Imperial College of Science, Technology and Medicine

University of London

Department of Computing

A Policy Framework for Management of Distributed Systems

Nicodemos C. Damianou

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering of the University of London, and for the Diploma of the Imperial College of Science, Technology and Medicine

London, February 2002

Στους γονείς μου, Κωνσταντίνο και Σοφία

(To my parents Constantinos and Sofia)

Abstract

Policy-based management is one of the latest developments in network and distributed systems management. Academic and commercial settings, as well as standardisation bodies are concentrating on policy-based management as a very promising solution for managing large-scale distributed systems. The use of policy-based management in areas such as security is particularly attractive. The introduction of new technologies (e.g. active networks, mobile agents) and the use of the Internet for providing services to customers, increase the security concerns associated with today's networked environments. Security management involves specification and deployment of access control policies as well as activities such as registration of users or logging and auditing events for dealing with access to critical resources or security violations. The management actions to be performed when an event occurs depend on the enterprise policy.

The need is evident for a policy language to support the specification of access control and other management policies. In this thesis we propose a policy framework to support security and management of distributed systems. The framework consists of a policy specification language, an architecture for deploying policies based on the language and a set of tools for specifying and managing policies. In conjunction with the language, the toolkit permits integrated administration of resources, people and policy information with automated policy deployment. The toolkit comprises an Integrated Development Environment (IDE) with a policy compiler, as well as tools for managing policies and roles at runtime.

The policy language is a declarative, object-oriented language for specifying security and management policies for distributed object systems. The language is flexible, expressive and extensible to cover the wide range of requirements implied by the current distributed systems paradigms. It includes support for access control policies, and delegation to cater for temporary transfer of access rights to agents acting on behalf of a client. The language also supports policies to express management activity, which take the form of event-triggered rules called obligation policies. Domains are used to facilitate the specification of policies relating to large systems with millions of objects; policies are specified for collections of objects stored in domains instead of individual objects, thus allowing for scalability and flexibility. Composite policies are included to allow the basic security and management policies relating to roles, organisational units and specific applications to be grouped together. Composite policies are essential to cater for the complexity of policy administration in large enterprise information systems. Application specific constraints on groups of policies can be specified using meta-policies. The language is easy to use by policy users, and we use a structural operational semantics approach to specify its formal semantics.

Acknowledgements

My first and foremost thanks go to my supervisor Professor Morris Sloman. This thesis would not have been possible without his inspiration, constructive criticism and experienced guidance. I thank Morris for arranging for financial support during my PhD studies. I am also grateful to Dr. Emil Lupu whose guidance, support and critical analysis were extremely important towards completion of this thesis.

Special thanks also go to Dr. Naranker Dulay who has contributed a great deal to the work presented in this thesis, with his useful comments and advice.

Many thanks to my friends and colleagues in the Distributed Software Engineering section for many stimulating and pleasant discussions. These include: Roberto, Yiannis, Tyrone, Yiorgos, Leonidas, Sebastian, Oscar, Dan, Krish, Sammy and Toshio. Thanks to all members of the DSE section for making this a memorable experience. Special thanks to Siv Sivzattian who took the time to read the thesis and provide me with invaluable comments.

My most profound gratitude goes to my family who have been the timeless source of inspiration for me to pursue a doctorate; especially my parents Constantinos and Sofia, to whom this thesis is dedicated. They have given me the opportunity to undertake this work and have been constantly supporting me over the years.

Statement of Contribution

This thesis is the result of the author's participation in work at Imperial College on distributed systems policy-based management over the last 3 years. Many of the ideas developed in this thesis are the result of group discussions with Professor Morris Sloman, Dr. Emil Lupu and Dr. Naranker Dulay.

The thesis is motivated by and based on a policy-driven management model researched by Sloman et al. The model includes management domains, management policies and role-based management ideas, which are described in Chapter 2 and credited accordingly. This thesis develops these ideas and presents a complete policy-based management framework, which includes a policy specification language and an architecture for deploying policies. The policy specification language did not exist in previous work and is a main contribution of this thesis. The same is true of the architecture for deploying and enforcing policies which includes automated policy distribution to their enforcement points, as well as dynamic adaptation to changes in the domains and managed objects of the system. The formal semantics for the policy language presented in Chapter 5 is the author's individual work.

The implementation of the management tools and components of the management framework described in this thesis are the author's individual work with the exception of the domain browser implemented by Toshio Tonouchi using the interfaces designed by the author to interact with the other management tools.

Table of Contents

List of Figures

List of Abbreviations

AC	Access Controller
ACO	Authorisation Control Object
AEO	Authorisation Enforcement Object
AST	Abstract Syntax Tree
CIM	Common Information Model
COPS	Common Open Policy Service
CORBA	Common Object Request Broker Architecture
DAC	Discretionary Access Control
DMTF	Distributed Management Task Force
DSE	Domain Scope Expression
EC	Enforcement Component
IC	Intermediate Code
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
JDMK	Java Dynamic Management Kit
JMAPI	Java Management Application Programming Interface
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
LDAP	Lightweight Directory Access Protocol
MAC	Mandatory Access Control
OCL	Object Constraint Language
OCO	Obligation Control Object
OEO	Obligation Enforcement Object
OMG	Object Management Group
ORB	Object Request Broker
PCIM	Policy Core Information Model
PCO	Policy Control Object
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PMC	Policy Management Component
RAT	Refrain Action Table
RCO	Refrain Control Object
REO	Refrain Enforcement Object
RPT	Refrain Policy Table
RBAC	Role Based Access Control
RBM	Role Based Management
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
UPO	User Profile Object
URD	User Representation Domain

Chapter 1

Introduction

Policy-based management has recently become a widely employed and promising solution for managing enterprise-wide networks and distributed systems. Such systems are driven by business needs, which require management solutions that are both self-adapting and that dynamically change the behaviour of the managed system. In today's Internet-based environments security concerns tend to increase as programmable mechanisms are introduced to enable such adaptation. Furthermore, the heterogeneity of security mechanisms used to implement access control render security management an important and difficult task. The focal point in the area of policy-based management is the notion of *policy* as a means of driving management procedures. Although the technologies for building management systems are available, work on the specification and deployment of policies is still scarce. The precise and explicit specification of implementable policies is important in order to achieve the organisational goals using currently available technologies. In this thesis we address the problem of policy specification for enterprise-wide distributed systems. We propose a policy specification language and show how this guides the design of a strongly-distributed management architecture. In this chapter, we discuss the motivation behind the ideas presented in this thesis, we identify the requirements for a policy management framework, and conclude by highlighting our contribution and presenting an outline of the structure of the thesis.

