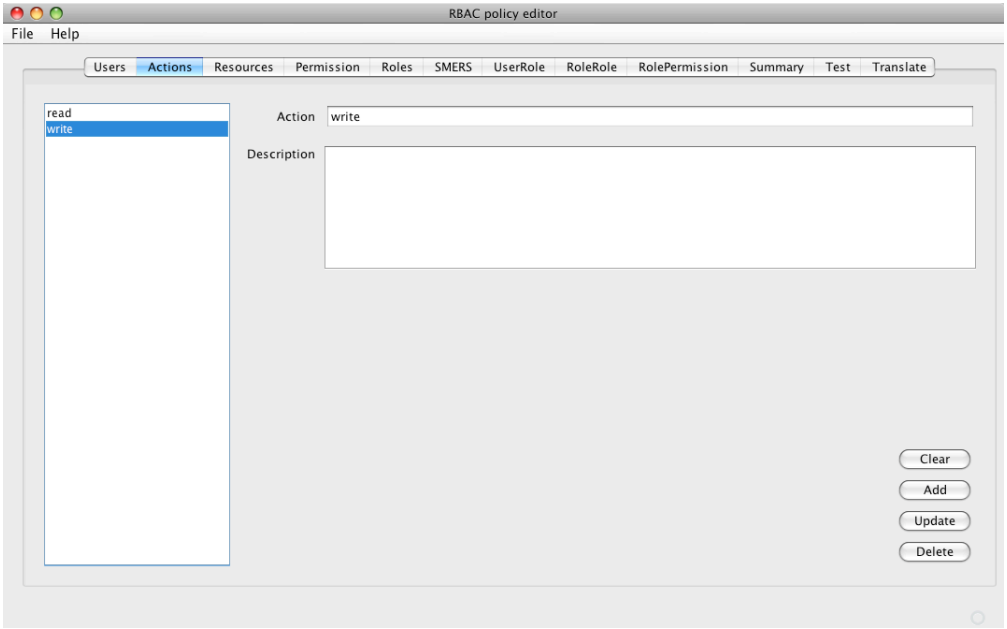

Figure 7.3: RBAC policy translation tool: action panel

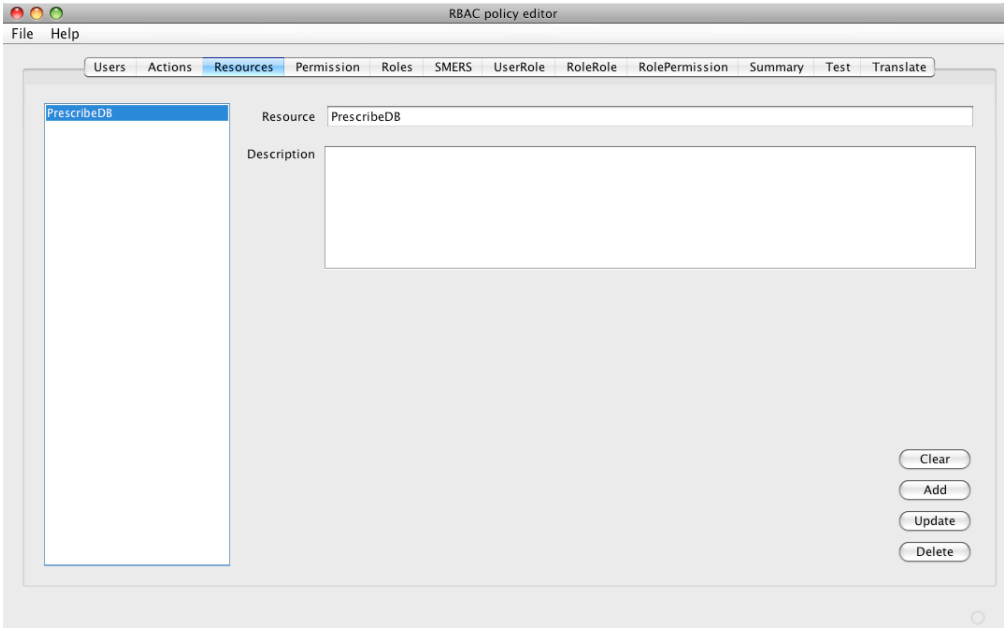

Figure 7.4: RBAC policy translation tool: resource panel

Once actions and resources have been defined it is then possible to define permissions. The next pane allows for the capturing of permissions, where each permission is a combination of a single action and a single resource, which is consistent with the formal definition of *PRMSBase* of Chapter 4. Figure 7.5 illustrates the capturing of the permission relating the action *write* to the resource *prescribeDB* and giving it the name *writePrescription*. The pane also shows a

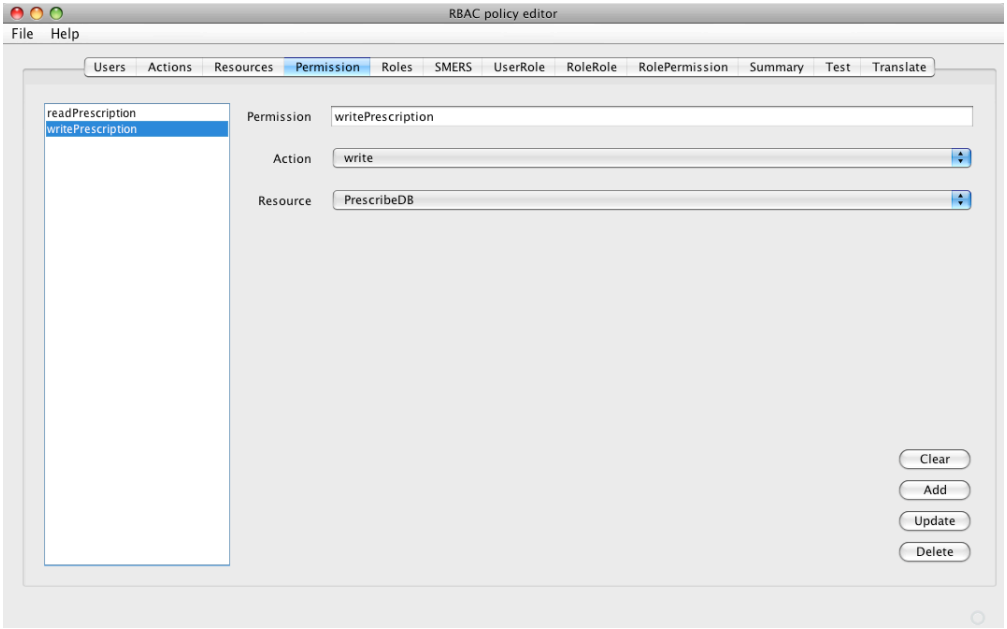

Figure 7.5: RBAC policy translation tool: permission panel

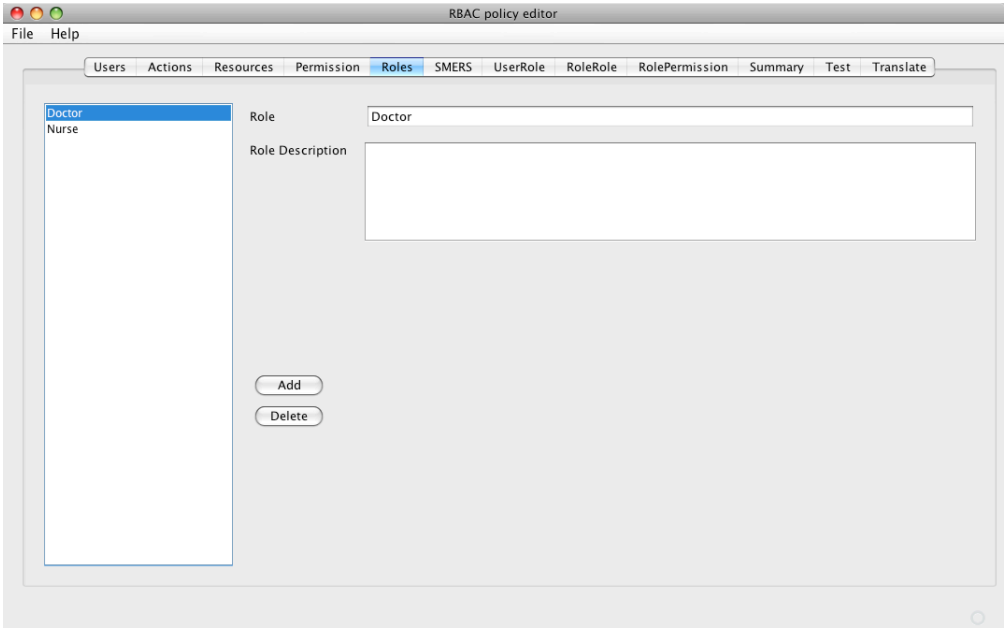

Figure 7.6: RBAC policy translation tool: role panel

second permission *readPrescription* in the list on the left-hand side. Hence, the first four panes essentially allow the capture of the basic information required to develop policies which represent restrictions on who can perform actions on given resources.

The fifth pane captures basic information pertaining to roles. It allows the entry of the name for each of the roles along with a description of the role's intended purpose. Figure 7.6 shows the

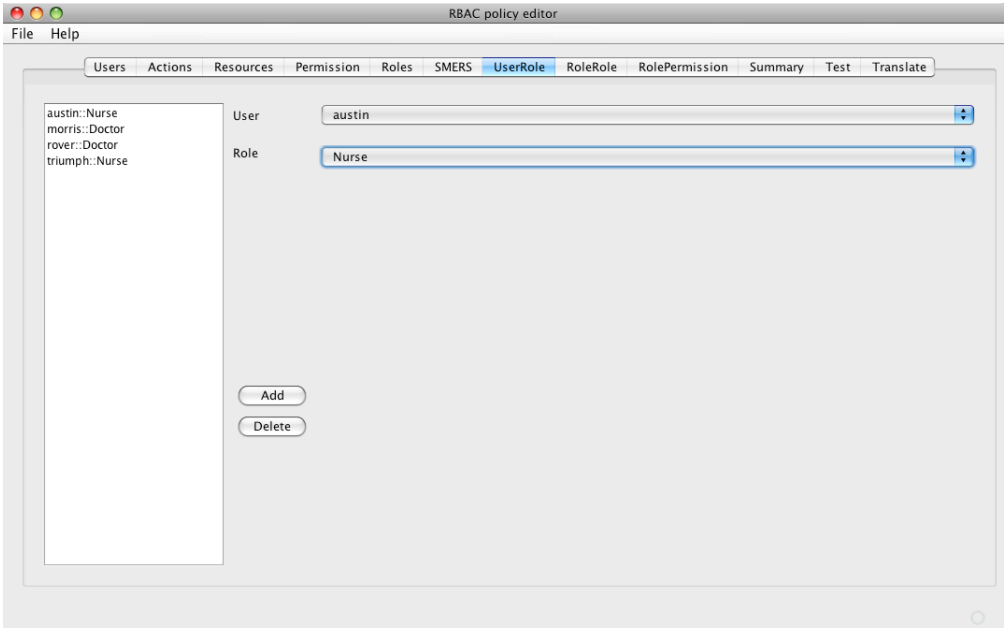

Figure 7.7: RBAC policy translation tool: user-role panel

role *Doctor* being entered and that the role *Nurse* also exists in the list of roles on the left-hand side of the pane.

The sixth pane enables the capturing of SMER constraints (see Section 4.1.6) to which the policy has to conform. This is achieved by the defining of names for SMER constraints which then associate a set of roles with a number indicating the maximum number of roles from the set that any given user may be associated with. As the running example does not have any SMER constraints, an illustration of this pane has been omitted.

The following three panes permit the capturing of the main RBAC relationships: *UA*, *RH* and *PA*. The user-role pane, illustrated in Figure 7.7, captures the *UA* relation by facilitating the allocation of roles to users, in the case illustrated associating the user *austin* with the role *Nurse*. The role-role pane allows the capturing of the *RH* relation by permitting the definition of a role hierarchy, which essentially states which roles directly inherit the permissions of other roles. The capturing of the fact that the role *Doctor* inherits the permissions of the role *Nurse* is illustrated in Figure 7.8. The final pane in this group of three is the role-permission pane which captures the *PA* relation by allowing permissions to be allocated to roles. The granting of the permission *readPrescription* to the role *Nurse* is shown in Figure 7.9.

The next pane is a summary pane which allows the policy editor to manually check that users have the collection of permissions that they should have. This is achieved by selecting a user and reviewing the information displayed which relates to the roles the user has been given explicitly, the roles the user has inherited (if any), along with the possible routes of inheritance and finally the effective set of permissions that the user has — annotated with the roles which grant the particular permission.

In Figure 7.10 the user *morris* is considered. The summary indicates that *morris* has been directly assigned the role *Doctor*. In addition, *morris* has the role *Nurse*, which is an inherited

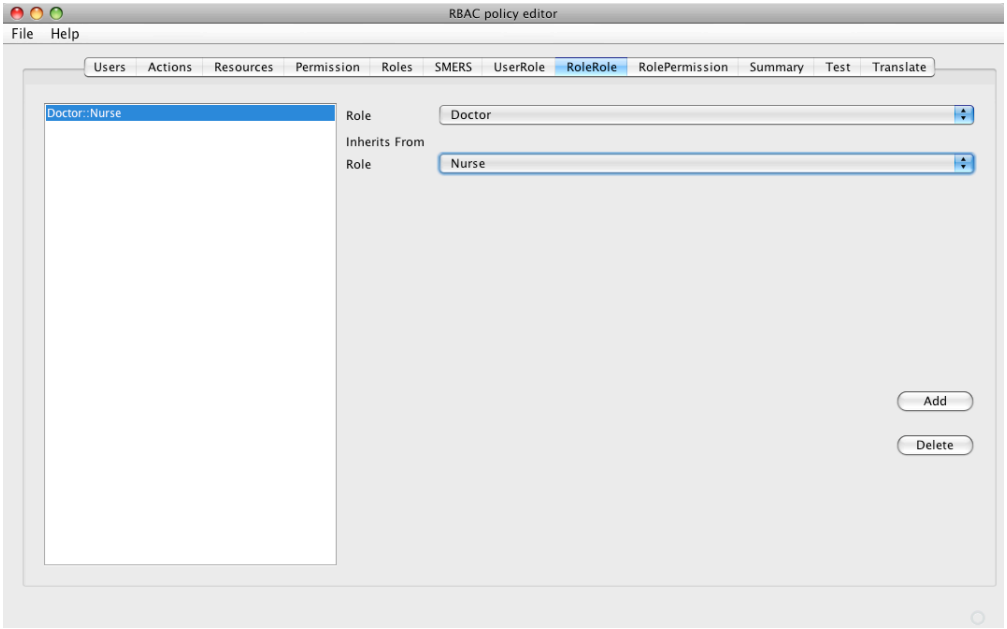

Figure 7.8: RBAC policy translation tool: role-role panel

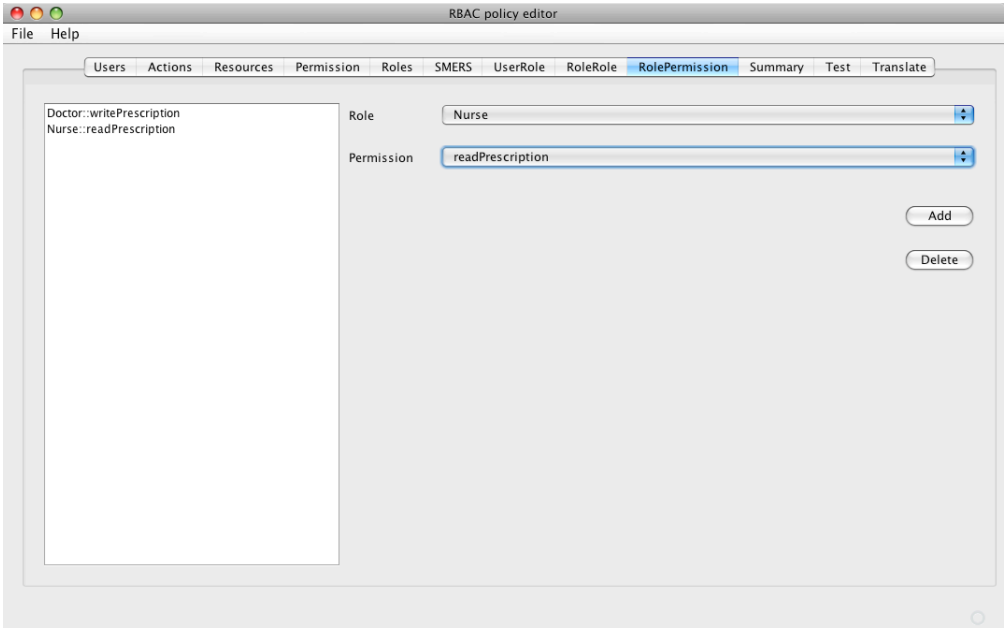

Figure 7.9: RBAC policy translation tool: role-permission panel

role via the role *Doctor*. Finally, the effective permissions that *morris* has are listed along with the role which is associated with the permission. In this case, *morris* has permission to perform the action *read* on the resource *PrescribeDB* by virtue of having the *Nurse* role. *morris* also has the permission to perform the action *write* on the resource *PrescribeDB* by being associated with the role *Doctor*.

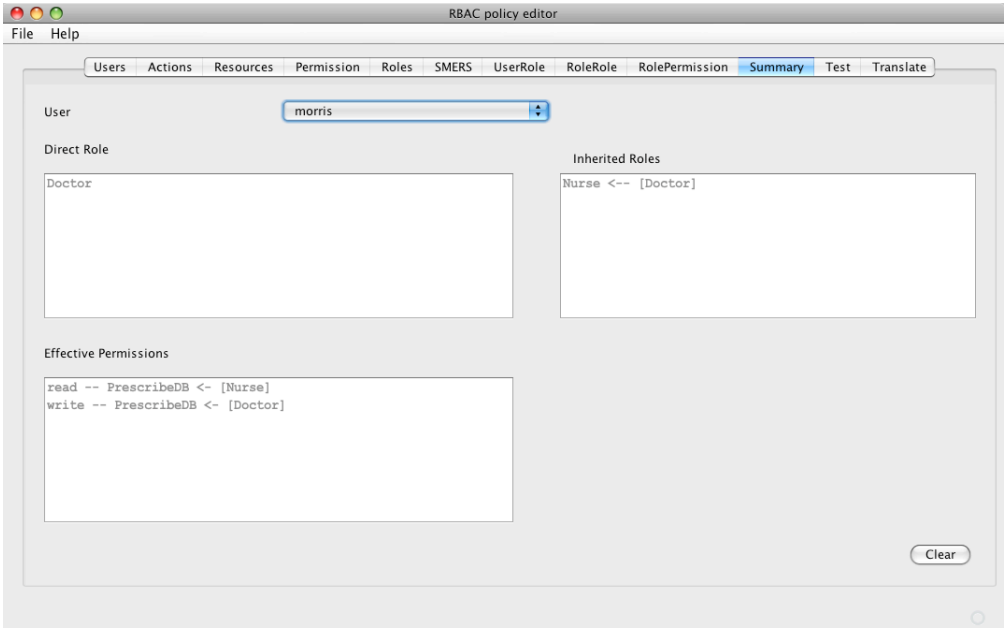

Figure 7.10: RBAC policy translation tool: summary panel

The penultimate pane performs two functions and is illustrated in Figure 7.11. The top half of the pane permits the user to define an access control request by selecting a user, an action and a resource. Once the request is defined it can be tested against the RBAC policy which has been constructed, with the result of the evaluation being displayed in the output panel adjacent to the request selection drop-downs. An example of this is shown in Figure 7.11, which captures the request (shown on the left-hand side) of *morris* asking to perform the action *write* on the resource *PrescribeDB*. The result of the request is shown on the right-hand side, which in this case is *Permit*. In addition to the actual result of the request, the result area also echos the request evaluated as an extra verification that the correct request was used.

The lower part of the pane is used to check that the policy is consistent with a number of constraints (this is described in detail in Section 7.3). The testing of a single constraint *AllPermissionsReachable* is also illustrated in the bottom half of Figure 7.11. This additional constraint performs a check on an Alloy representation of the defined policy to identify any permissions that have been defined but not associated with a role.

The final pane provides access to the mechanisms necessary to perform the translation between RBAC and XACML, along with a method for testing that the XACML generated is consistent with the original RBAC. The top portion of the pane has a section for describing a request (in a similar way to the previous pane, which captures the fact that a user wishes to perform an action on a resource), along with the evaluation panels that display the result of applying that request to the RBAC and XACML representations of the policy, which permits manual comparison of the results to confirm consistency. The top half of Figure 7.12 shows the evaluation of the request by *morris* to perform the action *write* on the resource *PrescribeDB* against the original RBAC and the XACML policy generated by the translation process. In this case, the RBAC and XACML evaluations of the request both return *Permit*.