1.1 Motivation

A typical enterprise network system consists of a large number of heterogeneous network devices such as routers, and servers running a variety of applications and offering services to a large number of users. Devices, services, applications, servers and users as well as the relationships between them are all targets of management systems used to manage enterprise networks. The complexity of the managed systems results in high administrative costs and long deployment cycles for business initiatives, and imposes two requirements on their management systems. Although these requirements have long been recognised their importance is now becoming increasingly critical: (i) management must be *distributed* in order to be scalable and cope with the size of enterprise networks, and (ii) management procedures must be *automated* to reduce administrative

costs. Manual management is expensive and the effort and time needed for management increases exponentially as a system expands.

As networked systems are increasingly driven by changing business needs, their management becomes even more complex. In order to adapt to changing business requirements, distributed systems switch from the traditional client-server model to a service-driven model: "*The service-driven network is a new approach to the provision of network computing that concentrates on the services you want to provide. These services range from the low-level services that manage relationships between networked devices to the value-added services you provide to end-users.*" [Sun 1999a]. Networked environments are designed to be highly adaptable to support rapid deployment of such customised services. Thus, management also needs to be dynamic and flexible to deal with the evolution of the systems being managed.

The requirements for management systems identified above, can be facilitated with policy-based management approaches where the support for distribution, automation and dynamic adaptation of the behaviour of the managed system is achieved by using policies. The main benefits from using policy are improved scalability and flexibility for the management system. *Scalability* is improved by uniformly applying the same policy to large sets of devices and objects, while *flexibility* is achieved by separating the policy from the implementation of the managed system. Policy can be changed dynamically, thus changing the behaviour and strategy of a system, without modifying its implementation or interrupting its operation. Policy-based management is largely supported by standards organisations such as the Internet Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF), and most network equipment vendors.

As enterprises are increasingly leveraging Internet technologies to adopt e-business practices, they expose internal resources to customers, and require that enterprise-wide authorisation policies be easily established and implemented. Authorisations must be enforced both at the application level and in network elements, and must be explicit, i.e. authorisation policies must be precisely and unambiguously stated to define the set of acceptable requests. Various techniques have emerged for programming network elements to support adaptive services, such as active networks, mobile agents, and management by delegation. While these approaches support the programming of new functionality into network elements and host devices, they increase the security concerns regarding access to network resources and services. Authorisation policies must therefore specify which users are permitted to program network elements, which services users are permitted to access and under what circumstances.

A plethora of mechanisms are used within the same enterprise computing systems to provide security at different levels: application level (e.g. databases), object-middleware level, operating-system level or network level. In addition, access control is typically distributed across many

heterogeneous components which enforce authorisation policies for a variety of target objects. The proliferation of non-integrated security mechanisms and products, each with independent administration and application development interfaces, leads to separate authorisation policy implementations within each individual application and system. This makes it difficult to support global security policies in accordance with enterprise access control goals. It should be possible to provide consistent security across the distributed object system and associated legacy systems. Policy must be separated from the security mechanisms that enforce access control, in order to enable the specification and integrated administration of global policies.

Delegation is often used in access control systems to cater for temporary transfer of access rights between users or agents. Secure delegation of administrative functions is common in organisational systems where security management is delegated on a hierarchical basis. A user's ability to delegate access rights must be tightly controlled by security policies especially in those systems which allow cascaded delegation of access rights.

Management and security of distributed systems are interdependent and each needs the services of the other [Hyland et al. 1998]. Catering for security management involves not only specification and deployment of access control policies, but also providing support for user registration, logging and auditing of access to critical resources, and responding to security violations (e.g. repeated attempts to delete a sensitive file). Thus, security specifications must also cover obligation policies to assign responsibility for performing actions related to security management and enforcement. In addition, the policy system must be self-managed so that policies can be specified about who is authorised to modify other policies.

Managing security in heterogeneous, distributed environments can become expensive and error-prone; it requires security administration to be distributed to multiple policy administrators, which makes it difficult to provide consistent security policies across the entire system. It is thus necessary to analyse security policies for conflicts and inconsistencies, which may lead the system to insecure states. Analysis of policy specification is also important in order to ensure that enterprise security goals are met. In large-scale systems with large numbers of both users and policies, it must be possible to analyse the policies in order to check for the existence of policies that implement the high-level security goals of the organisation. This involves subject and target review of policy to identify which policies apply to certain subjects or target objects, and assumes two things: (i) a policy notation that can be analysed, and (ii) centralised access to deployed policies.

Security administration in large intra- and inter-enterprise systems requires management of access controls based on *roles* relating permissions to organisational positions or groups rather than individual identities. Role-based administration has proven to be a very important technique in

reducing administrative costs. In addition, the large and constantly changing population of managed objects in such systems requires policies to be specified in terms of *groups* of objects rather than individual ones. Policy specification for individual objects does not scale for large enterprise systems.

Finally, management, even when policy-based, is an evolutionary process. Policy-based resource allocation, the association between policy and the devices/entities on which it must be implemented and even the policies themselves are subject to frequent reviews and changes. Decentralised administration of large and complex organisational structures is both difficult and error prone and administrators must be isolated from the details of the underlying implementations and policy representations. This can be achieved with tools that allow for integrated administration and hide the heterogeneity of devices and the complexity of policy deployment.

In this thesis we define policy as: *A persistent declarative specification, derived from management goals, of a rule defining choices in behaviour of a system*, based on the definition of policy in [Moffett et al. 1993; Sloman 1994b]. This definition identifies various properties of policy:

- *Persistent* in the sense that a one-off command to perform an action is not a policy. In addition, policies are relatively static compared to the state of the managed system.
- *Declarative* in the sense that policies define choices in behaviour in terms of the conditions under which predefined operations or actions can be invoked rather than changing the functionality of the actual operations themselves, i.e. they specify *what* behaviour is desired, not *how* the behaviour will be achieved and maintained.
- *Derived from management goals* because we view policies as being derived from business goals, service level agreements or trust relationships. We sometimes refer to management goals as high-level or abstract policies in this thesis.

1.2 Policy Framework Requirements

In this section we identify the requirements that need to be addressed when defining a policy management framework. The main requirement is the design of a language that can be used to specify both security and management policies. However, the framework must also address the issue of policy deployment for a variety of application areas and on a variety of platforms and systems. The requirements identified in this section are derived from the experience gained within the policy group at Imperial College over the past 10 years, and from the survey of the literature on policy-based management.

1.2.1 Policy Specification Language

A lot of work within the greater scope of distributed systems management has already resulted in architectures and technologies that provide the basic infrastructure and tools required to implement policy-based management solutions. This includes mechanisms, protocols, models, management paradigms and proprietary solutions to many of the problems involved in this area. The need to integrate the various solutions to enable the engineering of integrated management systems has already been acknowledged [Hegering et al. 1999] and, in recent years, emphasis was placed on policy specification [Stone et al. 2001] and information models for managed objects. We believe that a common high-level language for different applications of policy-based management related to both security and network management is important. The following is a list of requirements that the language specification must support:

- Authorisation policies to explicitly specify the set of acceptable requests in the system. Any request made by a subject can be defined in terms of an action on an object, so authorisation policies must define the relationship between subjects and actions on target objects. Authorisation policy specification must also allow for both positive and negative authorisations in order to conveniently support exceptions [Samarati et al. 2000]. Negative authorisations are supported by many security platforms (e.g. Windows NT/2000) and can be used to temporarily remove access rights from subjects.
- Implementation-independent authorisation policies that can map onto various access control mechanisms for firewalls, operating systems, object-middleware, databases and programming languages such as Java.
- Delegation policy specification to cater for temporary transfer of access rights to agents acting on behalf of a client. Delegation policies enable decentralised privilege administration permitted within discretionary access control models to allow users to delegate their privileges to others.
- Constraint-based authorisations including temporal authorisation [Bertino et al. 1998] as well as restricting the validity of the policy based on the state of the system, or the state of the policy targets.
- Constraints on a set of policies in order to restrict the policies that can be specified in the system under certain conditions, prioritise policies, and model well-known constraints such as separation of duty, and user-role/role-permission assignment constraints.
- Event-condition-action rules to define the management actions that need to be performed periodically, or when triggered by events. These policies can be used to specify security management actions, quality of service rules, or in a more general context as a constrained form of programming network elements and end-user agents.

- Structuring techniques to facilitate the specification of policies relating to large systems with millions of objects. This requires the ability to apply policies to large collections of objects rather than specifying policy for individual objects. It should be possible to specify policies for heterogeneous target objects grouped together based on arbitrary application-specific criteria.
- Composite policies, which allow the basic security and management policies relating to roles, organisational units or specific applications to be grouped together. Composite policies are essential to cater for the complexity of policy administration in large enterprise information systems.
- Reuse and parameterisation of policy specifications.
- Analysis of policies for conflicts and inconsistencies in the specification. This also includes the ability to determine which policies apply to an object or what objects a particular policy applies to. Declarative languages make such analysis easier.
- Self-management by treating policies as objects in order to enable the specification of policies whose targets are other policies. This provides policy-based control over access to policies as well as automated activation and deactivation of policy objects based on events, to support security of the managed system and self-adaptation.

In addition to the above requirements, the design of the policy language must also address:

- Extensibility to cater for new types of policy that may arise in the future.
- Ease of use; the language must be comprehensible and easy to use by policy users. If this is not the case, then errors are inevitable.

1.2.2 Policy Deployment

Architectures for enforcing policies are moving towards strongly distributed paradigms, using technologies such as mobile code, distributed objects, intelligent agents or programmable networks. Borrowing the terminology from [Martin-Flatin et al. 1999], those paradigms in which the management task is delegated to distributed entities which actively participate in the management decision making, are called *strongly distributed*. These entities interpret and enforce policies, and their behaviour is dynamically changed by those policies. In contrast, *weakly distributed* are those paradigms in which the management decision-making is concentrated in a few managers, with distributed agents or entities acting merely as data collectors. An example is the OSI management framework [ISO/IEC 1989; Langsford 1994]. In this thesis we aim to design a strongly distributed management architecture based on a generic policy language which satisfies the requirements discussed in the previous section. We identify the following requirements for policy deployment and enforcement:

- Support for compilation of policies into different runtime representations based on the requirements of the underlying services or applications, and storage in distributed directory services. Centralised access to policies stored in distributed repositories must be possible in order to allow for analysis of deployed policies.
- Flexible and distributed enforcement of access control policies on a variety of security platforms and mechanisms. The need for application developers to code customised security into each application must be avoided, and integration of existing and new applications with the access control enforcement mechanisms must be possible with minimum overhead.
- Automated distribution of policies to their enforcement components and dynamic adaptation of the management system to changes in the stored policies and the managed objects to which the policies apply. This must include the ability to easily enable and disable deployed policies, or retract them from their enforcement components.
- Support for role-based management using the existing access control mechanisms. This includes the assignment of subjects to roles, the automated enforcement of role policies for each subject assigned to a role, and the selective activation and deactivation of assigned roles to subjects.
- Implementation of tools to enable administrators to manage policy for large-scale systems. These tools hide low-level implementation details, support the specification, deployment and coordination of policies within the system, and allow easy per user/per device review of policy.

1.3 Contribution

In this thesis we define a framework for the specification and deployment of policies to support security and general management of enterprise-wide distributed systems. The main contribution of this thesis is a generic policy specification language and the design of a strongly-distributed architecture for deploying policies. We propose a language for specifying both security and management policies, which is both declarative and simple to use for policy administrators. The language builds on experience gained in policy-based management at Imperial College over the past 10 years [Sloman et al. 1994a; Sloman 1994b; Marriott 1997; Lupu et al. 1997b; Lupu 1998]. We have refined and elaborated the concepts resulting from this experience into an integrated policy-based management framework, which presents a significant engineering challenge and constitutes one of the main contributions of this thesis.