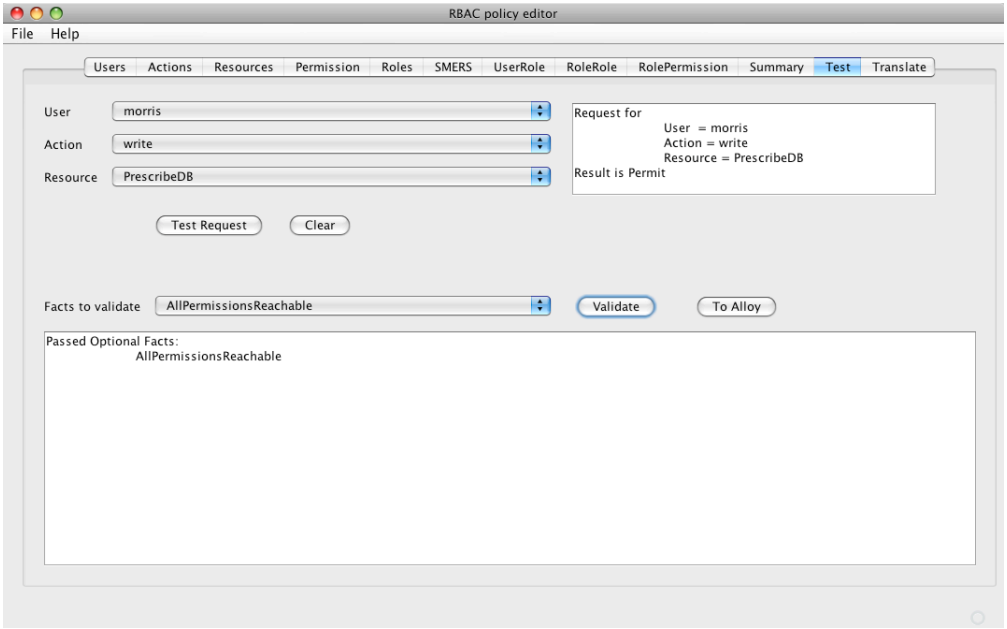

Figure 7.11: RBAC policy translation tool: test panel

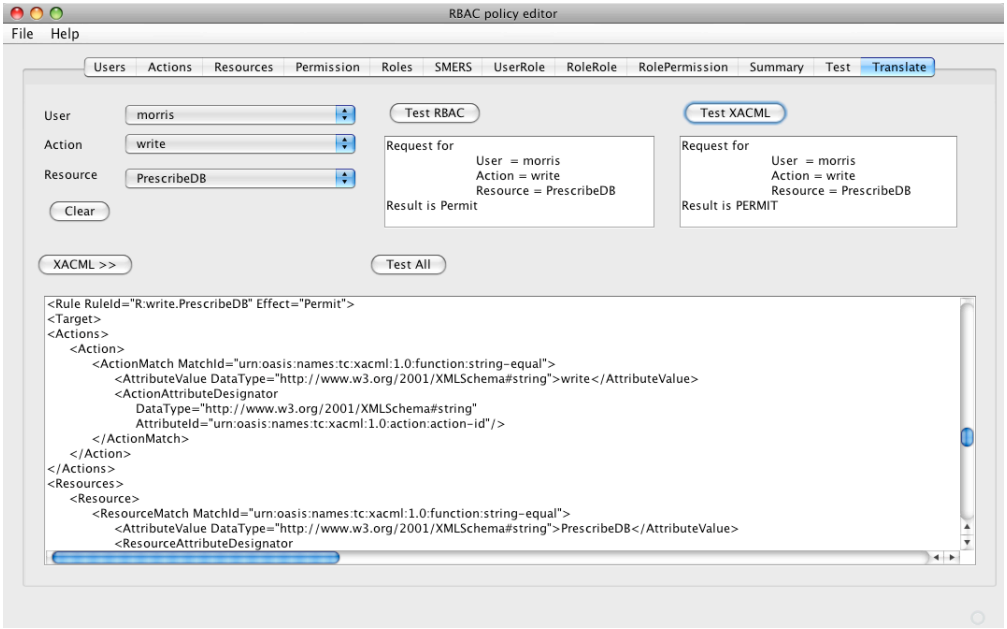

Figure 7.12: RBAC policy translation tool: translate panel showing XACML

The bottom portion of the pane can behave in two distinct ways. First, it can display the XACML that results from performing the translation, which is illustrated in Figure 7.12. The second mode allows the results of multiple requests to be reported automatically, which is described in Section 7.4 and is illustrated in Figure 7.13.

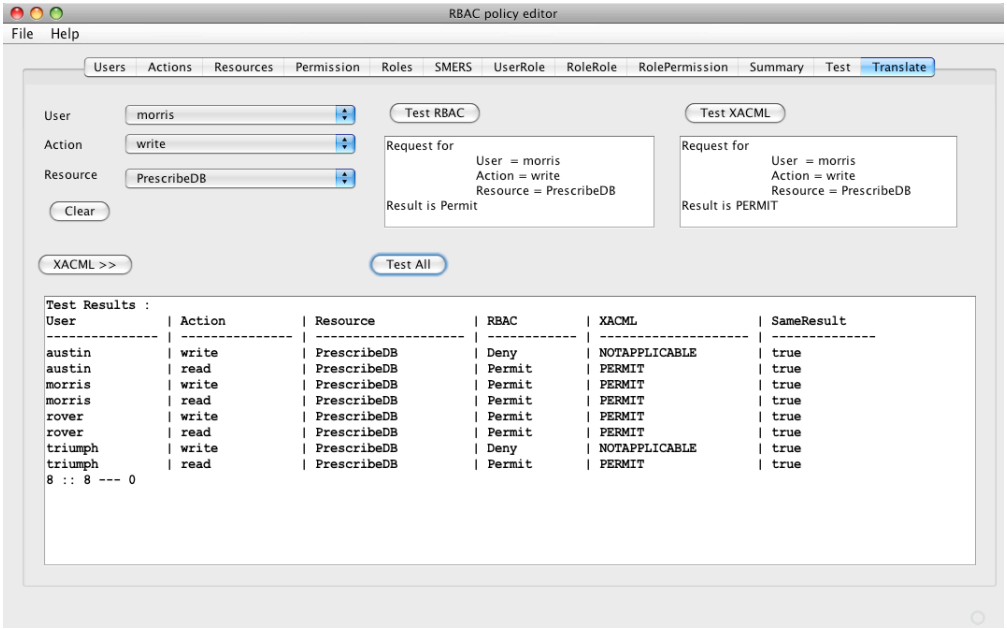

Figure 7.13: RBAC policy translation tool: translate panel showing comparison results

7.3 Constraint checking

This section details how the prototype tool allows an RBAC policy to be checked against a number of constraints, which is only possible because of the definition of the Alloy model of RBAC of Chapter 4. The method of performing this checking is broken into several parts. Firstly, the constraints of interest are added to the Alloy model of RBAC as additional facts. Secondly, the RBAC policy captured by the tool is exported. Finally, the facts chosen by the user are evaluated and the results returned.

Examining this in more detail, the Alloy model of RBAC is extended by adding a number of facts. As an example, a fact may be used to define the constraint which will be breached if any individual user has all possible permissions, either directly or through inheritance. This could be written as a fact called `NobodyCanDoEverything` and affects all hierarchies. Although this could have been added as an additional constraint on the definition of `Hierarchy`, by having it as a separate constraint permits the policy writer to decide if it is of interest to them and allows for it to be tested independently.

```
fact NobodyCanDoEverything {
  all h : Hierarchy |
    all u : h.USERS | u.(h.UA).*(h.RH).(h.PA) != h.PRMS
}
```

If it is found that the fact `NobodyCanDoEverything` does not hold, then the policy tool calls a function called `fun_NobodyCanDoEverything` which reports all the `Hierarchy` and `User` pairs that breach the constraint. By creating facts and functions using this naming convention, the tool can automatically pick up new constraints and report any breaches without prior knowledge

of their existence.

```
fun fun_NobodyCanDoEverything() : Hierarchy -> User {
  { h : Hierarchy, u : User |
    u in h.USERS && u.(h.UA).*(h.RH).(h.PA) = h.PRMS }
}
```

Similar facts such as `NobodyHasAllRoles` and `EverybodyCanDoSomething` are defined in Appendix D. While giving a user all permissions could be considered an error, there are also constraints relating to redundancy in the policy. One example is `NoRedundantPermissions` which restricts a role from being assigned a permission that it already holds due to inheritance.

```
fact NoRedundantPermissions {
  all h : Hierarchy |
    all r : h.ROLES | no (r.(h.PA) & r.^(h.RH).(h.PA))
}

fun fun_NoRedundantPermissions() :
  Hierarchy -> Role -> PRMSBase {
  { h : Hierarchy , r : Role , p : PRMSBase |
    r in h.ROLES && p in h.PRMS &&
    p in (r.(h.PA) & r.^(h.RH).(h.PA)) }
}
```

The facts `AllRolesHaveAPermission` and `AllPermissionsReachable` could also be considered as redundancy constraints and are defined in Appendix D.

To permit the defined policy to be checked using the Alloy Analyzer, the policy first has to be converted into an Alloy instance. There is an XML format for instances which contains elements for signature and field relations, as well as other information such as the command that was used to create the instance, values of quantified variables and details of the model. The application does not create instances in this format directly as it contains significant typing information and requires the explicit calculation of signature and field relations. Instead, the application exports the model in a JavaScript Object Notation (JSON)[1] format in which the fields of each object are described in isolation. Below is a fragment of JSON which defines the roles `Doctor` and `Nurse`.

```
{ id : Doctor, type : Role },
{ id : Nurse, type : Role }
```

The above fragment of JSON is treated as being equivalent to the following Alloy statements, where `one` is used to signify a unique value.

```
one sig Doctor extends Role {}
one sig Nurse extends Role {}
```

[1] http://www.json.org

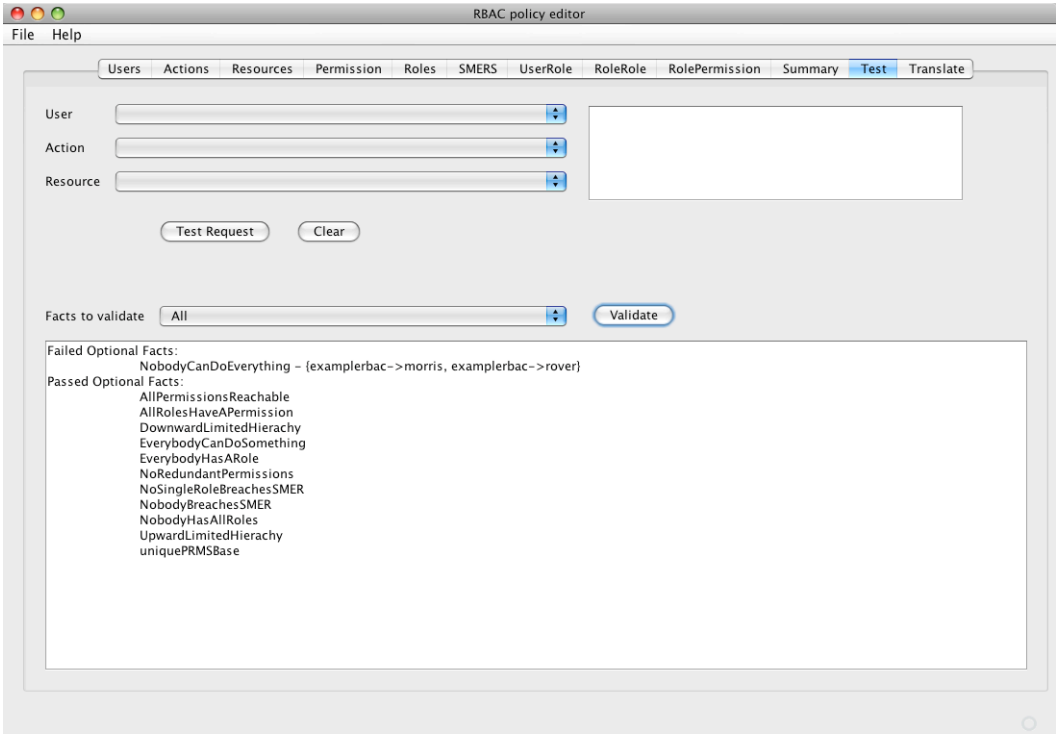

Figure 7.14: Translation tool showing RBAC policy validation

To perform constraint checking, the Alloy model is first parsed using a modified version of the parser used by the Alloy Analyzer. By combining the type information from the Alloy model with the JSON file, it is then possible to calculate the signature and field relations and create a standard XML instance file. The XML instance file is then loaded using standard Alloy API calls to create a solution object that can be used to perform evaluations.

Using the parsed Alloy model the application can list all the facts and evaluate them. It is possible that the instance that has been created is not valid, in which case some of the facts may be false, possibly including facts associated with the signature declarations. Although the Alloy Analyzer would never create such an instance, that does not prevent the libraries from evaluating invalid instances.

If a failed fact has an associated function in the model the application will evaluate it and display the results. This result will indicate the source of the problem; however, if further analysis is still required, it is possible to export the instance as a standard Alloy model populated with `one sig` declarations as above.

In Figure 7.14 the result of constraint checking the policy associated with the running example is shown. It indicates that the users `morris` and `rover` can in fact do everything. This is the expected result for the running example as `morris` and `rover` both have the role `Doctor` which inherits the permissions of `Nurse`. The other facts all pass successfully.

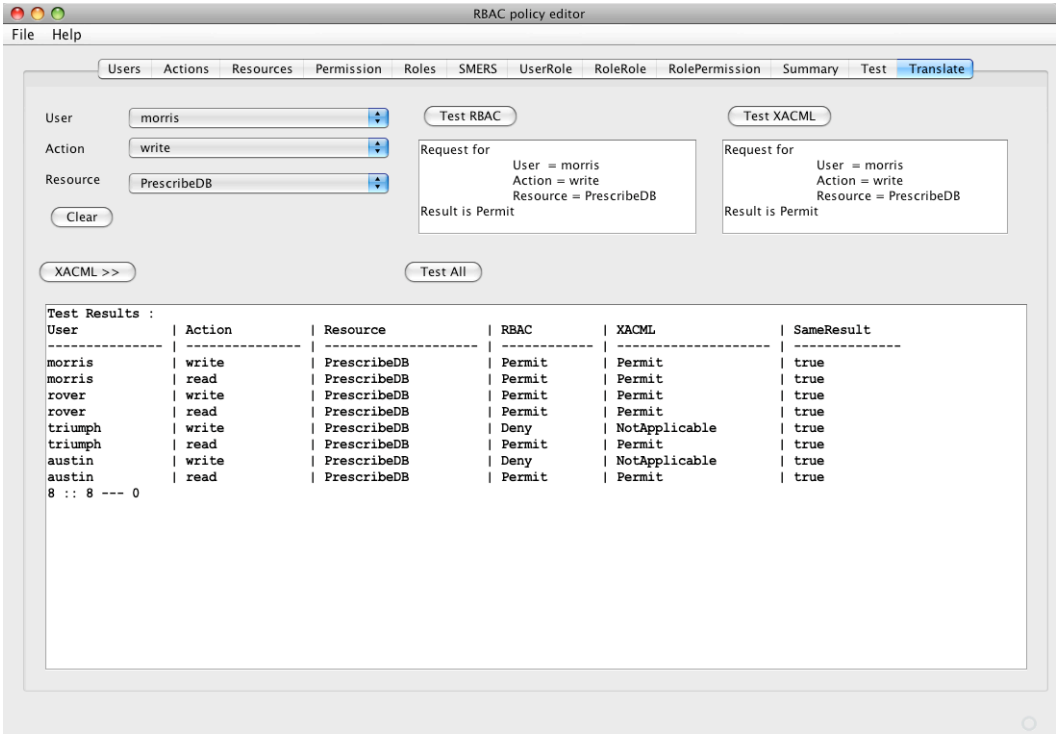

Figure 7.15: Translation tool showing testing of translated policy

7.4 Translation checking

This section describes in more detail the checks that are performed as part of the translate pane, which is illustrated in Figure 7.15. The `Test All` button performs the task of generating requests, evaluating the requests using RBAC and XACML, and comparing the results.

The requests that are generated correspond to every valid combination of user, action and resource from the users, actions and resources captured by the policy. As can be seen in Figure 7.15 this amounts to eight combinations as there are four users, two possible actions and a single resource.

Each of the requests is evaluated against the RBAC version of the policy as well as the XACML version of the policy which resulted from running the translation. For the purpose of evaluating the results, a `NotApplicable` returned by XACML is considered to be a `Deny`. The results of evaluating each request is shown in the table displayed at the bottom of the pane. Figure 7.15 shows this for the policy relating to the running example. The table captures the user, action and resource that made up the request, followed by the RBAC and XACML results and a Summary column stating if the results were in fact the same. The last line at the bottom of the table, `8 :: 8 --- 0`, is a summary of the results. The first number represents the number of requests that were evaluated; the second number reports how many requests returned the same result when evaluated by RBAC and XACML; the third number reports how many requests failed to yield the same result when evaluated by RBAC and XACML. For a successful test to have occurred, the first and second number should be equal and the third number should be zero.

7.5 Testing and validation

The prototype application described in this chapter has been tested internally and externally. The internal testing involved using well-understood RBAC policies, which were then captured using the application and translated into XACML. The application result-comparison functionality was used in two ways. Firstly, it was used to check that the result of each request applied to the original RBAC and the resulting XACML was consistent. Secondly, each of the individual results was checked to ensure that it was the result which was expected from the RBAC policy, this check being possible because the original RBAC policy was well understood. Finally, the resulting XACML was deployed to a sif test node and checked to ensure that the policy permitted access as expected.

The external testing was performed by 18 students who used the application as part of their practical sessions on a Data Security course taught as part of the University of Oxford's Software Engineering Programme. The students were divided into a number of groups, with each group being responsibile for developing access control policies to provide access to their own data source. The scenario they were given described legitimate data accesses that should be permitted to users from other groups. Each of the groups developed RBAC policies to describe the access that was to be permitted to each member of their own group — as well as access afforded to members of collaborating groups. The students used the application to capture RBAC access control policies that they had developed. Once they had captured the RBAC policy, they used the tool to check that the expected result was given for a number of possible requests. Once the students were happy with the RBAC policy they had defined, they translated it to XACML and used the tool to check it for consistency. The students then deployed the XACML policy to a sif environment, where it was subjected to additional testing. This included verifying that requests submitted by each of the groups only permitted access consistent with the defined policy.

In addition to the above, tests the application was used in the case study described in Chapter 8 to translate the RBAC policy supplied into an XACML policy suitable for use by sif.

In addition to the basic testing, a number of simple scaling tests were performed. A number of automatically generated RBAC policies were used to assess how scaleable the application is. The consistency checking performed by Alloy failed when the number of atoms exceeded 1000 (where each atom is either a user, a role, an action, a resource or a permission). This figure is in line with the theoretical maximums that Alloy is designed to deal with, which is approximately $2^{31/n}$ atoms — where n is the largest arity of any relation in the Alloy model. The rest of the tool which captures RBAC and performs the translation to XACML continues to function well beyond this size, and appears to be limited by the memory available to the application. There is however a slow-down in performing the results comparison test as it checks every possible user, action and resource combination. Further consideration about issues of scaleability are considered in Section 9.2.2.

7.6 Summary

This chapter has described a prototype application which permits the capturing of RBAC policies which can then be converted into equivalent XACML policies. In addition to allowing the

translation of policies between representations, it also permits additional checks to be made to enhance the users' confidence in the resulting policy.

The addition of the constraint checking functionality illustrates one benefit of having an Alloy model of RBAC: the ability to add new constraints to the model allows additional checks to be placed on the access control policies appropriate to the environment of the user developing the policy.

The prototype application represents an initial step towards fulfilling some of the use cases of Chapter 3. For example, if the use case *Use existing access control application and leverage new technologies that use a different access control paradigm* is considered, the goal is achieved by having the ability to translate between access control representations. In the case of the application presented in this chapter, this principle is illustrated by demonstrating the ability to translate an RBAC representation of a policy into an XACML representation. Additionally, the application also provides the facility to test that the resulting XACML policy is equivalent to the RBAC policy by checking that the two versions of the policy return the same results to access control requests.

Chapter 8

Case study

This chapter presents a case study as a means of illustrating and validating the work presented in this thesis. The case study is based on the needs of a research group associated with a multi-disciplinary collaborative project who required distributed access to various data sources. The access control policies for the data sources were defined in RBAC and required translating into XACML to enable the deployment of the policy to the middleware chosen to facilitate the sharing of the data. The original RBAC policy was captured using the prototype application of Chapter 7. Subsequent to the capture of the RBAC policy, the application was then used to generate an equivalent XACML policy which was deployed to the middleware. In addition, the generated XACML was tested with numerous requests to verify that the results were consistent with the original RBAC policy. The rest of this chapter provides some background to the case study along with a description of the access control requirements. An illustrative section of the RBAC policy is then considered with respect to the formal definition of the translation process of Chapter 6. The formal representation is further considered alongside the XACML generated by the prototype application of Chapter 7 to illustrate the correspondence between the formal representation and the XACML representation of the policy. The final part of the chapter shows some sample outputs generated by the prototype application relating to the validation of the translation process.

8.1 Background

The case study described in this chapter is the result of work undertaken during the course of the GIMI project [91, 93]. The focus of GIMI was the development of distributed, service-oriented infrastructures and applications to support a wide variety of healthcare research, training and delivery activities.

The middleware developed during GIMI, to provide the necessary services, was the sif framework described in Section 2.7.3. The sif middleware provides the facility to share resources between interested parties in a secure way, with the resource owner defining the policy to control access to their own resource. In the case of sif, the access control framework deployed only supported access control policies defined using XACML.