The policy language is designed by closely following the requirements identified in Section 1.2, and its novel aspects lie in the fact that it includes support for a wide range of policy requirements,

with particular emphasis on flexibility, scalability, and extensibility. Existing work either concentrates on specific application areas or is based on specification methodologies which do not scale or provide the needed flexibility. The proposed language borrows the well-documented typing and inheritance features from object-oriented languages to provide for reusability. Policy types can be instantiated multiple times allowing parameterisation of policies with application or system specific parameters. In addition, types can be extended by specialisation to an infinite depth, thus permitting scalability, while preserving a structured specification. The language also introduces flexibility since different policy instances can be created to cater for special conditions. For example, similar service control policies may be applied on servers at different sites of an organisation at different times. These policies can be instantiated from a common policy type as both the servers to which the policies apply and the time interval during which the policies apply can be specified as instantiation parameters.

The language includes policies which range from single basic rules for access control and event-triggered management adaptation, to organisational policies used to structure responsibility based on the organisational models. Roles, relationships, and their configurations into management structures enable the specification of policies for large enterprise-wide systems. This can be used to reflect the management responsibilities and access rights of users based on their roles in the organisation, and the configurations of these roles in departments, divisions and other organisational structures. Issues such as delegation policies, information filtering access controls, and specification of application specific constraints (e.g. separation of duty) are also addressed in the language.

The deployment architecture is also novel in that it allows the automated distribution of policies to their enforcement components avoiding the need to manually manage the associations between the policies and the entities that implement them. This is facilitated by the fact that policies explicitly identify their subjects and targets in terms of domains of actual objects in the system. Domains are a unit of management similar to file directories in operating systems, and provide hierarchical structuring of objects. The use of domains enables a consistent state of deployment to be maintained, although the membership of objects in domains can change. The deployment architecture addresses the issue of compiling policies into a variety of underlying specifications (i.e. for access control enforcement) to support uniform implementation of policies across heterogeneous systems, storage of policies in distributed repositories, and enforcement of policies specified in roles.

The proposed policy framework is directly suitable for security management, which is the focus of this thesis, but can also be applied to other areas of management e.g. quality of service, storage, and configuration management. Our enforcement architecture is meant to be used as a reference model

and as a guide to management using the proposed language. It is not restricted to specific communication protocols, mechanisms or information models, and it is based on simple ideas that can be implemented using existing technologies and protocols.

As proof of concept, we describe a prototype implementation of the enforcement architecture, which covers subject-based management policies and roles. The implementation includes a policy administration toolkit that supports the specification and management of policies. It comprises an Integrated Development Environment (IDE) with a policy compiler, as well as tools for managing policies and roles at runtime.

1.4 Thesis Structure

In Chapter 2 we provide a survey of related work concentrating on policy specification. We give an account of security management approaches, including models and languages. We then concentrate on the area of policy-based management and discuss the advances which are related to the ideas presented in this thesis. We conclude with an account of the background work on which this thesis is based.

Chapters 3 and 4 present the proposed policy specification language, its syntax and usage. In Chapter 3 we describe the basic policy features of the language which include: authorisation and delegation policies for access control as well as obligation and refrain policies for subject-based management specifications. The features of the language that enable composition of the basic policies for reusability or to allow for role-based management are described in Chapter 4. Chapter 4 includes the description of meta-policies, which are used to specify application specific constraints that cannot be specified in individual policies.

In Chapter 5 we present an operational semantics for the policy language using a term-rewrite system. The semantics provide an unambiguous description for the execution of the various elements of the policy specification and can enable further work in policy analysis.

Chapter 6 is devoted to the description of the policy compiler, its design and implementation. We present our approach to generating Java objects from policy specifications and supporting multiple back-ends for mapping policies to a variety of underlying representations. We also describe tool support for specifying policies.

In Chapter 7 we describe the enforcement architecture for the deployment of policies. The architecture includes the storage of policies, their distribution and their enforcement by automated distributed entities. We also provide an overview of the prototype implementation. This comprises

the enforcement of obligation and refrain policies, the enforcement of composite policies such as roles, and a policy management toolkit.

In Chapter 8 we give a critical evaluation of the work and in Chapter 9 we conclude and suggest directions for future work.

Chapter 2

Background and Related Work

In this chapter we give an account of related work in the area of policy specification and deployment, followed by a description of those concepts on which the work presented in this thesis is based. We choose to describe the related work on policy-based management at Imperial College separately from other related work, because this forms the basis for the policy framework presented in this thesis. We start by providing an overview of security policy models, and follow this with a description of related work in the areas of policy specification, which is the focus of our survey, policy management architectures, and tool support. Figure 2.1 shows how these areas are related in a layered approach to building policy-based management systems. Note that a lot of the related work discussed in this chapter was carried out concurrently with the work presented in this thesis.

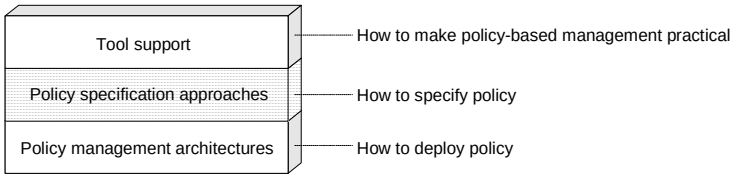


Figure 2.1 Policy survey areas

2.1 Security Policy Overview

The definitions most frequently proposed for computer security identify three primary objectives for security: *confidentiality* (sometimes called secrecy) related to the disclosure of information, *integrity* related to the modification of information, and *availability* related to the denial of access to information. To achieve these objectives three mutually supportive technologies are used: *Authentication*, *Access Control* and *Audit*. Access control is concerned with limiting the activity of legitimate users who have been successfully authenticated, and is the process of ensuring that every access to a system and its resources is controlled and that only those accesses that are authorised can take place. There are three basic components to an access control system: the *subjects*, the *targets* and the *rules* which specify the ways in which the subjects can access the targets. The set of high-level rules according to which access control must be regulated are traditionally called *access control policy* [Samarati et al. 2000]. The study of access control has identified a number of useful *access control models*, which provide a formal representation of security policies and allow the

proof of properties about an access control system. Note that the use of the term *policy* is often used in the literature to refer to both high-level security policies as defined above, and actual authorisation rules to be enforced.

2.1.1 Access Control Models

Access control policies have been traditionally divided into *discretionary* and *mandatory* policies. Discretionary policies are concerned with the specification of authorisation rules to govern the access of users to the information, whereas mandatory policies are mostly concerned with controlling information flow between the objects of a system. *Information flow* policies are often described as a separate type of policy, and are directly related to the issue of data confidentiality. Recently *role-based access control* policies are attracting increasing attention, particularly in commercial applications, and are often seen as an alternative to traditional discretionary and mandatory access control. Figure 2.2 shows a relationship between the four generic models mentioned. We view role-based policies as more closely applying the principles of discretionary access control. On the other hand information flow policies are more closely related to mandatory access control.

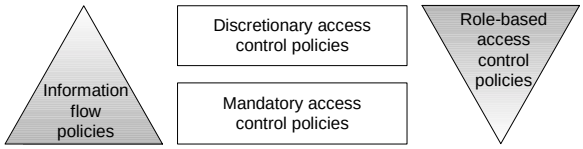


Figure 2.2 Classification of access control models

Discretionary Policies

Discretionary Access Control (DAC) policies restrict access to objects based on the identity of the subjects and/or groups to which they belong. With DAC, access control is at the discretion of the user, and the controls are discretionary in the sense that a subject with certain access permissions can pass those permissions on to any other subject. The notion of delegation of access rights is thus an important part of any system supporting DAC. Basic definitions of DAC policies use the access matrix model as a framework for reasoning about the permitted accesses. In the access matrix model the state of the system is defined by a triple (S,O,A), where S is the set of subjects, O is the set of objects and A is the access matrix where rows correspond to subjects, columns correspond to objects and entry A[s,o] reports the privileges of s on o.

Although the access matrix model remains the main mechanism for reasoning about DAC, several extensions have been proposed. Abadi at al. [Abadi et al. 1993] present a calculus for access control, which formalises access control lists and theories for deciding whether requests should be granted or not. The calculus is based on the notion of principals as the sources of requests, and

includes both simple and composite principals whereby principals can be defined in potentially nested groups. They view delegation as a basic primitive and define it as the ability of a principal A to give to another principal B the authority to act on A's behalf. The specification of rules to define under which the conditions delegation of access rights can take place is often neglected in security policy specification languages; the proposed calculus is not able to support temporal constraints on authorisations or delegations. Bertino et al. [Bertino et al. 1998] propose a formal model for extending authorisations with temporal constraints. A temporal expression is associated with an authorisation to specify periodic authorisations and restrict their validity to specific time periods. In addition, the model includes derivation rules to enable the runtime derivation of new authorisations based on the presence or absence of other authorisations in specific periods of time. Other work on discretionary policies [Samarati et al. 2000] acknowledges the need to attach more general conditions to authorisation rules to specify their validity based on system state, the state of objects on which the authorisation is defined or on accesses previously executed (history-based authorisations). Finally, many researchers identify the need for both positive and negative authorisations as a way to conveniently specify exceptions in authorisation rules [Samarati et al. 2000]. Negative authorisations specify accesses that should not be granted, and can be used to temporarily remove access rights from subjects to which positive authorisations are applied. The combined use of both positive and negative policies brings to the problem of potential inconsistencies when both a positive and a negative policy apply to the same access [Lupu et al. 1999].

Mandatory Policies

Mandatory access control (MAC) policies enforce access control on the basis of fixed regulations mandated by a central authority. Mandatory security policies are typified by the Bell-LaPadula lattice-based model. Lattice-based models were defined to deal with the issue of data confidentiality, and concentrate on restricting information flow in computer systems. This is achieved by assigning a security classification to each subject (an active entity that can execute actions) and each object (a passive entity storing information) in the system. Subjects and objects form a lattice based on their classification, which is used to enforce some fixed mandatory policies regarding the actions that subjects can execute on objects. The Bell-LaPadula model, which inspired most of the lattice-based access control models, is summarised below.

The Bell-LaPadula identifies objects (O) and subjects (S) with: $S \subseteq O$, and defines the following operations that subjects can perform on objects:

- Execute (no observation, no alteration)
- Read (observation, no alteration)
- Append (no observation, alteration)

- Write (observation, alteration)

The model defines a set of totally ordered classifications C (Top-secret, Secret, Classified, Un-classified), and a set of categories K, partially ordered by set inclusion (e.g. NATO, NUCLEAR etc). A level is defined as: $L = C \times K$, i.e. each level has two components, a classification from the set C and a subset of the set of categories K. Thus $L = (c, k)$ where $c \in C$ and k is a subset of K, and levels are related in an ordered relation as specified below; we use the $\propto$ symbol to denote the relation between levels:

$$L = (c, k) \propto L' = (c', k') \Rightarrow c \leq c' \text{ and } k \subset k'.$$

Each object o is assigned exactly one level, L(o) called the *classification* of the object, and each subject s is assigned two levels: L(s) and maxL(s). maxL(s) is called the *clearance* of the subject and it is a static level; it is the maximum level for the subject s. L(s) is the current *security level* of the subject. Note that it must be true that: $L(s) \propto maxL(s)$. The Bell-LaPadula defines the following two properties, which constitute the mandatory access rules of the model and must be observed:

Simple Security Property:

- A subject can have read access only to objects at or below its clearance, i.e. For a subject s to be able to observe object o the following must be true: $L(o) \propto maxL(s)$.

*-Property (Star-Property):

- For subject s to be able to read object o the following must be true: $L(o) \propto L(s)$.
- For subject s to be able to write object o the following must be true: $L(o) = L(s)$.
- For subject s to be able to append to object o the following must be true: $L(s) \propto L(o)$.

The conceptual framework of the Bell-LaPadula model forms the basis of other derived models one of which is the Biba model (see description in [Anderson et al. 2001]). The Biba model uses similar controls as those used in the Bell-LaPadula model for providing *integrity* of data.