The *plugin* mechanism of the sif middleware provides a standardised way of facilitating access

to resources by having a well-defined interface for each plugin type. One or more plugins of the appropriate type are then deployed for each resource that the sif middleware is to provide access to. The ability for any given user to interact with a particular plugin is governed by the deployed access control policy written in XACML.

For example, consider a user *usr* who wishes to perform an action *act* on a resource *res*, with the resource being accessed via a plugin. The application that *usr* is using initiates communication with the sif middleware requesting that *act* is performed on *res* on behalf of *usr*. The sif middleware then formulates an XACML request — based on *usr*, *act* and *res* — which is then processed by the XACML PDP, with the request being evaluated in the context of the currently active XACML policy. Based on the evaluation of the request, *usr* is either granted or denied permission to perform *act* on *res*.

Although there were several application development teams within the GIMI project, this case study is based exclusively on work carried out by the long term conditions group, who were concerned with various research projects related to two distinct medical conditions: Diabetes and Asthma.

The Diabetes research focused on research into how effectively patients could self-manage their condition, and involved the collection of regular measurements of glucose levels via mobile telephone. One aspect of the research concerned the validation of algorithms that predicted trends in glucose levels, which, if validated, could provide useful early warning to a patient if their levels were moving outside the recommended range. The other aspect was to evaluate the effectiveness of different methods of monitoring self-management.

The Asthma research involved gathering peak expiratory flow data (PEF) via mobile devices.[1] Regular PEF measurements are used to monitor the progress of patients suffering from Asthma, as it is essentially a measure of the effectiveness of a person's lungs, with the reading being indicative of the degree of obstruction in the patient's airways. The primary focus of this research was to ascertain if a correlation existed between the prevailing weather conditions and the likelihood of having a bad asthmatic episode. This involved the joining of PEF data with data provided by the Meteorological Office.

The case study of this chapter is concerned with the need of the Diabetes group to enable external access to data collected about diabetic patients as part of two studies. A separate plugin was developed for each data source they wished to share. Alongside the plugins, an RBAC description of the access control they wished to apply to the data sources was also supplied. The members of the Diabetes group indicated that they had no desire to learn XACML for the sole purpose of developing an equivalent access control policy in an alternate access control policy description paradigm, even though it was required by the sif middleware. It was therefore proposed that this access control scenario would be used as a test for the RBAC-to-XACML translation application.

Figure 8.1 provides an overview of the scenario the case study is based on. A number of users wish to request access to a number of resources via a sif node. The users are shown on the left of Figure 8.1 and the resources on the right. To illustrate this, consider *user*1 wishing to access data. The application they use initiates a request to the interface provided by the sif node, illustrated by line 1. This request is translated into an access control request and passed

[1] PEF is a measure of a person's maximum speed of expiration and is usually measured with a peak flow meter.

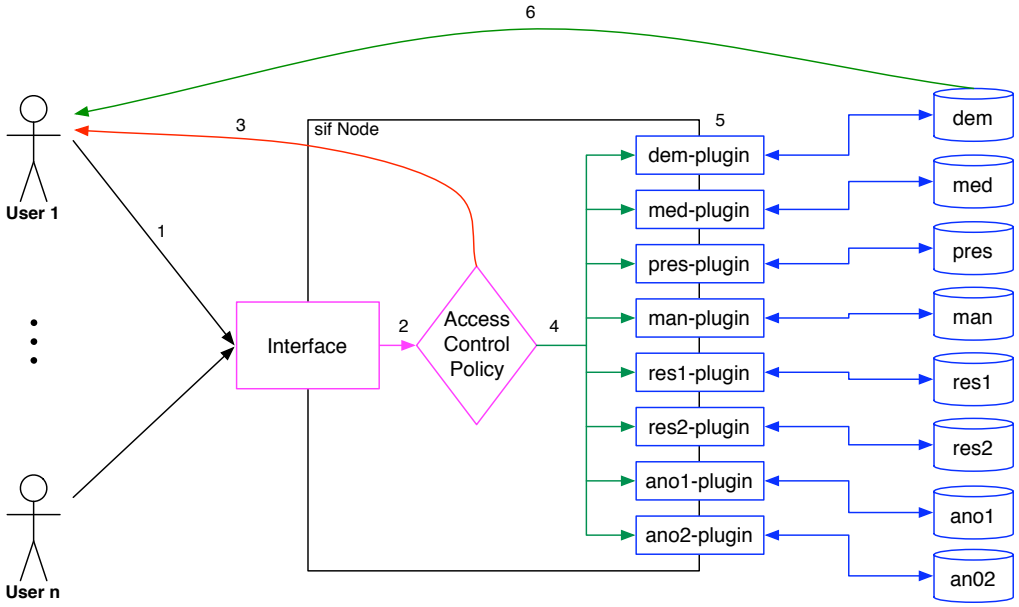

Figure 8.1: Outline of the case study

on — line 2 — to be evaluated by the access control policy. If the access control policy in place rejects the request then the user is informed accordingly (line 3). If the request is approved then the request is passed on to the relevant plugin, in this case via $dem - plugin$ (line 4). The plugin, 5, processes the request and the resulting data is returned to the user, illustrated by line 6.

8.2 Requirements

To define the overall access control policy it is first necessary to define the resources being accessed and the actions that can be performed on each resource, the combination of which will then make up the permissions. It is then necessary to define the different roles to which the permissions are assigned as well as how the roles relate to each other in terms of inheritance. Finally the individual users need to be associated with the roles that are necessary to permit the user to perform their job function.

The resources associated with the case study are all plugins that access data sources. The following give the names of the plugins and a description of the data they expose:

- *dem*: contains basic demographic data pertaining to the patients
- *med*: contains medical case notes detailing each interaction with the patient and the results of tests
- *pres*: contains details about any medication that is prescribed to the patient
- *res1*: research data relating to research project 1
- *res2*: research data relating to research project 2

- *ano1*: anonymised data relating to project 1 for statistical analysis
- *ano2*: anonymised data relating to project 2 for statistical analysis
- *man*: management data containing information necessary for the smooth running of the department and projects

The actions that can be performed on the resources described above are constrained by the fact that each of the resources represents a data source. The actions that are permitted on data sources are:

- *read*: permits the reading of a data source
- *write*: permits the writing to a data source

The roles which will be used to construct the overall access control policy are now defined with the pertinent information relating to each of the roles being shown in Table 8.1. Each row gives the name of a role, a short description about the role, a list of the permissions that are directly assigned to the role (if any), and any roles that are inherited by the role.

Role	Description	Permissions	Inherits
secretary	Checks the patients in and validates that the demographic data is correct, updating the data if necessary.	(*read*, *dem*)	
nurse	Records basic medical details along with the results of tests performed by the patient and dispenses prescribed medication as appropriate.	(*read*, *med*), (*write*, *med*), (*read*, *pres*)	*secretary*
doctor	Performs examinations and records the findings. Prescribes any medication required.	(*write*, *pres*)	*nurse*
doctor1	Performs additional tests relating to project 1 and records findings in project database.	(*read*, *res1*), (*write*, *res1*)	*doctor*
doctor2	Performs additional tests relating to project 2 and records findings in project database.	(*read*, *res2*), (*write*, *res2*)	*doctor*
consultant	Monitors all projects and management data.	(*read*, *man*)	*doctor*, *doctor1*, *doctor2*
adm	Maintains overall management data for the organisation and projects.	(*read*, *man*), (*write*, *man*)	*secretary*
stat1	Performs statistical analysis on the anonymous data relating to project 1.	(*read*, *ano1*)	
stat2	Performs statistical analysis on the anonymous data relating to project 2.	(*read*, *ano2*)	
res1	Generates and reviews data for project 1. Additionally, generates the anonymous data for use by the statisticians.	(*write*, *ano1*), (*read*, *res1*), (*write*, *res1*)	*stat1*
res2	Generates and reviews data for project 2. Additionally, generates the anonymous data for use by the statisticians.	(*write*, *ano2*), (*read*, *res2*), (*write*, *res2*)	*stat2*
snrres	Manages all projects.		*res1*, *res2*

Table 8.1: Case study roles

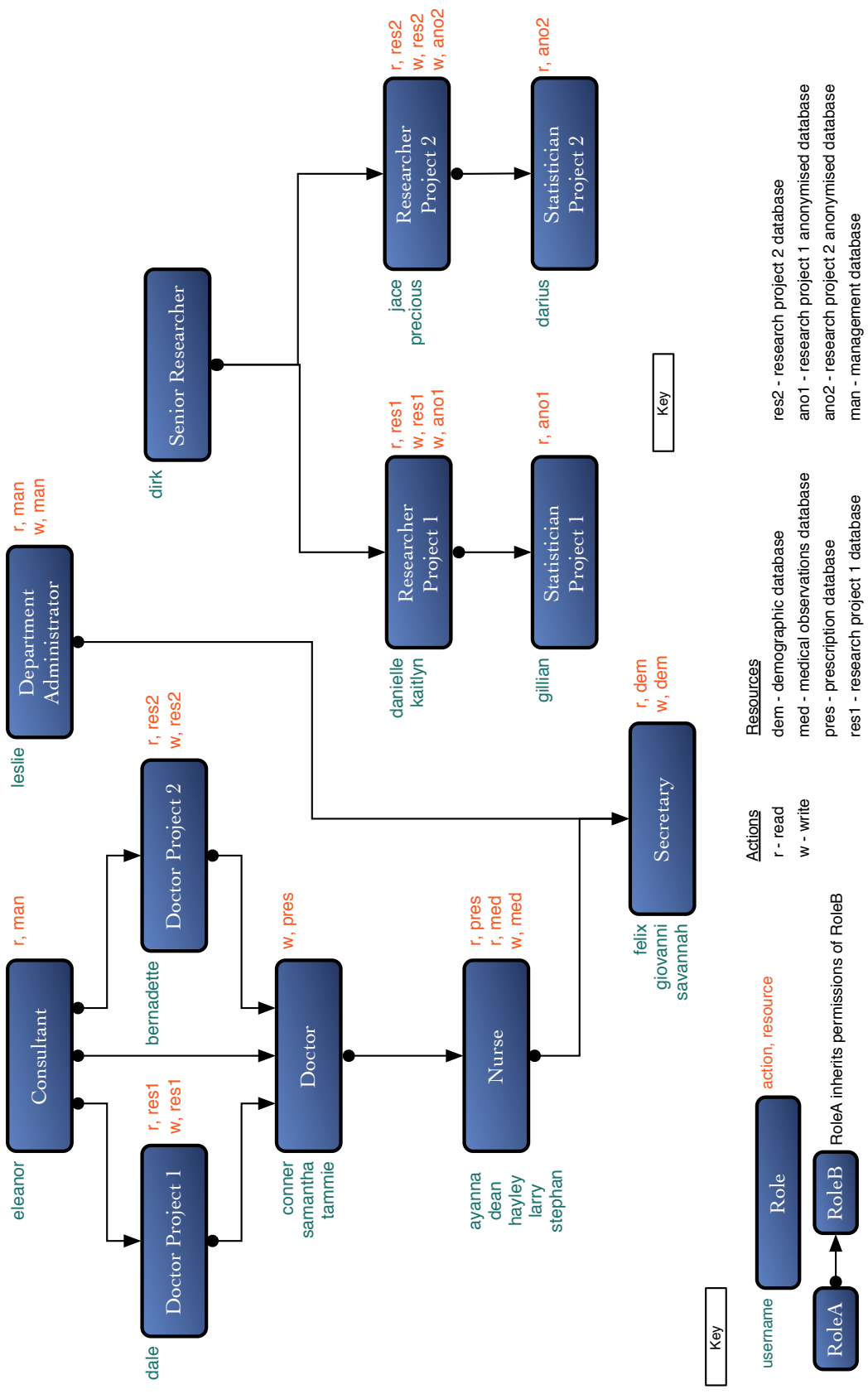

Figure 8.2: Representation of the RBAC policy of the case study

To permit the generation of the access control policy, a list of users who will be associated with the roles needs to be provided. The users related to the case study have had their names changed, with the following being used: *gillian*, *bernadette*, *savannah*, *stephan*, *eleanor*, *hayley*, *darius*, *tammie*, *dean*, *conner*, *ayanna*, *danielle*, *samantha*, *dale*, *larry*, *dirk*, *felix*, *jace*, *giovanni*, *leslie*, *precious*, *kaitlyn*.

Finally, the relationship between users and roles is given as the following: *samantha* has the role *dr*, *dale* has the role *dr*1, *ayanna* has the role *nu*, *tammie* has the role dr, *darius* has the role *stat*2, *stephan* has the role *nu*, *dirk* has the role *sres*, *conner* has the role *dr*, *bernadette* has the role *dr*2, *dean* has the role *nu*, *giovanni* has the role *sec*, *felix* has the role *sec*, *precious* has the role *resp*2, *savannah* has the role *sec*, *leslie* has the role *adm*, *jace* has the role *resp*2, *larry* has the role *nu*, *gillian* has the role *stat*1, *hayley* has the role *nu*, *danielle* has the role *resp*1, *kaitlyn* has the role *resp*1, and *eleanor* has the role *cons*.

The full policy including the role hierarchy and permission assignments is illustrated in Figure 8.2. This representation is a useful aid in visualising the overall shape of the access control policy and in this case illustrates the two distinct hierarchies — with one relating to the medical personal and the other the research staff. It is also possible to notice that the same permissions occur in both sub-hierarchies, which is not excluded by the definition of RBAC and can be useful when deciding on hierarchies to represent complex requirements. An example of the same permission occurring in both sub-hierarchies is the permission (*read*, *res*1), which is associated with the role *DoctorProject*1 and the role *ResearcherProject*1.

The role names used in Figure 8.2 are moderately long and have been abbreviated when being used to capture the access control policy with the application of Chapter 7 as well as when producing the formal representation of the policy. For instance, the role *DoctorProject*1 in Figure 8.2 will become *dr*1 when used in the application or formal representation. All of the abbreviations follow a similar pattern and each role is easily reconcilable between the representations.

The next section illustrates the use of the access control policy application of Chapter 7 to capture the basic RBAC policy.

8.3 Policy capture using application

First, the component parts of the RBAC policy need to be entered using the appropriate panes, with Figure 8.3 illustrating the user pane. Once the RBAC policy has been entered, a number of checks could be performed.

For example, the summary pane is used to ensure that the policy has been entered correctly and each user has been associated with the roles and permissions expected; Figure 8.4 illustrates a summary for *conner*, with the roles and permissions associated with the user *conner* being shown. After checking the summary, a number of additional tests are performed to check that the expected access control decision is given for various requests, as well as using Alloy to evaluate if the policy conformed to the additional constraints of interest. A simple request of *gillian* asking to *read* the resource *ano*1 is illustrated in the top half of Figure 8.5, with the bottom half of Figure 8.5 illustrating the use of Alloy to check additional properties of the entered RBAC access control policy. In the case of Figure 8.5, the constraints being tested are concerned with the ensuring that the RBAC policy has no additional parts that are not necessary and no one user

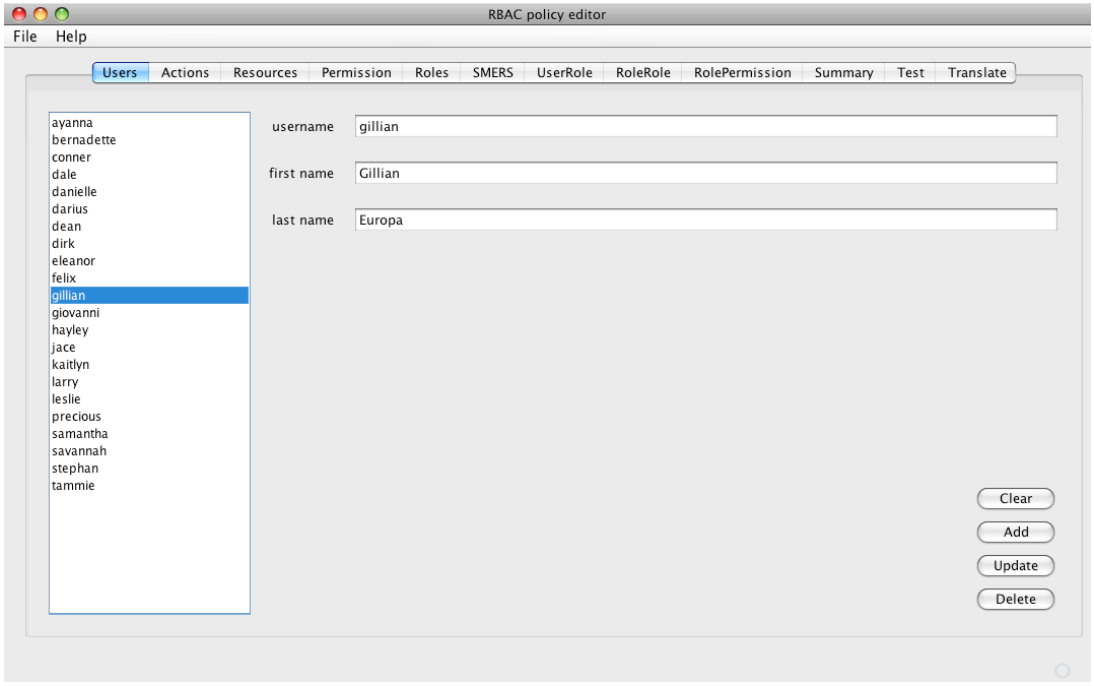

Figure 8.3: Entering the users for the RBAC policy

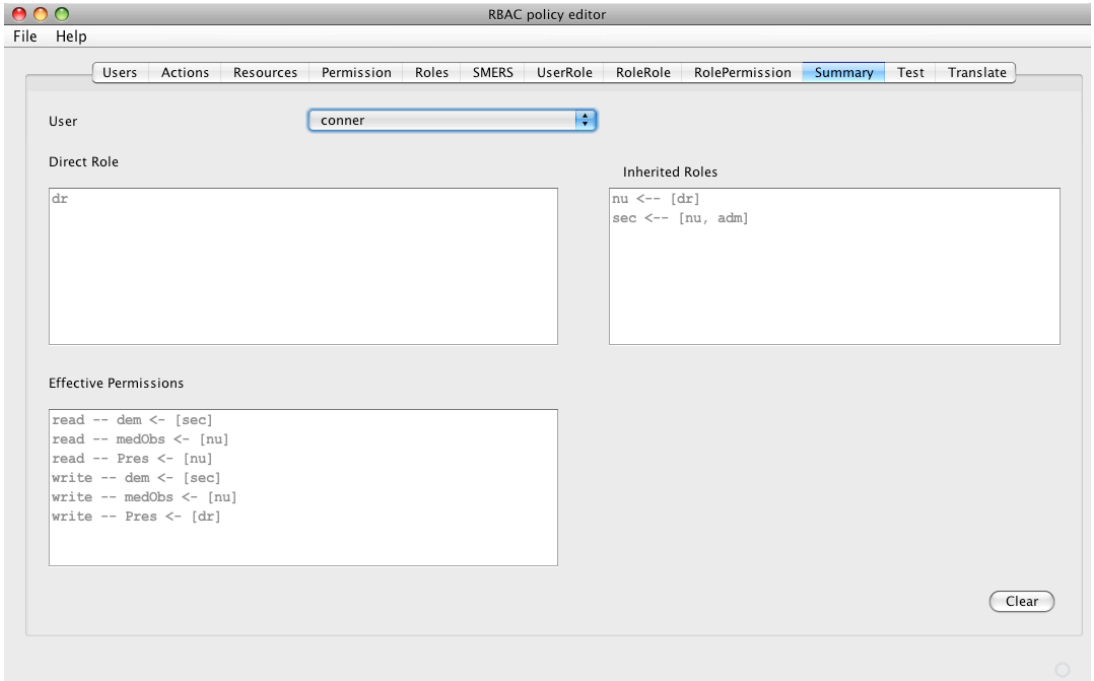

Figure 8.4: Summary of RBAC roles and permissions for Conner

has been given permission to do everything.

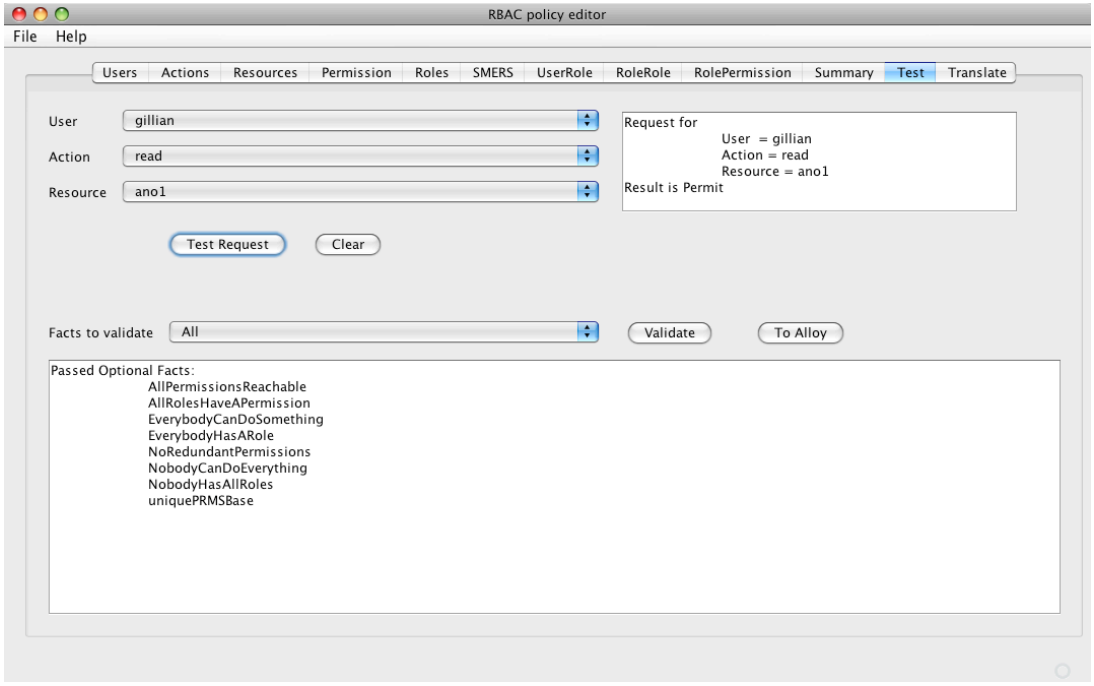

Figure 8.5: Checking a request and conformance to the Alloy model

8.4 Formalisation and transformation of the RBAC policy

Next, the formal representation of the access control policy is considered, firstly represented in terms of Z and secondly in terms of Alloy. The Alloy representation is automatically generated by the application as described above.

8.4.1 Z representation of RBAC

The following is a Z representation of the required RBAC access control policy.

$$
\begin{array}{|l}
rbach : Hierarchy \\
\hline
rbach.UA = \{(samantha, dr), (dale, dr1), (ayanna, nu), (tammie, dr), (darius, stat2), (stephan, nu), \\
\qquad (dirk, sres), (conner, dr), (bernadette, dr2), (dean, nu), (giovanni, sec), (felix, sec), \\
\qquad (precious, resp2), (savannah, sec), (leslie, adm), (jace, resp2), (larry, nu), (gillian, stat1), \\
\qquad (hayley, nu), (danielle, resp1), (kaitlyn, resp1), (eleanor, cons)\} \\
rbach.PA = \{(resp1, (read, res1)), (stat2, (read, ano2)), (adm, (read, man)), (dr2, (write, res2)), \\
\qquad (dr2, (read, res2)), (resp2, (read, res2)), (stat1, (read, ano1)), (sec, (read, dem)), \\
\qquad (cons, (read, man)), (nu, (read, medObs)), (sec, (write, dem)), (nu, (write, medObs)), \\
\qquad (dr, (write, pres)), (nu, (read, pres)), (dr1, (read, res1)), (adm, (write, man)), \\
\qquad (resp1, (write, res1)), (resp2, (write, res2)), (resp2, (write, ano2)), (resp1, (write, ano1)), \\
\qquad (dr1, (write, res1))\} \\
rbach.RH = \{(dr1, dr), (sres, resp2), (cons, dr1), (cons, dr), (nu, sec), (resp1, stat1), (resp2, stat2), \\
\qquad (dr2, dr), (adm, sec), (cons, dr2), (dr, nu), (sres, resp1)\} \\
rbach.USERS = \{gillian, bernadette, savannah, stephan, eleanor, hayley, darius, tammie, dean, conner, \\
\qquad\quad ayanna, danielle, samantha, dale, larry, dirk, felix, jace, giovanni, leslie, precious, \\
\qquad\quad kaitlyn\} \\
rbach.ROLES = \{dr2, dr, cons, nu, dr1, sres, resp2, stat1, sec, adm, stat2, resp1\} \\
rbach.PRMS = \{(write, ano1), (read, res2), (read, medObs), (read, ano2), (write, dem), (read, ano1), \\
\qquad\quad (write, res1), (read, res1), (read, man), (write, man), (read, dem), (write, medObs), \\
\qquad\quad (write, res2), (read, pres), (write, ano2), (write, pres)\}
\end{array}
$$

8.4.2 Alloy representation of RBAC

The following is a fragment of the Alloy representation generated by the application as described in the previous section, with the full Alloy model presented in Appendix E.2.

```
one sig examplerbac extends Hierarchy {}
{
  UA = dean -> nu + dale -> dr1 + larry -> nu + gillian -> stat1 + precious -> resp2 + leslie -> adm +
       danielle -> resp1 + dirk -> sres + felix -> sec + giovanni -> sec + savannah -> sec + tammie -> dr +
       bernadette -> dr2 + samantha -> dr + stephan -> nu + conner -> dr + eleanor -> cons + ayanna -> nu +
       kaitlyn -> resp1 + hayley -> nu + jace -> resp2 + darius -> stat2
  RH = dr -> nu + resp1 -> stat1 + sres -> resp2 + sres -> resp1 + adm -> sec + resp2 -> stat2 + dr2 -> dr +
       nu -> sec + cons -> dr2 + cons -> dr + cons -> dr1 + dr1 -> dr
  PA = resp2 -> rRes2 + resp2 -> wRes2 + sec -> rDem + dr2 -> rRes2 + resp1 -> rRes1 + dr -> wPres + dr1 -> wRes1 +
       resp1 -> wRes1 + nu -> rMedObs + dr2 -> wRes2 + adm -> wMan + nu -> rPres + cons -> rMan + nu -> wMedObs +
       stat2 -> rAno2 + resp2 -> wAno2 + dr1 -> rRes1 + adm -> rMan + sec -> wDem + resp1 -> wAno1 + stat1 -> rAno1
  USERS = samantha + dale + eleanor + jace + dirk + savannah + giovanni + felix + gillian + tammie + larry +
          kaitlyn + ayanna + danielle + dean + stephan + darius + bernadette + conner + leslie + precious +
          hayley
  ROLES = adm + nu + sec + cons + dr1 + dr2 + stat1 + stat2 + sres + resp2 + resp1 + dr
  PRMS = wRes1 + wAno1 + wAno2 + rAno2 + rAno1 + rRes1 + rPres + rRes2 + wRes2 + wPres + wMan + rMedObs + rMan +
         rDem + wMedObs + wDem
}
```

There is a strong correspondence between the Z and Alloy models of the RBAC policy. For example, the Z relation $rbac.UA$ is equivalent to the Alloy relation `UA` with the elements $(samantha, dr)$ representing the same information as `samantha -> dr`.

Each of the permissions captured in `PRMS` and used in the relations in `PA` are expanded in Appendix E.2. For example, the permission `rAno1` used in the element `stat1 -> rAno1` of the relation `PA` is fully defined as:

```
one sig rAno1 extends PRMSBase {}
{ resource = ano1 && action = read }
```

This description of `rAno1` means that the relation `stat1 -> rAno1` corresponds to the element $(stat1, (read, ano1))$ of the *rbach.PA* relation in the Z representation, with $(read, ano1)$ corresponding to the permission `rAno1`.

8.4.3 Z translation of RBAC to XACML

Next, the Z-based formal translation process of Chapter 6 is used to translate the access control policy defined above. For the purpose of this discussion, only the portion of the RBAC policy relating to the role $stat1$ will be considered in detail. The results of the formal translation of parts of the policy relating to $stat1$ is considered below, along with the formal representations relationship to the XACML produced by the translation application.

To this end, the following is a Z representation of the elements of the RBAC policy which relate to the role $stat1$.

$$\begin{aligned} UA \rhd \{stat1\} &= \{(gillian, stat1)\} \\ \{stat1\} \lhd RH^+ &= \emptyset \\ \{stat1\} \lhd PA &= \{(stat1, (read, ano1))\} \end{aligned}$$

First, the policy set that captures the relationship between users and the role $stat1$ is considered. From the above it can be seen that the only user that is directly related to the role $stat1$ in the UA relation is *gillian*. Given this, the identifier of the policy set relating users to the role $stat1$ and the identifier of the target of the policy set is given by:

$$\begin{aligned} &genPSidru\,(stat1, \{gillian\}) \mapsto rupsStat1 \in getPolicySet' \\ &genTidPSru\,(stat1, \{gillian\}) \mapsto tpsrustat1 \in getTarget' \end{aligned}$$

The policy set $rupsStat1$ is captured by the following:

$$\begin{aligned} rupsStat1 = \langle\!| &psid \rightsquigarrow genPSidru\,(stat1, \{gillian\}), target \rightsquigarrow genTidPSru\,(stat1, \{gillian\}), \\ &pca \rightsquigarrow polPermitOverride, inPolSet \rightsquigarrow \langle PolSet\,(genPSidrr\,(stat1))\rangle, obli \rightsquigarrow \emptyset |\!\rangle \end{aligned}$$

Figure 8.6 illustrates the fragment of the XACML policy, generated by the translation application, which corresponds to the RBAC role $stat1$. This makes it possible to compare the formal representation of translation process with the actual XACML which results from the translation application.

The policy set identifier, *psid*, which in the formal representation has the value $genPSidru\,(stat1, \{gillian\})$ is given the value `PolicySetId="RU:stat1"` on line 2 of Figure 8.6. Next, the *target* of $genTidPSru\,(stat1, \{gillian\})$ corresponds to the target block between lines 4 and 15, and is discussed in more detail below. The policy-combining algorithm, *pca*, captured formally as *polPermitOverride* corresponds to the entry on line 3:

```
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides".
```

```
01 <PolicySet
02   PolicySetId="RU:stat1"
03   PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">
04   <Target>
05     <Subjects>
06       <Subject>
07         <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
08           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">gillian</AttributeValue>
09           <SubjectAttributeDesignator
10             DataType="http://www.w3.org/2001/XMLSchema#string"
11             AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
12         </SubjectMatch>
13       </Subject>
14     </Subjects>
15   </Target>
16   <PolicySet ...
17     PolicySetId="RR:stat1"
18     PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">
19     <Target/>
20     <Policy ...
21       PolicyId="RP:stat1:read.ano1"
22       RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
23       <Target/>
24       <Rule RuleId="R:read.ano1" Effect="Permit">
25         <Target>
26           <Actions>
27             <Action>
28               <ActionMatch
29                 MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
30                 <AttributeValue
31                   DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
32                 <ActionAttributeDesignator
33                   DataType="http://www.w3.org/2001/XMLSchema#string"
34                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
35                 />
36                </ActionMatch>
37             </Action>
38           </Actions>
39           <Resources>
40             <Resource>
41               <ResourceMatch
42                 MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
43                 <AttributeValue
44                   DataType="http://www.w3.org/2001/XMLSchema#string">ano1</AttributeValue>
45                 <ResourceAttributeDesignator
46                   DataType="http://www.w3.org/2001/XMLSchema#string"
47                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
48                 />
49               </ResourceMatch>
50             </Resource>
51           </Resources>
52         </Target>
53       </Rule>
54     </Policy>
55   </PolicySet>
56 </PolicySet>
```

Figure 8.6: Translation application XACML fragment relating to role $stat1$

The contents of *inPolSet*, which is the policy set identifier of $PolSet\,(genPSidrr\,(stat1))$, is captured in Figure 8.6 between lines 15 and 55; further consideration of this policy set is given below. Finally as the obligation, *obli* is an empty set it is omitted from the XACML.

Next the target with the *targetid* of $genTidPSru\,(stat1, \{gillian\})$ is considered in more detail.

The Z representation of this target is shown below, which, as stated above, corresponds to lines 4 – 15 of Figure 8.6.

$$tpsrustat1 = \langle\!| tid \rightsquigarrow genTidPSru\,(stat1, \{gillian\}), sub \rightsquigarrow \langle targetElementEquals\ gillian\rangle, \\ act \rightsquigarrow \langle\rangle, res \rightsquigarrow \langle\rangle, env \rightsquigarrow \langle\rangle |\!\rangle$$

The subject captured by *sub* corresponds to the block `Subjects` between lines 4 and 14. This block indicates that the target will match a subject portion of a request with the value `gillian` which is captured in the XACML on line 8. The formal representation of this is given by *targetElementEquals gillian*, which represents a function that only returns the value *TRUE* if the input is *gillian*, thus producing a *sub* that only matches with the subject *gillian*. The empty sequences associated with *act*, *res* and *env* indicate that they match any value in the respective part of a request. The sections are also omitted from the XACML policy.

Next, the policy set that relates entirely to the *stat* role, which was referenced in the above *inpolset*, is shown below.

$$genPSidrr\,(stat1) \mapsto rrpsStat1 \in getPolicySet'$$

$$rrpsStat1 = \langle\!| psid \rightsquigarrow genPSidrr\,(stat1), target \rightsquigarrow empty\_target\_id, \\ pca \rightsquigarrow polPermitOverride, inPolSet \rightsquigarrow \langle Pol\,(genPid\,(stat1))\rangle, obli \rightsquigarrow \emptyset |\!\rangle$$

This policy set corresponds to lines 15 – 55 of Figure 8.6. For the remainder of this discussion, the identifiers are ignored as they are just concrete values resulting from function application. The main points of interest are that the policy references *empty_target_id* as the *target*, which is represented on line 19 of the XACML as `<Target/>`. The combining algorithm of *polPermitOverride* is captured in the XACML as above on line 18. The obligation is again an empty set, which is omitted from the XACML.

Finally, the policy referenced by $Pol\,(genPid\,(stat1))$ in *inpolset* represents lines 20 – 54 in the XACML and is considered next.

The policy with the identifier of $genPid\,(stat1)$ is shown below.

$$genPid\,(stat1) \mapsto policyStat1 \in getPolicy'$$

$$policystat1 = \langle\!| pid \rightsquigarrow genPid\,(stat1), tid \rightsquigarrow empty\_target\_id, \\ inPol \rightsquigarrow \langle genRid\,(read, ano1)\rangle, rca = rulPermitOverride, obli = \emptyset |\!\rangle$$

Again, the *empty_target_id* is translated to `<Target/>`, which, in this case, is on line 23 of the XACML. The rule-combining algorithm is represented in a similar way to the policy-combining algorithms above and in this case is captured on line 22. Finally, the rule with the identifier $genRid\,(read, ano1)$ referenced in *inPol* represents lines 24 to 53 in the XACML and is considered below.

The rule with the identifier $genRid\,(read, ano1)$ is represented by:

$$genRid\,(read, ano1) \mapsto ruleReadAno1 \in getRule' \\ ruleRead = \langle\!| rid \rightsquigarrow genRid\,(read, ano1), target \rightsquigarrow genTidR(read, ano1), \\ condition \rightsquigarrow \langle\rangle, effect \rightsquigarrow Permit |\!\rangle$$

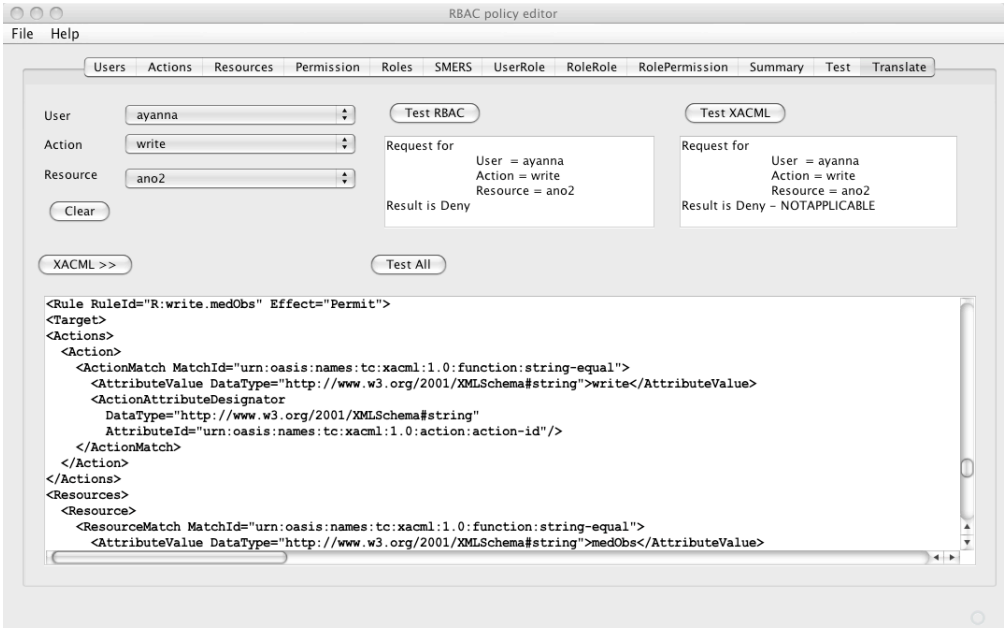

Figure 8.7: XACML resulting from translation and testing of the conversion

The rule defines an *effect* of *Permit* captured as part of line 24 and an empty condition which is omitted from the XACML as it always returns true. The *target* of the rule points at the target with an identifier of $genTidR(read, ano1)$. The formal definition of the target is shown below with the XACML representation being captured in lines 25 – 52.

$$genTidR\,(read, ano1) \mapsto trReadStat1 \in getTarget'$$

$$\begin{aligned} trReadStat1 = \langle\!|\, & tid \rightsquigarrow genTidR\,(read, ano1), sub \rightsquigarrow \langle\rangle, \\ & act \rightsquigarrow \langle targetElementEquals\ read\rangle, \\ & res \rightsquigarrow \langle targetElementEquals\ ano1\rangle, env \rightsquigarrow \langle\rangle \,|\!\rangle \end{aligned}$$

The *act* being set to match an action of *read* is captured in lines 26 – 38 of the XACML, with the matching value of `read` appearing on line 31. In a similar way, *res* being set to match $ano1$ is captured in the XACML between lines 39 and 51 with the value to match of `ano1` appearing on line 44. The target of the XACML omits subject and environment blocks as *sub* and *env* are set to empty sequences.

The remaining parts of the formal model can also be mapped onto XACML in a similar way; however, the resulting XACML is many tens of pages in length, again illustrating the verbosity of XACML.

8.5 Application translation

With the RBAC policy captured, it becomes possible to perform the translation into XACML. Figure 8.7 illustrates a portion of the XACML that results from the translation process as well as the results of applying a request to the RBAC and XACML representations. The results of

the evaluations are consistent as the response to the request is *Deny* in the case of the RBAC evaluation and *NotApplicable*, which is treated as *Deny* in the case of XACML.

A fragment of the resulting XACML policy which represents the role *stat*1 is included as Figure 8.6. The policy set representing the role is applicable to *gillian* and has the permission that allows the *read* action to be performed on the *ano*1 resource.

A final check that the XACML policy that resulted from the translation is consistent with the original RBAC policy is to utilise the 'test-all' facility. This feature generates a set of requests that represent every possible combination of the users, actions and resources defined in the RBAC policy. Each of the resulting requests is then evaluated by the RBAC and XACML representations of the policy. The following is a portion of the output which resulted from the case study. The first three columns effectively represent a request, and the next two columns give the result of applying the request to RBAC and XACML respectively. The final column indicates if the result from the two evaluations is consistent. The last line of the output is a summary of how many requests were applied, as well as the number that returned the same result and the number of errors. For the 'test-all' evaluation of the case study, the result was $Total = 352 :: SameResult = 352 - - - Errors = 0$, which indicates that the XACML generated is consistent with the original RBAC.

The data owners were happy with the resulting policy and felt confident that the testing and formal underpinning of our approach was sufficient to assure them that the final policy conformed to their needs.

```
Test Results :
User          | Action        | Resource        | RBAC         | XACML                | SameResult
------------- | ------------- | --------------- | ------------ | -------------------- | ----------
tammie        | write         | Pres            | Permit       | Permit               | true
tammie        | write         | man             | Deny         | NotApplicable        | true
tammie        | write         | res2            | Deny         | NotApplicable        | true
tammie        | write         | ano1            | Deny         | NotApplicable        | true
tammie        | write         | dem             | Permit       | Permit               | true
.
.
.
gillian       | read          | ano2            | Deny         | NotApplicable        | true
bernadette    | write         | Pres            | Permit       | Permit               | true
bernadette    | write         | man             | Deny         | NotApplicable        | true
bernadette    | write         | res2            | Permit       | Permit               | true
bernadette    | write         | ano1            | Deny         | NotApplicable        | true
bernadette    | write         | dem             | Permit       | Permit               | true
bernadette    | write         | res1            | Deny         | NotApplicable        | true
.
.
.
precious      | read          | dem             | Deny         | NotApplicable        | true
precious      | read          | res1            | Deny         | NotApplicable        | true
precious      | read          | medObs          | Deny         | NotApplicable        | true
precious      | read          | ano2            | Permit       | Permit               | true
Total = 352 :: Same Result = 352 --- Errors = 0
```

8.6 Summary

This chapter describes a case study which has been used to validate the work presented in this thesis. The first part of the chapter describes the general background to the case study, where the long term conditions group wished to utilise the sif middleware to provide access to several data sources to facilitate research. This is followed by a description of the access control requirements needed by the Diabetes group. The access control policy supplied was defined in terms of RBAC and therefore required translating into XACML to facilitate their use by the access control mechanism provided by the sif middleware.

The required RBAC policy is then captured both formally and by using the application described in Chapter 7. A section of the policy is translated formally and the correspondence between the formal representation and the XACML representation is considered. The rest of the chapter summarises the use of the prototype application of Chapter 7 to perform the translation to XACML and perform some tests an the resulting policy.