Non-Discretionary Policies

Administrative policies [Sandhu et al. 1994] determine who is authorised to modify the allowed access rights and exist only within discretionary policies. In mandatory policies the access control is determined entirely on the basis of the security classification of subjects and objects. Administrative policies can be divided into: (i) *Centralised* where a single authoriser (or group) is allowed to grant and revoke authorisations to the users. (ii) *Hierarchical* where a central authoriser is responsible for assigning administrative responsibilities to other administrators. The administrators can then grant and revoke access authorisations to the users of the system according

to the organization chart. (iii) *Cooperative* where special authorisations on given resources cannot be granted by a single authoriser but needs cooperation of several authorisers. (iv) *Ownership* where a user is considered the owner of the objects he/she creates. The owner can grant and revoke access rights for other users to that object, and (v) *Decentralised* where the owner or administrator of an object can also grant other users the privilege of administering authorisations on the object.

The above discussion gives rise to another form of access control called Non-Discretionary Access Control (NDAC) [Abrams 1993] in addition to the two traditional classifications of discretionary and mandatory. NDAC identifies the situations in which authority is vested in some users, but there are controls on delegation and propagation of authority. As Abrams graphically explains, if one envisions an authority tree routed in the security administrator then mandatory is the case in which the tree has no branches, discretionary is the case in which the branches extend to every user, and non-discretionary is the case in which there are branches that do not extend to every user. Any global and persistent access control policy relying on access control decision information not directly controlled by the security administrator is non-discretionary, and in NDAC, policy for delegating authority must be explicit.

Another type of mixing DAC and MAC is work towards enriching discretionary policies with military restrictions to achieve control of information flow in commercial applications such as database systems [Samarati et al. 1998]. Discretionary policies do not enforce any control on the flow of information once this information is acquired by a process, making it possible for processes to leak information to users not allowed to read it. Myers et al. [Myers et al. 1997] extend the idea of multi-level lattice-based security policies with mechanisms to allow owners of information to declassify that information themselves.

Role-based Policies

The idea behind role-based specification of policies has existed for some time, and approaches related to the collection of privileges for assigning authorisation in database systems date back to the early 1990s with the introduction of *named protection domains* as a way of grouping the policies needed to accomplish a specific task [Baldwin 1990]. However, it was only until recently that Role-Based Access Control (RBAC) has been acknowledged as a separate model [Gligor 1995; Sandhu et al. 1996]. Role-based models regulate the access of users to the information on the basis of the activities the users execute in the system. In RBAC a role is defined as the set of access rights associated with a particular position within an organisation, or a particular working activity. RBAC models simplify authorisation management by including mechanisms for assigning access rights and users to roles; instead of specifying access rights in terms of users, permissions and users are assigned to roles, and a user assumes the permissions associated with the roles that user is assigned to.

Sandhu et al. [Sandhu et al. 1996] have specified four conceptual RBAC models in an effort to standardise RBAC. We discuss these models in order to provide an overview of the features supported by RBAC implementations. $RBAC_0$ contains *users*, *roles*, *permissions* and *sessions*. Permissions are attached to roles and users can be assigned to roles to assume those permissions. A user can establish a session to activate a subset of the roles to which the user is assigned. $RBAC_1$ includes $RBAC_0$ and introduces *role hierarchies* [Sandhu 1998]. Hierarchies are a natural means for structuring roles to reflect an organisation's lines of authority and responsibility, and are specified using inheritance between roles. Role *inheritance* enables reuse of permissions by allowing roles to be specified as junior from which senior roles can inherit permissions. For example a member of the research-staff role in an academic institution inherits the permissions of the employee role which is considered to be junior to that of a research staff member. Although the propagation of permissions along role hierarchies further simplifies administration by reducing considerably the number of permissions in the system, it is not always desirable. Organisational hierarchies do not always correspond to permission-inheritance hierarchies; the employee role may have permissions which the research-staff role should not inherit because they are specific to the employee role. Such situations lead to exceptions and complicate the specification of role hierarchies [Moffett 1998].

$RBAC_2$ includes $RBAC_0$ and introduces *constraints* to restrict the assignment of users or permissions to roles, or the activation of roles in sessions. Constraints are used to specify application-depended conditions, and satisfy well-defined control principles such as the principles of least-privilege and separation of duties. Finally, $RBAC_3$ combines both $RBAC_1$ and $RBAC_2$, and provides both role hierarchies and constraints. In recent work Sandhu et al. [Sandhu et al. 2000] propose an updated set of RBAC models in an effort to formalise RBAC. The models are called: flat RBAC, hierarchical RBAC, constrained RBAC and symmetrical RBAC, and correspond to the $RBAC_0$ – $RBAC_3$ models. Although the updated models define more precisely the basic features that must be implemented by an RBAC system, their description remains informal. A number of variations of RBAC models have been developed, and several proposals have been presented to extend the model with the notion of relationships between the roles [Barkley et al. 1999], as well as with the idea of a team, to allow for team-based access control where a set of related roles belonging to a team are activated simultaneously [Thomas 1997].

Other Security Models

Over the years other, often more sophisticated security models have been proposed to formalise security policies required for commercial applications. The most well known is the Clark-Wilson model [Clark et al. 1987], which attempts to present in a formal, abstract way commercial data processing practices. Its main goal is to ensure the integrity of an organisation’s accounting system

and to improve its robustness against insider fraud. The Clark-Wilson model recommends the enforcement of two main principles, namely the principle of well-formed transactions where data manipulation can occur only in constrained ways that preserve and ensure the integrity of data, and the principle of separation of duty. The latter reduces the possibility of fraud or damaging errors by partitioning the tasks and associated privileges so cooperation of multiple users is required to complete sensitive tasks. Authorised users are assigned privileges which do not lead to execution of conflicting tasks. This principle has since been adopted as an important constraint in security systems.

Other models include a security policy model that specifies clear and concise access rules for clinical information systems [Anderson 1996]. This model is based on access control lists and his authors claim it can express Bell-LaPadula and other lattice-based models. Finally the Chinese-wall policy (see description in [Anderson et al. 2001]) was developed as a formal model of a security policy applicable to financial information systems, to prevent information flows which cause conflict of interest for individual consultants. The basis of the model is that people are only allowed to access information which is not held to conflict with any other information that they already possess. The model attempts to balance commercial discretion with mandatory controls, and is based on a hierarchical organisation of data. It thus falls in the category of lattice-based access control models.

Access Control Policy Specification Trends

Recent proposals include a trend towards languages able to express different access control policies in a single framework in order to provide a common mechanism able to enforce multiple policies. This enables uniform specification and composition of access control policies across administrative domains and for a number of different platforms. We give an account of work in this area in Sections 2.2.1 and 2.2.2.