Although the example used in this case study is relatively simple, it does provide a useful validation of the approach described in this thesis. The application successfully translated the access control policy from the RBAC into (the very verbose) XACML. The resulting policy was utilised by the sif middleware to enable decisions to be rendered for access requests to the resources being protected.

Chapter 9

Discussion

This chapter summarises the contributions of thesis, comments on the weaknesses and limitations of the various techniques employed, and outlines possible avenues of further work.

9.1 Contribution

9.1.1 Research question

The main contribution of this work is to define the initial steps towards a generalised formal mechanism for translation between access control representations. This work was guided by the requirements of Chapter 3 which were derived from use cases provided by a community of collaborators. These requirements led to the following research question:

> *Is it feasible to produce a framework, based upon formal modelling and analysis techniques, to facilitate the automated translation of policies between different access control representations while maintaining the intent of the original policy?*

This thesis has taken the initial steps towards developing a generalised mechanism for the formal translation between access control policies by presenting an example of such a translation.

9.1.2 Formalisation and transformation of access policies: in theory and practice

Arguably, the key requirement that emerges from Chapter 3 is the ability to perform access control policy preserving translations between different access control paradigms. An example of where this is necessary is when a new mechanism for accessing resources is put in place which utilises a different access control representation to the one currently favoured by the organisation. The ability to perform an accurate translation of the currently deployed access control policies to the new representation reduces the immediate impact on the organisation and reduces the chance of erroneous translations caused by lack of knowledge of the new access control paradigm.

The work presented in this thesis used RBAC and XACML as example access control paradigms to provide a proof-of-principle that these goals are realistic and achievable. To this end, the contribution presented in this thesis provides the first steps towards a solution and comprises several

elements. The first element is the definition of formal models in terms of both Z and Alloy. The definitions presented in terms of Z provide a clear description of each access control representation and facilitates an understanding of each paradigm, which is not necessarily clear from a simple text description. Additional confidence in the Z model is gained by the use of Alloy, which provides a mechanism that allows each model to be checked within a given scope.

The next contribution is the formal description of the translation between representations which performs a static transformation of one policy into another maintaining the same behaviour in relation to access requests that are presented. A model of the process is presented in terms of Z and Alloy. The Z model of the process provides a clear and succinct description of each aspect of the translation. The Alloy description, while being a slightly more abstract view of the process, does however facilitate the validation of the technique within a particular scope, which increases confidence in the overall process.

The final contribution is an application based on the formal models that allows the capture of an RBAC policy and the subsequent conversion of this policy into an XACML policy. The resulting XACML policy could then be deployed to a system that utilises XACML to capture the access control policies that are in effect. Although the application is geared towards the example access control representations used to illustrate the process, many of the component parts have been designed with reuse in mind.

9.1.3 Validation

To validate the work, a running example has been used to illustrate each of the formal models, the translation approach and the translation application.

The RBAC and XACML formal descriptions were used to represent the access control policy of the running example. The Z representation demonstrated a succinct and clear representation of the policy; The Alloy representation provided a means of testing the behaviour of each version of the access control policy within a certain scope.

The Z representation of the translation was validated by means of a walk-through of the translation of the RBAC version of the running example into XACML, whereas the Alloy representation of the translation was tested within a given scope by the Alloy Analyzer. The results of the various checks applied to the Alloy translation are illustrated in Appendix C.

The initial testing of the translation application involved the comparison of a manual conversion with the result provided by the application. This was achieved by supplying an example RBAC policy which was then transformed both manually and via the application. The manual conversion was performed by applying the formal translation process to the supplied RBAC policy. The automated transformation by the application was performed after the supplied RBAC policy had been entered using the policy capture panels of the application. The two resulting XACML policies were then compared both manually and by testing that the access control results provided were both consistent with each other and the original RBAC policy.

In addition to providing a means of translation, the application also has the facility to produce an Alloy representation of the RBAC policy under consideration. This Alloy representation can then be checked against the standard RBAC constraints as well as additional user-defined constraints added to the Alloy model of RBAC.

Further to the above validations, the application was used to capture and translate policies as part of a course on Data Security. Eighteen students were each given several exercises which involved designing RBAC policies. These policies were captured and tested using the application. In addition to the development of the RBAC policies, the students had to deploy these policies to a system which utilised XACML. The application was used to provide a translation of the RBAC policies into XACML to allow deployment to the system. This success of this exercise provided additional confidence in both the usability and the functionality of the application.

Additional benefits of the formal models

The formal models of access control policies developed as part of this thesis offer additional benefits over and above the ability to perform translations between different access control representation. One additional benefit explored in the application developed is the facility to define additional constraints on the policy. When considering an RBAC access control policy, the policy writer may consider it important that their policy should meet the following additional criteria:

- Does anyone have all permissions? That is, by virtue of being directly assigned roles and/or by inheriting roles, is any user associated with all the defined permissions.
- Do all roles have at least one permission associated with them either directly or through inheritance? This is a check that every defined role performs a useful function.
- Does every user have at least one role associated with them? This is a check to identify any users that have been defined but have not been granted any useful permissions.
- Is every permission reachable? That is, are there any defined permissions which are not referenced by a role.

These are a few examples of the additional constraints that could be defined within the Alloy model and automatically checked for when using the translation application to capture the RBAC policy.

9.2 Observations

9.2.1 General observations

Comments on Z style

While developing the formal models it became apparent that there are many different ways of representing each aspect of the model and a number of choices have to be made. The properties required of various aspects of the model can restrict the choices available, but there is usually some choice involved which results in individual styles of formal models. In addition to the required properties of the artefact being modelled, the reason for creating the formal model may also have an impact on the choices made. In the case of this thesis, the additional constraint on the model was to ensure the ease of translation to Alloy to allow some assurance of the validity of the models and translation to be gained.

Alloy translation

Although the translation from Z to Alloy has been done by hand it has been approached in a systematic way. The correspondence between the Z and Alloy models was illustrated in Section 2.6. One of the key points is that every atom in Alloy is effectively a set. This leads to the following equivalencies between Alloy and Z:

$$B.A \Leftrightarrow A(\!| \{B\} |\!)$$
$$B.A \Leftrightarrow A(\!| B |\!)$$

That is, in Alloy `A` and `B` are both effectively sets, even if `B` is a single value. The Alloy `B.A` is equivalent to the the Z $A(\!| \{B\} |\!)$, which is A the relational image of A on B. The two different translations shown above depend upon B. If B is not a set, then the first translation is needed — essentially creating a set which contains B; if, however, B is a set, then the second translation is used.

Tools

The lack of useable tools for Z has led to the hybrid approach taken in this thesis. The primary tools that are readily available for Z are type checkers: Fuzz and the type checker from the CZT project.

To overcome this, the Alloy Analyzer has been used to help validate the models produced and the overall approach taken to the translation between access control paradigms. There are however problems with Alloy, which is generally less expressive than Z. In addition to this, Alloy does not permit the use of recursion, which can result in difficulties in capturing systems in an entirely natural way. That is, for certain systems it would be natural to model particular processes as recursive, but for the model to work in Alloy these design decisions have to be rethought by either unwinding the recursion if it is a small number of steps or finding an alternative approach.

9.2.2 Limitations of the work

Scalability

While the model finder used by Alloy is capable of dealing with the case when the relations are all fixed, it still is restricted by internal data structures which put a limit on the total number of atoms of approximately $2^{31/n}$ where n is the largest arity of any relation in scope. As even a simple signature such as `sig S { x : univ -> univ}` yields a relation of arity 3 that places a practical limit of approximately 1000 total atoms including the 2 bitwidth integers that are automatically generated.

The translation process using the Alloy model is limited further because of the number of different signatures that have to be instantiated during the translation. This means that the Alloy translation process will only work for RBAC models with less than approximately 200 atoms in the RBAC model. However, the translation application can deal with many more items and has been tested with automatically generated RBAC policies with more than a 1000 atoms. The limits on the application are primarily due to the amount of memory available to the Java

virtual machine being used.

9.3 Future work

The primary aim of future work is to continue moving towards fulfilling the use cases and requirements of Chapter 3, as well as new requirements as they arise. The work in this thesis has demonstrated a mechanism for achieving the translation between access control policies using RBAC and XACML as a proof-of-concept.

The first task is to develop formal models of additional access control representations and the translations between between them. In addition, to assist in the translation process, the development of a formal model of a generic representation will be necessary — as discussed in Section 3.5.

The formal model of the generic representation would allow the formal definitions of translations to and from the generic representations to be developed for each of the access control models that have been defined formally.

Another use of the formal models of the access control representations and the formally defined translation paths between them, could be to facilitate the development of techniques to compare policies. The comparison would be to ascertain if two access control policies defined in different paradigms are, in fact equivalent — that is would provide the same access control results for every access control request presented. The comparison would be achieved by translating either one of the policies into the other, or by translating both into a generic representation to facilitate comparison.

Another important task will be to develop a method to automate the translation between Z and Alloy (extending the work of [69]), rather than perform manual translation. To facilitate the automated translation, the Z models would have to be developed in a way that took into account the limitations of Alloy. In addition to the translation between the formal representations, being able to go from the Alloy definition to code would also be a useful addition.

In addition to the translation between formal representations, it would be useful to have a better way of entering additional constraints on an Alloy model. The need for this would be to enter any additional user-defined constrains that are necessary over and above the general constraints of the access control paradigm being used. The examples given in this thesis were additional constraints on the RBAC model.

It is also necessary to develop improved tools for the capturing of access control policies in a number of different representations. Once the policies have been captured, the tools need to be able to translate the policy either into another specific representation or into a generic representation. In addition to being able to generate a generic representation a further requirement would be that the tool should be able to read in a policy in a generic representation and convert it into the representation native to the tool.

In addition to developing translations between different representations, it is necessary to formally define the constraints that need to be meet before a translation can occur. Of course, it may not always be possible to reverse translations as some information may be lost when changing between representations.

In addition to the general work outlined above, a possible future direction of work is to

consider combining policies — especially those originally captured using different access control representations, where it is important to be sure that the overall policy is valid and consistent. Formal descriptions of the translations and combinations of policies would provide a mechanism for validating that the overall policy conforms to the representation in which it is expressed. The formal description could also be utilised for checking the consistency of the overall policy, perhaps by ensuring that individuals are not permitted and denied the same permission within the resulting combined policy.

9.4 Concluding remarks

This thesis has used RBAC and XACML to demonstrate the use of formal methods to both describe access control mechanisms and to underpin translation between the different representations. In addition to providing a level of assurance that the translation results in a policy that provides the same result for any given access control request, the use of a formal description also provides the ability to validate that policies meet certain criteria before they are deployed. The work of this thesis has provided the requirements for, and the first steps towards, the development of a framework to facilitate the automated translation of policies between access control paradigms.

Bibliography

[1] Community Z Tools (CZT) project. http://czt.sourceforge.net/.

[2] Privacy Act of 1974, Title 5 United States Code §552a. http://www.justice.gov/opcl/1974privacyact-overview.htm.

[3] Web Services Architecture. W3C Working Group Note, W3C, February 2004. Latest version available at http://www.w3.org/TR/ws-arch/.

[4] Reference model for Service Oriented Architecture 1.0. http://www.oasis-open.org, August 2006. Committee Specification.

[5] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):706–734, 1993.

[6] D. Abi Haidar, N. Cuppens-Boulahia, F. Cuppens, and H. Debar. An extended RBAC profile of XACML. In *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, pages 13–22, New York, NY, USA, 2006. ACM.

[7] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[8] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, pages 120–131, June 2003.

[9] M. S. Ackerman, L. F. Cranor, and J. Reagle. Privacy in e-commerce: examining user scenarios and privacy preferences. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 1–8, New York, NY, USA, 1999. ACM.

[10] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 143–154. Morgan Kaufmann, 2002.

[11] S. R. Amendolia, F. Estrella, M. W. Hassan, T. Hauer, D. Manset, R. McClatchey, D. Rogulin, and T. Solomonides. MammoGrid: A service oriented architecture based medical grid application. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *Grid and Cooperative Computing*, volume 3251 of *Lecture Notes in Computer Science*, pages 939–942. Springer, 2004.

[12] A. H. Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *SWS '06: Proceedings of the 3rd ACM workshop on Secure Web Services*, pages 53–60, New York, NY, USA, 2006. ACM Press.

[13] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, Bedford, MA 01731, October 1972.

[14] ANSI. American National Standard for Information Technology — Role Based Access Control. ANSI INCITS 359–2004, February 2004.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[16] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proceedings of the IEEE Computer Security Symposium on Research in Security and Privacy*, pages 116–132, 1990.

[17] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):39–91, 2006.

[18] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[19] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.

[20] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:159, 2004.

[21] D. E. Bell and L. J. La Padula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corporation, 1975.

[22] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

[23] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, The Mitre Corportion, Bedford, MA, April 1977.

[24] J. M. Brady, D. J. Gavaghan, A. C. Simpson, M. Mulet-Parada, and R. P. Highnam. e-DiaMoND: A Grid-enabled federated database of annotated mammograms. In F. Berman, G. C. Fox, and A. J. G. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 923–943. Wiley Series, 2003.

[25] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the IEEE Computer Security Symposium on Research in Security and Privacy*, pages 215–228, April 1989.

[26] J. Bryans. Reasoning about XACML policies using CSP. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35, New York, NY, USA, 2005. ACM Press.

[27] J. Bryans and J. S. Fitzgerald. Formal engineering of XACML access control policies in VDM++. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *ICFEM'07: Proceedings of the 9th International Conference on Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2007.

[28] A. Bullock and S. Benford. An approach to access control for collaborative virtual environments. In *Proceedings of the 6th European Research Consortium for Informatics and Mathematics (ERCIM) Workshop on Distributed Virtual Environments*, pages 233–264, June 1994.

[29] A. Bullock and S. Benford. An access control framework for multi-user collaborative environments. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 140–149, New York, NY, USA, 1999. ACM.

[30] D. W. Chadwick and A. Otenko. The PERMIS X.509 role based privilege management infrastructure. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, SACMAT '02, pages 135–140, New York, NY, USA, 2002. ACM.

[31] D. W. Chadwick, G. Zhao, S. Otenko, R. Laborde, L. Su, and T. A. Nguyen. PERMIS: a modular authorization infrastructure. *Currency and Computation: Practice and Experience*, 20:1341–1357, August 2008.

[32] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service definition language (WSDL) 1.1. W3C Note, World Wide Web Consortium (W3C), March 2001. http://www.w3.org/TR/wsdl.

[33] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[34] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In T. Dimitrakos and F. Martinelli, editors, *Formal Aspects in Security and Trust*, volume 173 of *IFIP International Federation for Information Processing*, pages 203–218. Springer, 2005.

[35] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer-Verlag, 2001.

[36] J. E. Dobson and J. A. McDermid. Security models and enterprise models. In C. E. Landwehr, editor, *Database Security II: Status & Prospects*, pages 1–39. North Holland, 1989.

[37] S. E. Dunne and A. Howitt. Modelling Role-based Access Control in B. In H. Habrias and C. Attiogbé, editors, *The B Formal Method: From Research to Teaching*, pages 92–108. Association de Pilotage des Conférences B (APCB), 2008.

[38] D. F. Ferraiolo, D. Gilbert, and N. Lynch. An examination of federal and commercial access control policy needs. In *Proceedings of the NIST-NSA National (USA) Computer Security Conference*, pages 107–116, 1993.

[39] D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proceedings of the 15th National Information Systems Security Conference*, pages 554–563. National Institute of Standards and Technology, 1992.

[40] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based access control*. Artech House Publishers, 2003.

[41] D. F. Ferraiolo, R. S. Sandhu, S. Gavrilla, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, August 2001.

[42] I. Foster and C. Kesselman, editors. *The Grid: Blueprint For a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[43] A. G. Gale. PERFORMS: a self assessment scheme for radiologists in breast screening. *Seminars in Breast Disease: Improving and monitoring mammographic interpretative skills*, 6(3):148–152, 2003.

[44] J. Geddes, S. Lloyd, A. C. Simpson, M. Rossor, N. Fox, D. Hill, J. V. Hajnal, S. Lawrie, A. McIntosh, E. Johnstone, J. Wardlaw, D. Perry, R. Procter, P. Bath, and E. Bullimore. Neurogrid: Using Grid technology to advance Neuroscience. In *Proceedings of Computer-Based Medical Systems (CBMS) 2005*, pages 570–572. IEEE Computer Society, 2005.

[45] M. M. Greenberg, C. Marks, L. A. Meyerovich, and M. C. Tschantz. The Soundness and Completeness of Margrave with Respect to a Subset of XACML. Technical Report CS-05-05, Brown University, Providence, Rhode Island 02912, April 2005.

[46] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2. World Wide Web Consortium, Working Draft WD-soap12-20010709, July 2001.

[47] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[48] V. C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance checking of access control policies specified in XACML. In *Proceedings of the 1st IEEE International Workshop on Security in Software Engineering (IWSSE 2007), Beijing, China*, pages 275–280, July 2007.

[49] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, University of California, Santa Barbara, Santa Barbara, CA 93106-5110, September 2004.

[50] G. Hughes and T. Bultan. Automated verification of XACML policies using a SAT solver. In *Proceedings of the Workshop on Web Quality, Verification and Validation (WQVV '07)*, pages 378–392. Springer-Verlag, 2007.

[51] P. Humenn. The formal semantics of XACML. `http://lists.oasis-open.org/archives/xacml/200310/pdf00000.pdf`, October 2003.

[52] ISO. Information technology – Open Systems Interconnection – security frameworks for open systems: Access control framework. Technical Report ISO/IEC 10181-3:1996, International Organization for Standardization, 1996.

[53] ISO. Information technology – Z formal specification notation – syntax, type system and semantics. Technical Report ISO/IEC 13568, International Organization for Standardization, 2002.

[54] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[55] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[56] T. Jaeger. Managing access control complexity using metrics. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 131–139, New York, NY, USA, 2001. ACM.

[57] T. Jaeger and J. E. Tidswell. Rebuttal to the NIST RBAC model proposal. In *Proceedings of the fifth ACM workshop on role-based access control*, pages 65–66. ACM, 2000.

[58] E. J. Khayat and A. E. Abdallah. A formal model for flat role-based access control. In *ACS/IEEE International Conference on Computer Systems and Applications*, page 75. IEEE, July 2003.

[59] K. I. Kim, H. J. Ko, W. G. Choi, E. J. Lee, and U. M. Kim. A collaborative access control based on XACML in pervasive environments. In *ICHIT '08: Proceedings of the 2008 International Conference on Convergence and Hybrid Information Technology*, pages 7–13, Washington, DC, USA, 2008. IEEE Computer Society.

[60] V. Kolovski. Formalizing XACML using defeasible description logics. Technical Report TR-233-11, University of Maryland, 2006.

[61] P Kumaraguru and L Cranor. Privacy in India: Attitudes and awareness. In G. Danezis and D. Martin, editors, *Proceedings of the 2005 Workshop on Privacy Enhancing Technologies (PET 2005)*, volume 3856 of *Lecture Notes in Computer Science*, pages 243–258, 2005.

[62] B. W. Lampson. Protection. *SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[63] E. Laure and B. Jones. Enabling Grids for e-Science: The EGEE Project. Technical Report EGEE-PUB-2009-001, CERN, September 2008.

[64] A. Lawrence. Astrogrid: powering the virtual observatory. In A. S. Szalay, editor, *Proceedings of SPIE Virtual Observatories*, volume 4846, pages 6–12. SPIE, 2002.

[65] N. Li, Z. Bizri, and M. V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS-11)*, pages 42–51. ACM Press, October 2004.

[66] N. Li, J.-W. Byun, and E. Bertino. A critique on the ANSI standard on role-based access control. Technical Report CERIAS TR 2005-29, Department of Computer Science, Purdue University, 2005.

[67] N. Li, J. Hwang, and T. Xie. Multiple-implementation testing for XACML implementations. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 27–33. ACM, 2008.

[68] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, New York, NY, USA, 2003. ACM Press.

[69] P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Proceedings of Abstract State Machines, Alloy, B and Z (ABZ) 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 377–390, 2010.

[70] J. D. Moffett. Control principles and role hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*, RBAC '98, pages 63–69, New York, NY, USA, 1998. ACM.

[71] L. Momtahan, S. Lloyd, and A. C. Simpson. Switched lightpaths for e-health applications: Issues and challenges. In *CBMS '07: Proceedings of the Twentieth IEEE International Symposium on Computer-Based Medical Systems*, pages 459–464, Washington, DC, USA, 2007. IEEE Computer Society.

[72] L. Momtahan and A. C. Simpson. Switched lightpaths for e-health applications: a feasibility study. In *CBMS '06: Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems*, pages 469–472, Washington, DC, USA, 2006. IEEE Computer Society.

[73] T. Moses. OASIS eXtensible Access Control Markup Language (XACML) version 2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, February 2005. committee specification.

[74] M. Nash and K. Poland. Some conundrums concerning separation of duty. In *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 201–207. IEEE, 1990.

[75] OASIS. Core and hierarchical role based access control (RBAC) profile of XACML v2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf, February 2005.

[76] OASIS. Security Assertion Markup Language (SAML) 2.0 specification. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf, March 2005.

[77] Web Services Security (WSS) Technical Committee. www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, 2005.

[78] Web Services Reliable Messaging (WS-ReliableMessaging), Committee Draft 04. http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-spec-cd-04.pdf, August 2006.