Another direction is certificate-based access control aimed at specifying trust policies for access to resources from un-trusted sources e.g. over the Internet. Trust has long been tied to authorisation: "*Access control consists in deciding whether the agent that makes a statement is trusted on this statement; for example, a user may be trusted (hence obeyed) when he says that his files should be deleted.*" [Abadi et al. 1993]. However, its only very recently that work on certificate-based authorisation has been intensified, as part of trust management systems (see Section 2.2.3).

2.1.2 Security Management

Security models and access control specification have been major research topics for a long time, but relatively little work exists to support the specification for operational network security

management. Traditionally, security management has been considered a sub-function of network management, and has been identified as one of the five functional areas of the OSI management framework . As defined in the OSI management framework, security management is concerned not with the actual provision and use of encryption or authentication techniques themselves but rather with their management, including reports concerning attempts to breach system security. Two important aspects are identified: (i) managing the security environment of a network including detection of security violations and maintaining security audits, and (ii) performing the network management task in a secure way [Langsford 1994].

Sloman [Sloman et al. 1993], defines security management as the support for specification of authorisation policy, translation of this policy into information which can be used by security mechanisms to control access, management of key distribution, monitoring and logging of security activities. In [Hyland et al. 1998] they define security management as the real-time monitoring and control of active security applications that implement one or more security services. This manages risk by ensuring that the security measures are operational, in balance with current conditions, and compliant with the security policy. Janson [Janson 1994] notes that the main objective of security management is to create, delete, activate, suspend, query, update or otherwise maintain the status of managed security objects through continuous gathering or distribution of information about ongoing security-relevant activities. The define access-control management as a sub-area of security management, which is concerned with the maintenance of privilege and control attribute lists, as well as their distribution or centralisation across the network. Finally, the IETF defines security policies as those policies dealing with the verification of client identities, permitting or denying access to resources, selecting and applying appropriate authentication mechanisms, and performing accounting and auditing of resources [Moore et al. 2001].

The active aspects of security policy specification, identified by all of the definitions of security management reported above, include the stipulation of actions to be performed when events such as security violations are detected. These aspects can be specified using the notion of *event-condition-action* rules proposed by some of the approaches to management policy specification (see Section 2.2.4). Such rules are often called *obligations* and are sometimes bound to authorisation policies. Minsky et al. [Minsky et al. 1985] identify the use of obligation in conjunction with authorisation policies, where the execution of actions permitted by authorisations requires (i.e. triggers) the execution of an obligation to perform certain actions. This type of policy is considered and can be expressed by some of the approaches that we examine in the next section.

2.2 Policy Specification Approaches

In our definition of policy (see Section 1.1), we have identified management goals from which policies are derived and termed those as high-level policies. Throughout the literature one can find different opinions as to the number of levels in a policy specification. This is sometimes called a policy hierarchy [Moffett et al. 1993; Weis 1994a], and represents different views on policies, relationships between policies at different levels of this hierarchy, or abstractions of policies for the purpose of refining high-level management goals into low-level policy rules whose enforcement can be fully automated. The number of levels can be arbitrary but we accept three levels of policy specification:

- *High-level abstract* policies (also referred to as management goals), which can be business goals, service level agreements, trust relationships or even natural language statements. High-level abstract policies are not enforceable and their realisation involves refining them into one of the other two policy levels, which is outside the scope of this thesis.
- *Specification-level* policies, sometimes referred to as network-level or business-level policies (or even high-level policies by some researchers). These are the policies specified by a human administrator to provide abstractions for low-level policies but in a precise format. These policies relate to specific services, or objects and their interpretation can be automated.
- *Low-level* policies or configurations such as device configurations, security mechanism configurations (e.g. access control entries, firewall rules), directory schema entries and so on. The line separating low-level policies and device configuration is sometimes not clear, and directly specifying policies at this level is often a bottleneck to both scalability and interoperability.

Figure 2.3 summarises the approaches used for *specification-level* policies, indicated in order of ease of specification and flexibility from top to bottom. We divide these approaches into three main categories: policy specification languages, rule-based specifications, and formal logic languages. From a human input standpoint, the best way to specify policies is using a policy language because it provides considerable flexibility compared to the other approaches. However, the use of a generic high-level language compromises the ability to analyse policy specifications, a process that can be made considerably simpler with the design of declarative languages. In the rule-based approach policies are specified as sequences of rules of the form: *if condition then action*, and are mostly applied to quality of service management in IP networks. Finally, logic-based approaches are driven by the need to analyse the policy specification, but generally fail to directly map to an implementation and are not easily interpreted by humans. Formal logic is mostly used in the specification of security policies.

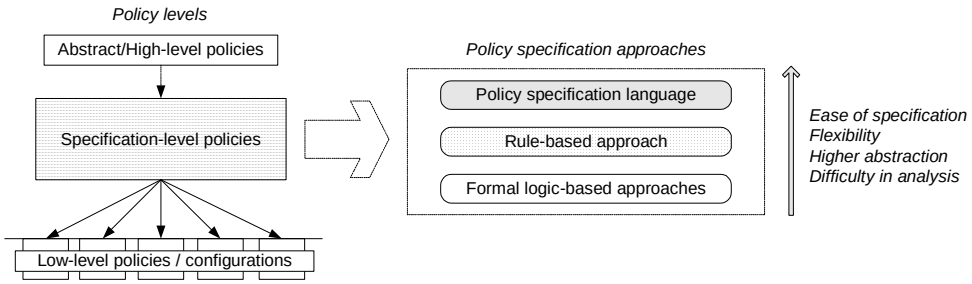


Figure 2.3 Policy levels and specification approaches

There are many ways to divide the discussion on the various policy specification approaches, e.g. based on the granularity of specification, based on the functionality, or based on the application domain. We start by presenting approaches used to specify security polices, including logic-based languages, high-level languages and work on specification of trust. We then present work in the area of management policy specification, followed by approaches specific to network management policies for quality of service and traffic routing.