[79] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3:85–106, May 2000.

[80] J. Park and R. Sandhu. Towards usage control models: beyond traditional access control. In *SACMAT '02: Proceedings of the seventh ACM Symposium on Access Control Models and Technologies*, pages 57–64, New York, NY, USA, 2002. ACM.

[81] J. Park and R. Sandhu. The UCON_{ABC} usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, February 2004.

[82] J. Pitt-Francis, M. O. Bernabeu, J. Cooper, A. Garny, L. Momtahan, J. Osborne, P. Pathmanathan, B. Rodriguez, J. P. Whiteley, and D. J. Gavaghan. Chaste: using agile programming techniques to develop computational biology software. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1878):3111–3136, 2008.

[83] J. Pitt-Francis, D. Chen, M. A. Slaymaker, A. C. Simpson, J. M. Brady, I. van Leeuwen, R. Reddington, P. Quirke, and D. J. Gavaghan. Multimodal imaging techniques for the extraction of detailed geometrical and physiological information for use in multi-scale models of colorectal cancer and treatment of individual patients. *Computational & Mathematical Methods in Medicine Mathematical Methods in Medicine*, 7(2):177 – 188, 2006.

[84] D. J. Power, E. A. Politou, M. A. Slaymaker, and A. C. Simpson. Towards secure grid-enabled healthcare. *Software: Practice and Experience*, 35(9):857–871, 2005.

[85] D. J. Power, M. A. Slaymaker, and A. C. Simpson. On formalizing and normalizing role-based access control systems. *The Computer Journal*, 52(3):305–325, May 2009.

[86] D. J. Power, M. A. Slaymaker, and A. C. Simpson. Automatic conformance checking of role-based access control policies via alloy. In *Engineering Secure Software and Systems (ESSOS 2011)*, volume 6542 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, 2011.

[87] R. Rieke. Modelling and analysing network security policies in a given vulnerability setting. *Critical Information Infrastructures Security*, pages 67–78, 2006.

[88] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[89] R. Sandhu, D. F. Ferraiolo, and D. R. 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 2000)*, pages 47–63, 2000.

[90] H. Shen and P. Dewan. Access control for collaborative environments. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-Supported Cooperative Work*, pages 51–58, New York, NY, USA, 1992. ACM.

[91] A. C. Simpson, D. J. Power, D. Russell, M. A. Slaymaker, V. Bailey, C. E. Tromans, J. M. Brady, and L. Tarassenko. GIMI: the past, the present, and the future. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368:3891–3905, 2010.

[92] A. C. Simpson, D. J. Power, D. Russell, M. A. Slaymaker, G. Kouadri-Mostefaoui, X. Ma, and G. Wilson. A healthcare-driven framework for facilitating the secure sharing of data across organisational boundaries. *Studies in Health Technology and Informatics*, 138:3–12, 2008.

[93] A. C. Simpson, D. J. Power, M. A. Slaymaker, and E. A. Politou. GIMI: Generic infrastructure for medical informatics. In *Proceedings of the 18th IEEE Symposium on Computer-Based Medical Systems*, pages 564–566, 2005.

[94] A. C. Simpson, D. J. Power, M. A. Slaymaker, D. Russell, and M. Katzarova. On the development of secure service-oriented architectures to support medical research. *International Journal of Healthcare Information Systems and Informatics*, 2(2):75–89, 2007.

[95] A. C. Simpson, M. A. Slaymaker, and D. J. Gavaghan. On the secure sharing and aggregation of data to support systems biology research. In Lambrix, P. and Kemp, G. J. L., editor, *Proceedings of the 7th International Conference on Data Integration in the Life Sciences (DILS 2010)*, volume 6254 of *Lecture Notes in Computer Science*, pages 58–73. Springer-Verlag, 2010.

[96] M. A. Slaymaker, D. J. Power, D. Russell, and A. C. Simpson. On the facilitation of fine-grained access to distributed healthcare data. In W. Jonker and M. Petkovic, editors, *Proceedings of Secure Data Management 2008*, volume 5159 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, 2008.

[97] M. A. Slaymaker, D. J. Power, D. Russell, G. Wilson, and A. C. Simpson. Accessing and aggregating legacy data sources for healthcare research, delivery and training. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, pages 1317–1324. ACM, March 2008.

[98] M. A. Slaymaker, D. J. Power, and A. C. Simpson. Formalising and validating RBAC-to-XACML translation using lightweight formal methods. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Proceedings of Abstract State Machines, Alloy, B and Z (ABZ) 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 349–362, 2010.

[99] M. A. Slaymaker, A. C. Simpson, J. M. Brady, D. J. Gavaghan, F. Reddington, and P. Quirke. A prototype infrastructure for the secure aggregation of imaging and pathology data for colorectal cancer care. In *Proceedings of the 19th IEEE Symposium on Computer Based Medical Systems*, pages 63–68. IEEE Computer Society Press, 2006.

[100] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, second edition, 1992.

[101] J. M. Spivey. *The Fuzz Manual*, second edition, 2000.

[102] S. Stepney and S. P. Lord. Formal specification of an access control system. *Software—Practice and Experience*, 17(9):575–593, 1987.

[103] D. J. Thomsen. Role-based application design and enforcement. In S. Jajodia and C. E. Landwehe, editors, *Database Security IV: Status & Prospects*, pages 151–168. North Holland, 1991.

[104] C. Tromans and J. M. Brady. A scatter model for use in measuring volumetric mammographic breast density. In Astley, S. M. and Brady, J. M. and Rose, C. and Zwiggelaar, R., editor, *Proceedings of 8th International Workshop on Digital Mammography (IWDM) 2006*, volume 4046 of *Lecture Notes in Computer Science*, pages 251–258. Springer-Verlag , June 2006.

[105] F. Turkmen and B. Crispo. Performance evaluation of XACML PDP implementations. In *SWS '08: Proceedings of the 2008 ACM workshop on Secure web services*, pages 37–44, New York, NY, USA, 2008. ACM.

[106] Q. Wang, H. Jin, and N. Li. Usable access control in collaborative environments: Authorization based on people-tagging. *Computer Security –ESORICS 2009*, pages 268–284, 2010.

[107] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. http://www.ietf.org/rfc/rfc3198.txt, 2001.

[108] A. F. Westin. e-Commerce and Privacy: What Net Users Want. Privacy & American Business, 1998.

[109] J. C. P. Woodcock and J. W. M. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[110] M. Wu, T. Zhao, and C. Wu. Public health data collection and sharing using HIPAA messages. *Journal of Medical Systems*, 29(4):303–316, August 2005.

[111] R. Yavatkar, D. Pendarakis, and R. Guerin. RFC 2753: A Framework for Policy-based Admission Control. http://www.rfc-archive.org/getrfc.php?rfc=2753, January 2000.

[112] C. Yuan, Y. He, J. He, and Z. Zhou. A verifiable formal specification for RBAC Model with Constraints of Separation of Duty. In H. Lipmaa, M. Yung, and D. Lin, editors, *Information Security and Cryptology*, volume 4318 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2006.

[113] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 56–65, New York, NY, USA, 2004. ACM.

[114] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.

Appendix A

XACML representations of the running example

A.1 Example XACML policy set

```
<?xml version="1.0" encoding="UTF-8"?> <PolicySet
  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
      http://docs.oasis-open.org/xacml/access_control-xacml-2.0-policy-schema-os.xsd"
  PolicySetId="PS_PrescribeDB"
  PolicyCombiningAlgId=
    "urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

  <Description>
     An example policy set for restricting read and write requests to prescribeDB.
  </Description>

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

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

  <Resources>
    <Resource>
      <ResourceMatch
         MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue
           DataType="http://www.w3.org/2001/XMLSchema#string">prescribeDB
        </AttributeValue>
        <ResourceAttributeDesignator
           AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </ResourceMatch>
    </Resource>
  </Resources>
</Target>

<Policy
   PolicyId="P_Dr"
   RuleCombiningAlgId=
      "urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Description>
     Policy for doctors -- Morris and Rover
  </Description>

  <Target>
    <Subjects>
```

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

<Rule
   RuleId="R_ReadPres"
   Effect="Permit">
  <Description>
     Read a the prescribeBD
  </Description>

  <Target>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
             read
          </AttributeValue>
          <ActionAttributeDesignator
             AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
             DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
</Rule>
<Rule
   RuleId="R_WritePres"
   Effect="Permit">
  <Description>
     Write to prescribeBD
  </Description>
  <Target>
```

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

<Policy
   PolicyId="P_Nurse"
   RuleCombiningAlgId=
     "urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Description>
     Policy for Nurses -- Austin or Triumph
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch
           MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
             DataType="http://www.w3.org/2001/XMLSchema#string">Austin
          </AttributeValue>
          <SubjectAttributeDesignator
             AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
             DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch
           MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
             DataType="http://www.w3.org/2001/XMLSchema#string">Triumph
          </AttributeValue>
          <SubjectAttributeDesignator
             AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
             DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </SubjectMatch>
      </Subject>
   </Subjects>
  </Target>

  <Rule
     RuleId="R_ReadPres"
     Effect="Permit">
```

```
      <Description>
         Read a the prescribeBD
      </Description>

      <Target>
        <Actions>
          <Action>
            <ActionMatch
               MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                 read
              </AttributeValue>
              <ActionAttributeDesignator
                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                 DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

A.2 A formal representation of the running example in XACML

$$
\begin{array}{|l}
Austin : Subject \\
Morris : Subject \\
Rover : Subject \\
Triumph : Subject \\
read : Action \\
write : Action \\
prescribeDB : Resource
\end{array}
$$

$$
\begin{array}{|l}
matchPrescribeDB : Resource \rightarrow EvalRes \\
\hline
\forall\, r : Resource \bullet r = prescribeDB \Rightarrow matchPrescribeDB\, r = TRUE \\
\forall\, r : Resource \bullet r \neq prescribeDB \Rightarrow matchPrescribeDB\, r = FALSE
\end{array}
$$

$$
\begin{array}{|l}
matchNurse : Subject \rightarrow EvalRes \\
\hline
\forall\, r : Subject \bullet r = Austin \lor r = Triumph \Rightarrow matchNurse\, r = TRUE \\
\forall\, r : Subject \bullet r \neq Austin \land r \neq Triumph \Rightarrow matchNurse\, r = FALSE
\end{array}
$$

$$matchDoctor : Subject \rightarrow EvalRes$$

$$\forall r : Subject \bullet r = Morris \lor r = Rover \Rightarrow matchDoctor\, r = TRUE$$
$$\forall r : Subject \bullet r \neq Morris \land r \neq Rover \Rightarrow matchDoctor\, r = FALSE$$

$$matchRead : Action \rightarrow EvalRes$$

$$\forall r : Action \bullet r = read \Rightarrow matchRead\, r = TRUE$$
$$\forall r : Action \bullet r \neq read \Rightarrow matchRead\, r = FALSE$$

$$matchWrite : Action \rightarrow EvalRes$$

$$\forall r : Action \bullet r = write \Rightarrow matchWrite\, r = TRUE$$
$$\forall r : Action \bullet r \neq write \Rightarrow matchWrite\, r = FALSE$$

$$xacml : XACML$$

$$T\_PrescribeDB : TargetID$$

$$(xacml.getTarget(T\_PrescribeDB)).tid = T\_PrescribeDB$$
$$(xacml.getTarget(T\_PrescribeDB)).sub = \langle matchDoctor, matchNurse \rangle$$
$$(xacml.getTarget(T\_PrescribeDB)).act = \langle\rangle$$
$$(xacml.getTarget(T\_PrescribeDB)).res = \langle matchPrescribeDB \rangle$$
$$(xacml.getTarget(T\_PrescribeDB)).env = \langle\rangle$$

$$T\_Doctor : TargetID$$

$$(xacml.getTarget(T\_Doctor)).tid = T\_Doctor$$
$$(xacml.getTarget(T\_Doctor)).sub = \langle matchDoctor \rangle$$
$$(xacml.getTarget(T\_Doctor)).act = \langle\rangle$$
$$(xacml.getTarget(T\_Doctor)).res = \langle\rangle$$
$$(xacml.getTarget(T\_Doctor)).env = \langle\rangle$$

$$T\_Nurse : TargetID$$

$$(xacml.getTarget(T\_Nurse)).tid = T\_Nurse$$
$$(xacml.getTarget(T\_Nurse)).sub = \langle matchNurse \rangle$$
$$(xacml.getTarget(T\_Nurse)).act = \langle\rangle$$
$$(xacml.getTarget(T\_Nurse)).res = \langle\rangle$$
$$(xacml.getTarget(T\_Nurse)).env = \langle\rangle$$

$T\_Read : TargetID$

$(xacml.getTarget(T\_Read)).tid = T\_Read$
$(xacml.getTarget(T\_Read)).sub = \langle\rangle$
$(xacml.getTarget(T\_Read)).act = \langle matchRead\rangle$
$(xacml.getTarget(T\_Read)).res = \langle\rangle$
$(xacml.getTarget(T\_Read)).env = \langle\rangle$

$T\_Write : TargetID$

$(xacml.getTarget(T\_Write)).tid = T\_Write$
$(xacml.getTarget(T\_Write)).sub = \langle\rangle$
$(xacml.getTarget(T\_Write)).act = \langle matchWrite\rangle$
$(xacml.getTarget(T\_Write)).res = \langle\rangle$
$(xacml.getTarget(T\_Write)).env = \langle\rangle$

$R\_ReadPres : RuleID$

$(xacml.getRule(R\_ReadPres)).rid = R\_ReadPres$
$(xacml.getRule(R\_ReadPres)).target = T\_Read$
$(xacml.getRule(R\_ReadPres)).effect = Permit$
$(xacml.getRule(R\_ReadPres)).condition = \langle\rangle$

$R\_WritePres : RuleID$

$(xacml.getRule(R\_WritePres)).rid = R\_WritePres$
$(xacml.getRule(R\_WritePres)).target = T\_Write$
$(xacml.getRule(R\_WritePres)).effect = Permit$
$(xacml.getRule(R\_WritePres)).condition = \langle\rangle$

$P\_Dr : PolicyID$

$(xacml.getPolicy(P\_Dr)).pid = P\_Dr$
$(xacml.getPolicy(P\_Dr)).target = T\_Doctor$
$(xacml.getPolicy(P\_Dr)).rca = rulPermitOverride$
$(xacml.getPolicy(P\_Dr)).inPol = \langle R\_ReadPres, R\_WritePres\rangle$
$(xacml.getPolicy(P\_Dr)).obli = \emptyset$

$$P\_Nurse : PolicyID$$

$$(xacml.getPolicy(P\_Nurse)).pid = P\_Nurse$$
$$(xacml.getPolicy(P\_Nurse)).target = T\_Nurse$$
$$(xacml.getPolicy(P\_Nurse)).rca = rulPermitOverride$$
$$(xacml.getPolicy(P\_Nurse)).inPol = \langle R\_ReadPres \rangle$$
$$(xacml.getPolicy(P\_Nurse)).obli = \emptyset$$

$$PS\_PrescribeDB : PolicySetID$$

$$(xacml.getPolicySet(PS\_PrescribeDB)).psid = PS\_PrescribeDB$$
$$(xacml.getPolicySet(PS\_PrescribeDB)).target = T\_PrescribeDB$$
$$(xacml.getPolicySet(PS\_PrescribeDB)).pca = polPermitOverride$$
$$(xacml.getPolicySet(PS\_PrescribeDB)).inPolSet = \langle Pol(P\_Dr), Pol(P\_Nurse) \rangle$$
$$(xacml.getPolicySet(PS\_PrescribeDB)).obli = \emptyset$$

A.3 An example XACML request

```
<?xml version="1.0" encoding="UTF-8" ?>
<Request
  xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:context:schema:os
  http://docs.oasis-open.org/xacml/access_control-xacml-2.0-context-schema-os.xsd">

    <Subject>
        <Attribute
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
          DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>Morris</AttributeValue>
          </Attribute>
    </Subject>
    <Resource>
        <Attribute
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
          DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>prescribeDB</AttributeValue>
        </Attribute>
    </Resource>
    <Action>
        <Attribute
          AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
          DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>write</AttributeValue>
        </Attribute>
    </Action>
    <Environment />
</Request>
```

A.4 A formal representation of an XACML request

$Rq\_MorrisWrite : RequestID$

We can then define the request as follows.

$morrisWrite : Request$

$morrisWrite.request = Rq\_MorrisWrite$
$morrisWrite.sub = \{Morris\}$
$morrisWrite.act = \{write\}$
$morrisWrite.res = \{prescribeDB\}$

Appendix B

Capturing use cases and requirements

In addition to discussions held with developers from the GIMI project that informed the use cases and requirements of Chapter 3, additional requirements were provided by other projects. Other than GIMI, the two projects which provided most input to the requirements gathering process were eDiaMoND and NeuroGrid.

B.1 eDiaMoND

The eDiaMoND project was a two-year project primarily concerned with ascertaining the feasibility of facilitating access to a federated resource of mammograms to achieve three main goals. The first was to provide a training facility that permitted radiologists to be trained using sample cases from multiple breast screen units. The second was to facilitate the testing of automated screening algorithms to ascertain if they were capable of identifying 'regions of interest' within a mammogram which warranted further, more detailed investigation. And thirdly to facilitate the remote reading of mammograms by units which had spare capacity to more evenly distribute the work load between centres.

The eDiaMoND project consisted of several groups working together towards the overall goal. A group of three developers from IBM, along with three developers based at the Oxford University Computing Laboratory, produced the underlying middleware which was used to federate the data. Mirada, a small commercial company, produced software which permitted a radiologist to read a mammogram as well as the core software to be utilised in the training application. In addition to the two technology teams, there were four application teams — each associated with a breast screen unit. The teams based at Edinburgh University and University College London were concerned with the training application, working closely with their screening units based at Ardmillian Hospital in Edinburgh and St George's Hospital in London. The developer based in the University of Oxford Engineering Science Department was concerned with developing image processing algorithms that could identify abnormalities within a mammographic image. The Breast Screening Unit at the Churchill Hospital, Oxford provided a number of appropriate

case for assisting in the initial work on the algorithms. The developer associated with Kings College London was primarily concerned with investigating the quality of digital mammograms and worked closely with Guy's Hospital London.

The use of medical data within the project resulted in the security of the data — and appropriate access to it — being at the forefront throughout. This led to a number of requirements and use cases that went on to inform the contribution of Chapter 3.

B.2 NeuroGrid

The NeuroGrid project was a three-year project which had the goal of enhancing collaboration between clinical, imaging and e-scientists to create a Grid-based network of neuroimaging centres and a neuroimaging tool-kit. Sharing data, experience and expertise to facilitate the archiving, curation, retrieval and analysis of imaging data from multiple sites and enable large-scale clinical studies. The aim was to demonstrate the principles using the exemplars in the areas of Stroke, Dementia and Psychosis, which were the areas of expertise of the contributing clinical partners.

The project team was made up of partners from: The University of Oxford, University College London, Imperial College London, the University of Edinburgh, the University of Nottingham, Newcastle University and Adenbrookes Hospital Cambridge.

A common concern relating to the security of data and the development of access control policies was expressed by the majority of partners associated with the project. The most common issue raised was the manual translation of access control policies into an unfamiliar paradigm. The reason for the translation was down to the middleware using a different access control mechanism to that used by the data owners. In addition to the middlewares access control being different, because the middleware used XACML it was considered a more complicated mechanism.

B.3 Elicitation and capture

The requirements were gathered in a number of ways, ranging from formal meetings to ad-hoc discussions. The formal meeting were divided between meetings with end users, meetings with developers and general meetings involving representatives of all participants. The end user meetings allowed non technical input to be considered by the developers of applications. The technical meetings involved the various developers associated with the different exemplars and generated the core technical requirements that would enable the requirements form the users to be achieved. The meetings between all groups provided fora for discussing various solutions and working out compromises between incompatible requirements.

Formal requirements documents were produced for both eDiaMoND and NeuroGrid which were validated by a process of consultation. The consultation process checked with each stakeholder group that the requirements presented in the document were consistent with the opinions that they had expressed and the compromises that been agreed to. Both of these documents helped inform the contribution of Chapter 3.

Additional requirements were also gathered from smaller associated projects such as the NCRI Informatics Initiative demonstrator project and work with OPTIMA, as described in Chapter 3.

Appendix C

Alloy validation of the translation

C.1 An Alloy representation of the running example

```
module accesscontrol/example1
open accesscontrol/rbactoxacml

-- Running example in Alloy
one sig Morris, Austin, Rover,  Triumph extends User {}
one sig Doctor, Nurse extends Role{}
one sig read, modify extends Action{}
one sig PrescribeDB extends Resource{}

one sig writePrescription extends PRMSBase { }
{ action = modify && resource = PrescribeDB }

one sig readPrescription extends PRMSBase { }
{ action = read && resource = PrescribeDB }

one sig prescribeh extends Hierarchy { }
{
    UA = (Morris -> Doctor) + (Rover -> Doctor) + (Austin -> Nurse) + (Triumph -> Nurse)
    PA = (Nurse -> readPrescription) + (Doctor -> writePrescription)
    RH = (Doctor -> Nurse)
    USERS = Morris + Rover + Austin + Triumph
    ROLES = Doctor + Nurse
    PRMS =  readPrescription + writePrescription
}
```

C.2 Alloy tests

```
module accesscontrol/example1test3

open accesscontrol/example1
-- Test the running example
-- General Tests of the generator functions
pred ruexist(ru : Rule){
    ru.@t.sub = none &&
```

```
    ru.@t.act = readPrescription.action &&
    ru.@t.res = readPrescription.resource &&
    ru.eff = Permit
}
run ruexist for 5 but 1 Rule expect 1

pred rpexist(rp : rolePolicy){
  rp.prms = readPrescription + writePrescription
}
run rpexist for 5 but 10 XACMLelementBase expect 1

pred rrpsexist(urps : rolerolePolicySet){
    urps.h = prescribeh
    urps.r = Doctor
}
run rrpsexist for 5 but 10 XACMLelementBase expect 1

pred urpsexist(urps : roleusersPolicySet){
    urps.h = prescribeh
    urps.r = Nurse
}
run urpsexist  for 5 but 10 XACMLelementBase expect 1

pred urpsexist1(urps : roleusersPolicySet){
    urps.h = prescribeh
    urps.r = Doctor
}
run urpsexist1  for 5 but 10 XACMLelementBase expect 1

pred xacmlconvexist (rps : rootPolicySet ){
    rps.h = prescribeh
}
run xacmlconvexist  for 6 but 12 XACMLelementBase  , 8 int  expect 1

-- Requests  - explicitly generate all requests associated with the defined users of the system
-- Request Austin Read
one sig reqAustinR extends Request { }
{ u = Austin &&  p = readPrescription }

-- Request Austin write
one sig reqAustinW extends Request { }
{ u = Austin && p = writePrescription }

-- Request Triumph Read
one sig reqTriumphR extends Request { }
{ u = Triumph && p = readPrescription }

-- Request Triumph write
one sig reqTriumphW extends Request { }
{ u = Triumph && p = writePrescription }

-- Request Morris Read
one sig reqMorrisR extends Request { }
{ u = Morris && p = readPrescription }
```

```
-- Request Morris write
one sig reqMorrisW extends Request { }
{ u = Morris && p = writePrescription }

-- Request Rover Read
one sig reqRoverR extends Request { }
{ u = Rover && p = readPrescription }

-- Request Rover write
one sig reqRoverW extends Request { }
{ u = Rover && p = writePrescription }


-- Specific tests
-- 1) Austin and Triumph can read but not write

--    Check the RBACH instance first
--        Expect to get Deny for Austin Write request - so no counter example expected
assert evalRbachAustinWriteA {  evalRBACH[prescribeh, reqAustinW ] = Deny }
check evalRbachAustinWriteA expect 0

--        Expect to get Deny for Austin Write request  - so counter example expected
assert evalRbachAustinWriteB {  evalRBACH[prescribeh, reqAustinW ] = Permit }
check evalRbachAustinWriteB expect 1

--        Expect to get Permit for Austin Read request - so no counter example expected
assert evalRbachAustinReadA {  evalRBACH[prescribeh, reqAustinR ] = Permit }
check evalRbachAustinReadA expect 0

--        Expect to get Deny for Triumph Write request - so no counter example expected
assert evalRbachTriumphWriteA {  evalRBACH[prescribeh, reqTriumphW ] = Deny }
check evalRbachTriumphWriteA expect 0

--        Expect to get Deny for Austin Write request  - so counter example expected
assert evalRbachTriumphWriteB {  evalRBACH[prescribeh, reqTriumphW ] = Permit }
check evalRbachTriumphWriteB expect 1

--        Expect to get Permit for Triumph Read request - so no counter example expected
assert evalRbachTriumphReadA {  evalRBACH[prescribeh, reqTriumphR ] = Permit }
check evalRbachTriumphReadA expect 0

--    Check the generated XACML
--        Expect to fail   Austin Write = Permit
assert evalXacmlAustinWriteA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqAustinW ] = Permit )
}
check evalXacmlAustinWriteA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Austin Write =Deny or  NotApplicable
assert evalXacmlAustinWriteB {
    all x: XACML | validXACML[prescribeh, x] =>
        (evalXACML[ x , reqAustinW ] =  Deny || evalXACML[ x , reqAustinW ] = NotApplicable )
```

```
}
check evalXacmlAustinWriteB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to pass   Austin Read = Permit
assert evalXacmlAustinReadA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqAustinR ] = Permit )
}
check evalXacmlAustinReadA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to fail   Triumph Write = Permit
assert evalXacmlTriumphWriteA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqTriumphW ] = Permit )
}
check evalXacmlTriumphWriteA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Triumph Write = NotApplicable
assert evalXacmlTriumphWriteB {
    all x: XACML | validXACML[prescribeh, x] =>
        (evalXACML[ x , reqTriumphW ] = Deny || evalXACML[ x , reqTriumphW ] = NotApplicable )
}
check evalXacmlTriumphWriteB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to pass   Triumph Read = Permit
assert evalXacmlTriumphReadA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqTriumphR ] = Permit )
}
check evalXacmlTriumphReadA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

-- 2) Morris and Rove can read and write
--    Check the RBACH instance first
--        Expect to get Permit for Morris Write request - so no counter example expected
assert evalRbachMorrisWriteA {  evalRBACH[prescribeh, reqMorrisW ] = Permit }
check evalRbachMorrisWriteA expect 0

--        Expect to get Permit for Morris Write request  - so counter example expected
assert evalRbachMorrisWriteB {  evalRBACH[prescribeh, reqMorrisW ] = Deny }
check evalRbachMorrisWriteB expect 1

--        Expect to get Permit for Morris Read request - so no counter example expected
assert evalRbachMorrisReadA {  evalRBACH[prescribeh, reqMorrisR ] = Permit }
check evalRbachMorrisReadA expect 0

--        Expect to get Permit for Rover Write request - so no counter example expected
assert evalRbachRoverWriteA {  evalRBACH[prescribeh, reqRoverW ] = Permit }
check evalRbachRoverWriteA expect 0

--        Expect to get Permit for Rover Write request  - so counter example expected
assert evalRbachRoverWriteB {  evalRBACH[prescribeh, reqRoverW ] = Deny }
check evalRbachRoverWriteB expect 1
```

```
--        Expect to get Permit for Rover Read request - so no counter example expected
assert evalRbachRoverReadA {  evalRBACH[prescribeh, reqRoverR ] = Permit }
check evalRbachRoverReadA expect 0

--    Check the generated XACML
--        Expect to fail Morris Write = Deny  || NotApplicable
assert evalXacmlMorrisWriteA {
    all x: XACML | validXACML[prescribeh, x] =>
        ( evalXACML[ x , reqMorrisW ] = Deny  || evalXACML[ x , reqMorrisW ] = NotApplicable )
}
check evalXacmlMorrisWriteA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Morris Write = Permit
assert evalXacmlMorrisWriteB {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqMorrisW ] = Permit )
}
check evalXacmlMorrisWriteB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to pass   Morris Read = Permit
assert evalXacmlMorrisReadA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqMorrisR ] = Permit )
}
check evalXacmlMorrisReadA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to fail Rover Write = Deny  || NotApplicable
assert evalXacmlRoverWriteA {
    all x: XACML | validXACML[prescribeh, x] =>
        ( evalXACML[ x , reqRoverW ] = Deny || evalXACML[ x , reqRoverW ] = NotApplicable)
}
check evalXacmlRoverWriteA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Rover Write = Permit
assert evalXacmlRoverWriteB {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqRoverW ] = Permit )
}
check evalXacmlRoverWriteB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to pass   Rover Read = Permit
assert evalXacmlRoverReadA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqRoverR ] = Permit )
}
check evalXacmlRoverReadA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

-- 3) Add an extra users to test that users outside the access control policy are always refused

one sig Jaguar extends User{}
```

```
-- Request Jaguar Read
one sig reqJaguarR extends Request { }
{ u = Jaguar && p = readPrescription }

-- Request Jaguar write
one sig reqJaguarW extends Request { }
{ u = Jaguar && p = writePrescription }

--     Test the RBACH instance
--        Expect to get Deny for Jaguar Write request - so no counter example expected
assert evalRbachJaguarWriteA {  evalRBACH[prescribeh, reqJaguarW ] = Deny }
check evalRbachJaguarWriteA expect 0

--        Expect to get Deny for Jaguar Write request  - so counter example expected
assert evalRbachJaguarWriteB {  evalRBACH[prescribeh, reqJaguarW ] = Permit }
check evalRbachJaguarWriteB expect 1

--        Expect to get Deny for Jaguar Read request - so no counter example expected
assert evalRbachJaguarReadA {  evalRBACH[prescribeh, reqJaguarR ] = Deny }
check evalRbachJaguarReadA expect 0

--        Expect to get Deny for Jaguar Read request  - so counter example expected
assert evalRbachJaguarReadB {  evalRBACH[prescribeh, reqJaguarR ] = Permit }
check evalRbachJaguarReadB expect 1

--    Test the generated XACML
--        Expect to fail - Jaguar Write = Permit
assert evalXacmlJaguarWriteA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqJaguarW ] = Permit )
}
check evalXacmlJaguarWriteA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Jaguar Write =Deny or NotApplicable
assert evalXacmlJaguarWriteB {
    all x: XACML | validXACML[prescribeh, x] =>
        (evalXACML[ x , reqJaguarW ] =  Deny || evalXACML[ x , reqJaguarW ] = NotApplicable )
}
check evalXacmlJaguarWriteB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

--        Expect to fail   Jaguar Read = Permit
assert evalXacmlJaguarReadA {
    all x: XACML | validXACML[prescribeh, x] => ( evalXACML[ x , reqJaguarR] = Permit )
}
check evalXacmlJaguarReadA for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 1

--        Expect to Pass   Jaguar Read = Deny or NotApplicable
assert evalXacmlJaguarReadB {
    all x: XACML | validXACML[prescribeh, x] =>
        (evalXACML[ x , reqJaguarR] =  Deny || evalXACML[ x , reqJaguarR ] = NotApplicable )
}
check evalXacmlJaguarReadB for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy,
```

```
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0
check evalXacmlJaguarReadB for 6 but 10 XACMLelementBase , 1 XACML , 1 Hierarchy,
      2 PRMSBase , 1 Core, 2 Action, 1 Resource expect 0

-- 4) Test that access control decisions are consistent
--    Assert that for all requests the access control decision given by the XACML generated by
--    translating prescribeh is equivelant to the access control decision given by prescribeh for
--    the same request
--    (i.e) RBACH Permit = XACML Permit
--          RBAC Deny = XACML Deny OR XACML NotApplicable

-- A function to return a set containig all the possible requests that valid users can make
fun allRequests ( ) : Request {
    { req : Request | some usr : User | some act : Action | some res : Resource |
        req.u = usr && req.p.action = act && req.p.resource = res }
}

-- Check all access control decisions are equivelant
assert evalTransAllEquivelant{
    all x: XACML | validXACML[prescribeh, x] =>
        all req : allRequests |  (
            ( evalRBACH[prescribeh,req] = Permit && evalXACML[x,req] = Permit ) ||
            ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = Deny ) ||
            ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = NotApplicable ) )
}
check evalTransAllEquivelant for 6 but 10 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy, 2 PRMSBase ,
      1 Core, 2 Action, 1 Resource expect 0
check evalTransAllEquivelant for 8 but 14 XACMLelementBase , 6 int , 1 XACML , 1 Hierarchy, 2 PRMSBase ,
      1 Core, 2 Action, 1 Resource expect 0
check evalTransAllEquivelant for 6 but 10 XACMLelementBase , 1 XACML , 1 Hierarchy, 2 PRMSBase , 1 Core,
      2 Action, 1 Resource expect 0

-- Check that a counter example occurs if NotApplicable is excluded from the equilvance
assert evalTransAllEquivelantFail{
    all x: XACML | validXACML[prescribeh, x]  =>
        all req : allRequests[] | (
            ( evalRBACH[prescribeh,req] = Permit && evalXACML[x,req] = Permit ) ||
            ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = Deny )  )
}
check evalTransAllEquivelantFail for 12  but 6 int expect 1
check evalTransAllEquivelantFail for 14  but 6 int  expect 1

-- Use a predicate to generate a valid instance
pred TranslationVaild (x : XACML) {
    validXACML[prescribeh, x] &&
    ( all req : allRequests[ ] | (
        ( evalRBACH[prescribeh,req] = Permit && evalXACML[x,req] = Permit ) ||
        ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = Deny ) ||
        ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = NotApplicable ) )
    )
}
run TranslationVaild  for 10 but 24 XACMLelementBase , 8 int , 1 XACML , 1 Hierarchy , 1 Core,
    2 Action, 1 Resource expect 1
run TranslationVaild  for 10 but 24 XACMLelementBase , 1 XACML , 1 Hierarchy , 1 Core, 2 Action,
```

```
    1 Resource expect 1

-- Use a predicate to generate to check no instance is generated when NotApplicable is ignored
pred TranslationVaildFail (x : XACML) {
    validXACML[prescribeh, x] &&
    ( all req : allRequests[] | (
        ( evalRBACH[prescribeh,req] = Permit && evalXACML[x,req] = Permit ) ||
        ( evalRBACH[prescribeh,req] = Deny && evalXACML[x,req] = Deny ) )
    )
}
run TranslationVaildFail  for 10 but 24 XACMLelementBase , 8 int , 1 XACML , 1 Hierarchy, 2 PRMSBase,
    1 Core, 2 Action, 1 Resource expect 0
run TranslationVaildFail  for 10 but 24 XACMLelementBase , 1 XACML , 1 Hierarchy, 2 PRMSBase , 1 Core,
    2 Action, 1 Resource expect 0
```

C.3 Execution results

```
Executing "Run ruexist for 5 but 1 Rule expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF
   23447 vars. 1235 primary vars. 55514 clauses. 173ms.
   Instance found. Predicate is consistent, as expected. 16ms.

Executing "Run rpexist for 5 but 10 XACMLelementBase expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF
   48495 vars. 1895 primary vars. 122242 clauses. 343ms.
   Instance found. Predicate is consistent, as expected. 51ms.

Executing "Run rrpsexist for 5 but 10 XACMLelementBase expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF
   48552 vars. 1895 primary vars. 122349 clauses. 348ms.
   Instance found. Predicate is consistent, as expected. 92ms.

Executing "Run urpsexist for 5 but 10 XACMLelementBase expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF
   48552 vars. 1895 primary vars. 122349 clauses. 362ms.
   Instance found. Predicate is consistent, as expected. 63ms.

Executing "Run urpsexist1 for 5 but 10 XACMLelementBase expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF
   48552 vars. 1895 primary vars. 122349 clauses. 355ms.
   Instance found. Predicate is consistent, as expected. 81ms.

Executing "Run xacmlconvexist for 6 but 8 int, 12 XACMLelementBase expect 1"
   Solver=minisat(jni) Bitwidth=8 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   76903 vars. 2691 primary vars. 200456 clauses. 613ms.
   Instance found. Predicate is consistent, as expected. 89ms.

Executing "Check evalRbachAustinWriteA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17349 clauses. 54ms.
   No counterexample found. Assertion may be valid, as expected. 4ms.

Executing "Check evalRbachAustinWriteB expect 1"
```

```
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16679 clauses. 45ms.
   Counterexample found. Assertion is invalid, as expected. 13ms.

Executing "Check evalRbachAustinReadA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 53ms.
   No counterexample found. Assertion may be valid, as expected. 4ms.

Executing "Check evalRbachTriumphWriteA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17349 clauses. 57ms.
   No counterexample found. Assertion may be valid, as expected. 3ms.

Executing "Check evalRbachTriumphWriteB expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16679 clauses. 53ms.
   Counterexample found. Assertion is invalid, as expected. 9ms.

Executing "Check evalRbachTriumphReadA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 53ms.
   No counterexample found. Assertion may be valid, as expected. 3ms.

Executing "Check evalXacmlAustinWriteA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33731 vars. 1276 primary vars. 80949 clauses. 223ms.
   Counterexample found. Assertion is invalid, as expected. 56ms.

Executing "Check evalXacmlAustinWriteB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35204 vars. 1276 primary vars. 83576 clauses. 237ms.
   No counterexample found. Assertion may be valid, as expected. 237ms.

Executing "Check evalXacmlAustinReadA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 221ms.
   No counterexample found. Assertion may be valid, as expected. 140ms.

Executing "Check evalXacmlTriumphWriteA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33731 vars. 1276 primary vars. 80949 clauses. 211ms.
   Counterexample found. Assertion is invalid, as expected. 50ms.

Executing "Check evalXacmlTriumphWriteB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35204 vars. 1276 primary vars. 83576 clauses. 220ms.
   No counterexample found. Assertion may be valid, as expected. 247ms.
```

```
Executing "Check evalXacmlTriumphReadA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 221ms.
   No counterexample found. Assertion may be valid, as expected. 143ms.

Executing "Check evalRbachMorrisWriteA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 57ms.
   No counterexample found. Assertion may be valid, as expected. 4ms.

Executing "Check evalRbachMorrisWriteB expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16744 clauses. 42ms.
   Counterexample found. Assertion is invalid, as expected. 9ms.

Executing "Check evalRbachMorrisReadA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 46ms.
   No counterexample found. Assertion may be valid, as expected. 3ms.

Executing "Check evalRbachRoverWriteA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 56ms.
   No counterexample found. Assertion may be valid, as expected. 2ms.

Executing "Check evalRbachRoverWriteB expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16744 clauses. 46ms.
   Counterexample found. Assertion is invalid, as expected. 11ms.

Executing "Check evalRbachRoverReadA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17284 clauses. 47ms.
   No counterexample found. Assertion may be valid, as expected. 4ms.

Executing "Check evalXacmlMorrisWriteA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33734 vars. 1276 primary vars. 80994 clauses. 219ms.
   Counterexample found. Assertion is invalid, as expected. 43ms.

Executing "Check evalXacmlMorrisWriteB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 226ms.
   No counterexample found. Assertion may be valid, as expected. 143ms.

Executing "Check evalXacmlMorrisReadA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 223ms.
   No counterexample found. Assertion may be valid, as expected. 154ms.
```

```
Executing "Check evalXacmlRoverWriteA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33734 vars. 1276 primary vars. 80994 clauses. 217ms.
   Counterexample found. Assertion is invalid, as expected. 78ms.

Executing "Check evalXacmlRoverWriteB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 235ms.
   No counterexample found. Assertion may be valid, as expected. 158ms.

Executing "Check evalXacmlRoverReadA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35201 vars. 1276 primary vars. 83531 clauses. 229ms.
   No counterexample found. Assertion may be valid, as expected. 146ms.

Executing "Check evalRbachJaguarWriteA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17349 clauses. 42ms.
   No counterexample found. Assertion may be valid, as expected. 3ms.

Executing "Check evalRbachJaguarWriteB expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16679 clauses. 42ms.
   Counterexample found. Assertion is invalid, as expected. 8ms.

Executing "Check evalRbachJaguarReadA expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   8025 vars. 530 primary vars. 17349 clauses. 46ms.
   No counterexample found. Assertion may be valid, as expected. 2ms.

Executing "Check evalRbachJaguarReadB expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF
   7679 vars. 530 primary vars. 16679 clauses. 48ms.
   Counterexample found. Assertion is invalid, as expected. 9ms.

Executing "Check evalXacmlJaguarWriteA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33731 vars. 1276 primary vars. 80949 clauses. 224ms.
   Counterexample found. Assertion is invalid, as expected. 41ms.

Executing "Check evalXacmlJaguarWriteB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35204 vars. 1276 primary vars. 83576 clauses. 228ms.
   No counterexample found. Assertion may be valid, as expected. 57ms.

Executing "Check evalXacmlJaguarReadA for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=OFF
   33731 vars. 1276 primary vars. 80949 clauses. 228ms.
```

```
   Counterexample found. Assertion is invalid, as expected. 57ms.

Executing "Check evalXacmlJaguarReadB for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35204 vars. 1276 primary vars. 83576 clauses. 235ms.
   No counterexample found. Assertion may be valid, as expected. 52ms.

Executing "Check evalXacmlJaguarReadB for 6 but 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
   35204 vars. 1276 primary vars. 83356 clauses. 228ms.
   No counterexample found. Assertion may be valid, as expected. 48ms.

Executing "Check evalTransAllEquivelant for 6 but 6 int, 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=6 SkolemDepth=1 Symmetry=20
   36284 vars. 1288 primary vars. 89834 clauses. 249ms.
   No counterexample found. Assertion may be valid, as expected. 474ms.

Executing "Check evalTransAllEquivelant for 8 but 6 int, 14 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=8 SkolemDepth=1 Symmetry=20
   70337 vars. 2072 primary vars. 181780 clauses. 492ms.
   No counterexample found. Assertion may be valid, as expected. 36197ms.

Executing "Check evalTransAllEquivelant for 6 but 10 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
   36284 vars. 1288 primary vars. 89614 clauses. 246ms.
   No counterexample found. Assertion may be valid, as expected. 382ms.

Executing "Check evalTransAllEquivelantFail for 12 but 6 int expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=12 SkolemDepth=1 Symmetry=OFF
   369280 vars. 8889 primary vars. 954584 clauses. 2726ms.
   Counterexample found. Assertion is invalid, as expected. 653ms.

Executing "Check evalTransAllEquivelantFail for 14 but 6 int expect 1"
   Solver=minisat(jni) Bitwidth=6 MaxSeq=14 SkolemDepth=1 Symmetry=OFF
   633536 vars. 13231 primary vars. 1643907 clauses. 5021ms.
   Counterexample found. Assertion is invalid, as expected. 784ms.

Executing "Run TranslationVaild1 for 10 but 8 int, 24 XACMLelementBase, 1 XACML, 1 Hierarchy,
           1 Core, 2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=8 MaxSeq=10 SkolemDepth=1 Symmetry=OFF
   219248 vars. 4674 primary vars. 600739 clauses. 2678ms.
   Instance found. Predicate is consistent, as expected. 1082ms.

Executing "Run TranslationVaild1 for 10 but 24 XACMLelementBase, 1 XACML, 1 Hierarchy, 1 Core,
           2 Action, 1 Resource expect 1"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=OFF
   218840 vars. 4674 primary vars. 596827 clauses. 2762ms.
   Instance found. Predicate is consistent, as expected. 339ms.
```

```
Executing "Run TranslationVaildFail for 10 but 8 int, 24 XACMLelementBase, 1 XACML, 1 Hierarchy,
           2 PRMSBase, 1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=8 MaxSeq=10 SkolemDepth=1 Symmetry=20
   189003 vars. 4266 primary vars. 505176 clauses. 1684ms.
   No instance found. Predicate may be inconsistent, as expected. 1150ms.

Executing "Run TranslationVaildFail for 10 but 24 XACMLelementBase, 1 XACML, 1 Hierarchy, 2 PRMSBase,
           1 Core, 2 Action, 1 Resource expect 0"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   188619 vars. 4266 primary vars. 501840 clauses. 1645ms.
   No instance found. Predicate may be inconsistent, as expected. 101355ms.

48 commands were executed. The results are:
   #1: Instance found. ruexist is consistent, as expected.
   #2: Instance found. rpexist is consistent, as expected.
   #3: Instance found. rrpsexist is consistent, as expected.
   #4: Instance found. urpsexist is consistent, as expected.
   #5: Instance found. urpsexist1 is consistent, as expected.
   #6: Instance found. xacmlconvexist is consistent, as expected.
   #7: No counterexample found. evalRbachAustinWriteA may be valid, as expected.
   #8: Counterexample found. evalRbachAustinWriteB is invalid, as expected.
   #9: No counterexample found. evalRbachAustinReadA may be valid, as expected.
   #10: No counterexample found. evalRbachTriumphWriteA may be valid, as expected.
   #11: Counterexample found. evalRbachTriumphWriteB is invalid, as expected.
   #12: No counterexample found. evalRbachTriumphReadA may be valid, as expected.
   #13: Counterexample found. evalXacmlAustinWriteA is invalid, as expected.
   #14: No counterexample found. evalXacmlAustinWriteB may be valid, as expected.
   #15: No counterexample found. evalXacmlAustinReadA may be valid, as expected.
   #16: Counterexample found. evalXacmlTriumphWriteA is invalid, as expected.
   #17: No counterexample found. evalXacmlTriumphWriteB may be valid, as expected.
   #18: No counterexample found. evalXacmlTriumphReadA may be valid, as expected.
   #19: No counterexample found. evalRbachMorrisWriteA may be valid, as expected.
   #20: Counterexample found. evalRbachMorrisWriteB is invalid, as expected.
   #21: No counterexample found. evalRbachMorrisReadA may be valid, as expected.
   #22: No counterexample found. evalRbachRoverWriteA may be valid, as expected.
   #23: Counterexample found. evalRbachRoverWriteB is invalid, as expected.
   #24: No counterexample found. evalRbachRoverReadA may be valid, as expected.
   #25: Counterexample found. evalXacmlMorrisWriteA is invalid, as expected.
   #26: No counterexample found. evalXacmlMorrisWriteB may be valid, as expected.
   #27: No counterexample found. evalXacmlMorrisReadA may be valid, as expected.
   #28: Counterexample found. evalXacmlRoverWriteA is invalid, as expected.
   #29: No counterexample found. evalXacmlRoverWriteB may be valid, as expected.
   #30: No counterexample found. evalXacmlRoverReadA may be valid, as expected.
   #31: No counterexample found. evalRbachJaguarWriteA may be valid, as expected.
   #32: Counterexample found. evalRbachJaguarWriteB is invalid, as expected.
   #33: No counterexample found. evalRbachJaguarReadA may be valid, as expected.
   #34: Counterexample found. evalRbachJaguarReadB is invalid, as expected.
   #35: Counterexample found. evalXacmlJaguarWriteA is invalid, as expected.
   #36: No counterexample found. evalXacmlJaguarWriteB may be valid, as expected.
   #37: Counterexample found. evalXacmlJaguarReadA is invalid, as expected.
   #38: No counterexample found. evalXacmlJaguarReadB may be valid, as expected.
   #39: No counterexample found. evalXacmlJaguarReadB may be valid, as expected.
   #40: No counterexample found. evalTransAllEquivelant may be valid, as expected.
   #41: No counterexample found. evalTransAllEquivelant may be valid, as expected.
```

```
#42: No counterexample found. evalTransAllEquivelant may be valid, as expected.
#43: Counterexample found. evalTransAllEquivelantFail is invalid, as expected.
#44: Counterexample found. evalTransAllEquivelantFail is invalid, as expected.
#45: Instance found. TranslationVaild1 is consistent, as expected.
#46: Instance found. TranslationVaild1 is consistent, as expected.
#47: No instance found. TranslationVaildFail may be inconsistent, as expected.
#48: No instance found. TranslationVaildFail may be inconsistent, as expected.
```

Appendix D

Translation Tool

D.1 Additional constraints

```
-- Constraints

fact EverybodyHasARole {
  all h : Hierarchy | all u : h.USERS | some u.(h.UA)
}

fun fun_EverybodyHasARole() : Hierarchy -> User {
  { h : Hierarchy , u : User | u in h.USERS && no u.(h.UA) }
}

fact EverybodyCanDoSomething {
  all h : Hierarchy | all u : h.USERS | some u.(h.UA).*(h.RH).(h.PA)
}

fun fun_EverybodyCanDoSomething() : Hierarchy -> User {
  { h : Hierarchy , u : User | u in h.USERS && no u.(h.UA).*(h.RH).(h.PA) }
}

fact NobodyHasAllRoles {
  all h : Hierarchy | all u : h.USERS | u.(h.UA) != h.ROLES
}

fun fun_NobodyHasAllRoles() : Hierarchy -> User {
  { h : Hierarchy , u : User | u in h.USERS && u.(h.UA) = h.ROLES }
}

fact NobodyCanDoEverything {
  all h : Hierarchy | all u : h.USERS | u.(h.UA).*(h.RH).(h.PA) != h.PRMS
}

fun fun_NobodyCanDoEverything() : Hierarchy -> User {
  { h : Hierarchy , u : User |
    u in h.USERS && u.(h.UA).*(h.RH).(h.PA) = h.PRMS }
}
```

```
fact NoRedundantPermissions {
  all h : Hierarchy | all r : h.ROLES | no (r.(h.PA) & r.^(h.RH).(h.PA))
}

fun fun_NoRedundantPermissions() : Hierarchy -> Role -> PRMSBase {
  { h : Hierarchy , r : Role , p : PRMSBase |
    r in h.ROLES && p in h.PRMS && p in (r.(h.PA) & r.^(h.RH).(h.PA)) }
}

fact AllRolesHaveAPermission {
  all h : Hierarchy | all r : h.ROLES | some r.*(h.RH).(h.PA)
}

fun fun_AllRolesHaveAPermission() : Hierarchy -> Role {
  { h : Hierarchy , r : Role | r in h.ROLES && no r.*(h.RH).(h.PA) }
}

fact NobodyBreachesSMER {
  all h : Hierarchy |
    all s : h.SC | all u : h.USERS | #(s.roles & u.(h.UA).*(h.RH)) < s.limit
}

fun fun_NobodyBreachesSMER() : Hierarchy -> SMER ->  User {
  { h : Hierarchy , s : SMER , u : User |
    s in h.SC && u in h.USERS && #(s.roles & u.(h.UA).*(h.RH)) >= s.limit }
}

fact NoSingleRoleBreachesSMER {
  all h : Hierarchy |
    all s : h.SC | all r : h.ROLES | #(s.roles & r.*(h.RH)) < s.limit
}

fun fun_NoSingleRoleBreachesSMER() : Hierarchy -> SMER ->  Role {
  { h : Hierarchy , s : SMER , r : Role |
    s in h.SC && r in h.ROLES && #(s.roles & r.*(h.RH)) >= s.limit }
}

fun ImmediateSuccessor(RH : Role -> Role) : Role -> Role  {
  let succ = *RH |
    { disj r1 , r2 : Role | r1 -> r2 in succ &&
      (no r3 : Role - (r1 + r2) | r1 -> r3 in succ && r3 -> r2 in succ) }
}

fact UpwardLimitedHierachy {
  all h : Hierarchy |
    let imm = ImmediateSuccessor[h.RH] |
      no disj r1 , r2 , r3 :  Role | (r2 + r3) -> r1 in imm
}

fun fun_UpwardLimitedHierachy() : Hierarchy -> Role {
  { h : Hierarchy , r1 : Role | let imm = ImmediateSuccessor[h.RH] |
    r1 in h.ROLES && some disj r2 , r3 : Role | (r2 + r3) -> r1 in imm }
}
```

```
fact DownwardLimitedHierachy {
  all h : Hierarchy | let imm = ImmediateSuccessor[h.RH] |
    no disj r1 , r2 , r3 :  Role | r1 -> (r2 + r3) in imm
}

fun fun_DownwardLimitedHierachy() : Hierarchy -> Role {
  { h : Hierarchy , r1 : Role | let imm = ImmediateSuccessor[h.RH] |
    r1 in h.ROLES && some disj r2 , r3 : Role | r1 -> (r2 + r3) in imm }
}

fact AllPermissionsReachable {
  all h : Hierarchy | (h.USERS).(h.UA).*(h.RH).(h.PA) = h.PRMS
}

fun fun_AllPermissionsReachable() : Hierarchy -> PRMSBase {
  { h : Hierarchy , p : PRMSBase |
    p in h.PRMS && p !in (h.USERS).(h.UA).*(h.RH).(h.PA) }
}
```

D.2 Validation of the running example

```
Failed Optional Facts:
NobodyCanDoEverything - {examplerbac->morris, examplerbac->rover}
Passed Optional Facts:
AllPermissionsReachable
AllRolesHaveAPermission
DownwardLimitedHierachy
EverybodyCanDoSomething
EverybodyHasARole
NoRedundantPermissions
NoSingleRoleBreachesSMER
NobodyBreachesSMER
NobodyHasAllRoles
UpwardLimitedHierachy
uniquePRMSBase
```

D.3 Comparing the RBAC policy with the XACML translation of the running example

```
Test Results :
User            | Action          | Resource             | RBAC         | XACML                | SameResult
--------------- | --------------- | -------------------- | ------------ | -------------------- | --------------
morris          | write           | PrescribeDB          | Permit       | Permit               | true
morris          | read            | PrescribeDB          | Permit       | Permit               | true
rover           | write           | PrescribeDB          | Permit       | Permit               | true
rover           | read            | PrescribeDB          | Permit       | Permit               | true
triumph         | write           | PrescribeDB          | Deny         | NotApplicable        | true
triumph         | read            | PrescribeDB          | Permit       | Permit               | true
austin          | write           | PrescribeDB          | Deny         | NotApplicable        | true
austin          | read            | PrescribeDB          | Permit       | Permit               | true
8 :: 8 --- 0
```

D.4 The XACML translation of the running example

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
  access_control-xacml-2.0-policy-schema-os.xsd"
  PolicySetId=""
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

  <Target/>

  <PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
      access_control-xacml-2.0-policy-schema-os.xsd"
    PolicySetId="RU:Nurse"
    PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >austin</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >triumph</AttributeValue>
              <SubjectAttributeDesignator
                 DataType="http://www.w3.org/2001/XMLSchema#string"
                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
      </Target>
      <PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
        xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
          access_control-xacml-2.0-policy-schema-os.xsd"
        PolicySetId="RR:Nurse"
        PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

        <Target/>

        <Policy  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
          xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
          access_control-xacml-2.0-policy-schema-os.xsd"
        PolicyId="RP:Nurse:read.PrescribeDB"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">

        <Target/>

        <Rule RuleId="R:read.PrescribeDB" Effect="Permit">
          <Target>
            <Actions>
              <Action>
                <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                    >read</AttributeValue>
                  <ActionAttributeDesignator
                    DataType="http://www.w3.org/2001/XMLSchema#string"
                    AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
                </ActionMatch>
              </Action>
            </Actions>
            <Resources>
              <Resource>
                <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                    >PrescribeDB</AttributeValue>
                  <ResourceAttributeDesignator
                    DataType="http://www.w3.org/2001/XMLSchema#string"
                    AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
                </ResourceMatch>
              </Resource>
            </Resources>
          </Target>
          <Description> </Description>
        </Rule>
      </Policy>
    </PolicySet>
</PolicySet>

<PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
    access_control-xacml-2.0-policy-schema-os.xsd"
  PolicySetId="RU:Doctor"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
            >morris</AttributeValue>
          <SubjectAttributeDesignator
```

```
          DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
      </SubjectMatch>
    </Subject>
    <Subject>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
          >rover</AttributeValue>
        <SubjectAttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
</Target>
<PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
    access_control-xacml-2.0-policy-schema-os.xsd"
  PolicySetId="RR:Doctor"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

  <Target/>

  <Policy  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
      access_control-xacml-2.0-policy-schema-os.xsd"
    PolicyId="RP:Doctor:write.PrescribeDB"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">

    <Target/>

    <Rule RuleId="R:write.PrescribeDB" Effect="Permit">
      <Target>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >write</AttributeValue>
               <ActionAttributeDesignator
                 DataType="http://www.w3.org/2001/XMLSchema#string"
                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >PrescribeDB</AttributeValue>
              <ResourceAttributeDesignator
```

```
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
          </ResourceMatch>
        </Resource>
      </Resources>
    </Target>
    <Description> </Description>
  </Rule>
</Policy>
<PolicySet  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
    access_control-xacml-2.0-policy-schema-os.xsd"
  PolicySetId="RR:Nurse"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides">

  <Target/>

  <Policy  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xacml-context="urn:oasis:names:tc:xacml:2.0:context:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
      access_control-xacml-2.0-policy-schema-os.xsd"
    PolicyId="RP:Nurse:read.PrescribeDB"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">

    <Target/>

    <Rule RuleId="R:read.PrescribeDB" Effect="Permit">
      <Target>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >read</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string"
                >PrescribeDB</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
      </Target>
```

```
            <Description> </Description>
          </Rule>
        </Policy>
      </PolicySet>
    </PolicySet>
  </PolicySet>
</PolicySet>
```

Appendix E

Case study

E.1 A Z representation of the case study policy

$$USERS = \{\ gillian, bernadette, savannah, stephan, eleanor, hayley, darius, tammie, dean, conner, ayanna, danielle, samantha, dale, larry, dirk, felix, jace, giovanni, leslie, precious, kaitlyn\ \}$$

$$ROLES = \{\ dr2, dr, cons, nu, dr1, sres, resp2, stat1, sec, adm, stat2, resp1\ \}$$

$$PRMS = \{\ (write, ano1), (read, res2), (read, medObs), (read, ano2), (write, dem), (read, ano1), (write, res1), (read, res1), (read, man), (write, man), (read, dem), (write, medObs), (write, res2), (read, pres), (write, ano2), (write, pres)\ \}$$

$$UA = \{\ (samantha, dr), (dale, dr1), (ayanna, nu), (tammie, dr), (darius, stat2), (stephan, nu), (dirk, sres), (conner, dr), (bernadette, dr2), (dean, nu), (giovanni, sec), (felix, sec), (precious, resp2), (savannah, sec), (leslie, adm), (jace, resp2), (larry, nu), (gillian, stat1), (hayley, nu), (danielle, resp1), (kaitlyn, resp1), (eleanor, cons)\ \}$$

$$RH = \{\ (dr1, dr), (sres, resp2), (cons, dr1), (cons, dr), (nu, sec), (resp1, stat1), (resp2, stat2), (dr2, dr), (adm, sec), (cons, dr2), (dr, nu), (sres, resp1)\ \}$$

$$PA = \{\ (resp1, (read, res1)), (stat2, (read, ano2)), (adm, (read, man)), (dr2, (write, res2)), (dr2, (read, res2)), (resp2, (read, res2)), (stat1, (read, ano1)), (sec, (read, dem)), (cons, (read, man)), (nu, (read, medObs)), (sec, (write, dem)), (nu, (write, medObs)), (dr, (write, pres)), (nu, (read, pres)), (dr1, (read, res1)), (adm, (write, man)), (resp1, (write, res1)), (resp2, (write, res2)), (resp2, (write, ano2)), (resp1, (write, ano1)), (dr1, (write, res1))\ \}$$

E.2 An Alloy representation of the case study policy

```
one sig write, read extends Action {}

one sig Pres, medObs, ano2, ano1, res2, res1, dem, man extends Resource {}

one sig nu, resp2, resp1, stat1, stat2, dr2, dr1, sres, adm, dr, cons, sec extends Role {}

one sig danielle, dean, samantha, savannah, tammie, leslie, jace, dale, gillian, larry, precious, felix,
        giovanni, stephan, hayley, eleanor, kaitlyn, dirk, darius, bernadette, conner, ayanna extends User {}

one sig wMedObs extends PRMSBase {} { resource = medObs && action = write }
```

```
one sig rPres extends PRMSBase {} { resource = Pres && action = read }

one sig rAno1 extends PRMSBase {} { resource = ano1 && action = read }

one sig wAno1 extends PRMSBase {} { resource = ano1 && action = write }

one sig wAno2 extends PRMSBase {} { resource = ano2 && action = write }

one sig rAno2 extends PRMSBase {} { resource = ano2 && action = read }

one sig rDem extends PRMSBase {} { resource = dem && action = read }

one sig wRes1 extends PRMSBase {} { resource = res1 && action = write }

one sig wRes2 extends PRMSBase {} { resource = res2 && action = write }

one sig rRes1 extends PRMSBase {} { resource = res1 && action = read }

one sig rRes2 extends PRMSBase {} { resource = res2 && action = read }

one sig wPres extends PRMSBase {} { resource = Pres && action = write }

one sig wDem extends PRMSBase {} { resource = dem && action = write }

one sig rMedObs extends PRMSBase {} { resource = medObs && action = read }

one sig rMan extends PRMSBase {} { resource = man && action = read }

one sig wMan extends PRMSBase {} { resource = man && action = write }

one sig examplerbac extends Hierarchy {}
{
  UA = dean -> nu + dale -> dr1 + larry -> nu + gillian -> stat1 + precious -> resp2 + leslie -> adm +
       danielle -> resp1 + dirk -> sres + felix -> sec + giovanni -> sec + savannah -> sec + tammie -> dr +
       bernadette -> dr2 + samantha -> dr + stephan -> nu + conner -> dr + eleanor -> cons + ayanna -> nu +
       kaitlyn -> resp1 + hayley -> nu + jace -> resp2 + darius -> stat2
  RH = dr -> nu + resp1 -> stat1 + sres -> resp2 + sres -> resp1 + adm -> sec + resp2 -> stat2 + dr2 -> dr +
       nu -> sec + cons -> dr2 + cons -> dr + cons -> dr1 + dr1 -> dr
  PA = resp2 -> rRes2 + resp2 -> wRes2 + sec -> rDem + dr2 -> rRes2 + resp1 -> rRes1 + dr -> wPres + dr1 -> wRes1 +
       resp1 -> wRes1 + nu -> rMedObs + dr2 -> wRes2 + adm -> wMan + nu -> rPres + cons -> rMan + nu -> wMedObs +
       stat2 -> rAno2 + resp2 -> wAno2 + dr1 -> rRes1 + adm -> rMan + sec -> wDem + resp1 -> wAno1 + stat1 -> rAno1
  USERS = samantha + dale + eleanor + jace + dirk + savannah + giovanni + felix + gillian + tammie + larry +
          kaitlyn + ayanna + danielle + dean + stephan + darius + bernadette + conner + leslie + precious +
          hayley
  ROLES = adm + nu + sec + cons + dr1 + dr2 + stat1 + stat2 + sres + resp2 + resp1 + dr
  PRMS = wRes1 + wAno1 + wAno2 + rAno2 + rAno1 + rRes1 + rPres + rRes2 + wRes2 + wPres + wMan + rMedObs + rMan +
         rDem + wMedObs + wDem
}
```

Glossary

ACL Access Control List

DAC Discretionary Access Control

DSD Dynamic Separation of Duty

GIMI Generic Infrastructure for Medical Informatics

MAC Mandatory Access Control

PA role-Permission Assignment

PAP Policy Access Point

PDP Policy Decision Point

PEP Policy Enforcement Point

PIP Policy Information Point

RBAC Role-Based Access Control

RH Role hierarchy

sif service-oriented interoperability framework

SMER Static Mutually Exclusive Roles

SOA Service Oriented Architecture

SSD Static Separation of Duty

UA User-role Assignment

VO Virtual Organisation

WS Web Services

XACML eXtensible Access Control Markup Language