2.2.1 Logic-Based Authorisation Languages

Formal logic-based approaches have been used to specify security policies, but are generally not intuitive and do not easily map onto implementation mechanisms. They assume a strong mathematical background, which can make them difficult to use and understand. The **authorisation specification language** (ASL) [Jajodia et al. 1997] is an example of a formal logic language for specifying access control policies. Although it provides support for role-based access control, the language does not scale well to large systems because there is no way of grouping rules into structures for reusability. Authorisation rules identify the actions authorised for specific users, groups or roles, but cannot be composed into roles to provide for reusability i.e. there is no explicit mechanism for assigning authorisations to roles; instead this is specified as part of the condition of authorisation rules. The following is an example of an authorisation rule in ASL, which states that all subjects belonging to group *Employees* but not to *Soft-Developers* are authorised to read *file1*.

```
cando(file1, s, +read) ← in(s, Employees) & ¬in(s, Soft-Developers)
```

The *cando* predicate can also be used to specify negative authorisations; the sign in front of the action in the *cando* predicate indicates the modality of the authorisation. However, there is no explicit specification of delegation and no way of specifying authorisation rules for groups of target objects that are not related by type. A *dercando* predicate is defined in the language to specify derived authorisations based on the existence or absence of *cando* rules (i.e. other authorisations in the system). In addition, two predicates *do* and *done*, can be used to specify history-dependent authorisations based on actions previously executed by a subject. The language includes a form of meta-policies called *integrity rules* to specify application-dependent conditions that limit the range

of acceptable access control policies. In a recent paper [Jajodia et al. 2000] the language has been extended with predicates used to evaluate hierarchical or other relationships between the elements of a system such as the membership of users in groups, inclusion relationships between objects or supervision relationship between users.

The type of logic used in ASL is called **stratified clause form logic**. Barker [Barker 2000] adopts a similar approach to express a range of access control policies using stratified clause form logic, with emphasis on RBAC policies. According to the author, this form of logic is appropriate for the specification of access control policies mostly due to its simple high-level declarative nature. The function-free normal clause logic adopted by Barker, defines a normal clause as an expression of the following form: `H ← L1,L2,...,Lm (m ≥ 0)`. The head of the clause, `H`, is an atom and `L1,L2,...,Lm` is a conjunction of literals that constitutes the body of the clause. If the conjunction of literals `L1,L2,...,Lm` is true (proved) then `H` is true (proved). A literal is an atomic formula or its negation and a normal theory is defined as a finite set of normal clauses. A stratified theory extends a normal theory by eliminating some forms of "recursion-via-negation", which makes the computation of the theory more efficient. The negation of literals is used to specify negative permissions

In [Barker et al. 2001] they show how policies specified in stratified logic can be automatically translated into a subset of SQL to protect a relational database from unauthorised read and update requests. The following example from [Barker et al. 2001] demonstrates their approach:

```
permitted(U,P,O) ← ura(U,R1),activate(U,R1),senior-to(R1,R2),rpa(R2,P,O)
```

The above clause specifies that user *U* has the permission *P* on object *O* if *U* is assigned to a role *R1*, *U* is active in *R1*, and *R1* inherits the *P* permission on *O* from *R2*. This expression assumes that the following predicates have been defined: *activate(U, R)* to denote that *U* is active in *R*, *ura(U,R)* to assign user *U* to role *R*, *rpa(R,P,O)* to assign permission *P* on object *O* to role *R*, and *senior-to(R1,R2)* to denote that role *R1* is senior to *R2*.

Ortalo [Ortalo 1998] describes a language to express security policies in information systems based on the logic of permissions and obligations, a type of modal logic called **deontic logic**. Standard deontic logic is static instead of dynamic, and centres on impersonal statements instead of personal; we see the specification of policies as a relationship between explicitly stated subjects and targets instead. In his approach he accepts the axiom `Pp = ¬O¬p` ("permitted p is equivalent to not p being not obliged") as a suitable definition of permission. This axiom is not appropriate for the modelling of obligation and authorisation policies because the two need to be separated. An obligation policy requires a relevant authorisation policy to permit the actions defined in the obligation, but having an obligation for an action does not imply the permission to execute the action.

The **role definition language** (RDL) [Hayton et al. 1998] is a formal language based on Horn clauses defined within the Oasis architecture for secure, interworking services defined at Cambridge University. RDL is based on sets of rules that indicate the conditions under which a client may obtain a name or role, where a role is synonymous to a named group. The conditions for entry to a role are described in terms of credentials that establish a client's suitability to enter the role, together with constraints on the parameters of those credentials. The work on RDL also falls into the category of *certificate-based access control,* which is adopted by trust-management systems described separately in Section 2.2.3. The following is an example of an authorisation rule in RDL, which establishes the right for clients assigned to the *SeniorHaematologist* role to invoke the *append* method if they also possess a certificate called *LoggedOn(km, s)* issued by the *Login service*, where *s* is a trusted server, i.e. if they have been logged on as *km* on a trusted machine.

```
append(haematology-field, y, x) ← SeniorHaematologist(x) ∧ Login.LoggedOn(km, s):s in
TrustedServers
```

Roles in RDL are also considered as credentials, and can be used to assign clients to other roles as in the following example where a user *x*, who belongs to both the *Haematologist* and the *SeniorDoctor* roles, is also a *SeniorHaematologist*:

```
SeniorHaematologist(x) ← Haematologist(x) ∧ SeniorDoctor(x)
```

RDL has an ill-defined notion of delegation, whereby roles can be delegated instead of individual access rights in order to enable the assignment of users to certain roles. The notion of *election* is introduced to enable a client to delegate a role that they do not themselves possess, to other clients. We believe that assignment of users to roles should be controlled with user-assignment constraints instead. The following example from [Hayton et al. 1998] specifies that the *chief examiner* may elect any logged on user who belongs to the group *Staff* to be an *examiner*, for the examination subject *e*.

```
Examiner(p,e) ← Login.LoggedOn(p,s) ◁ ChiefExaminer : p in Staff
```

Role-based Access Control

Some researchers have concentrated on formal specification languages which implement specific concepts described within the RBAC models, with emphasis on the specification of constraints.

Chen et al. [Chen et al. 1995] introduce a language based on set theory for specifying **RBAC state-related constraints**, which can be translated to a first-order predicate-logic language. They define an RBAC system state as the collection of all the attribute sets describing roles, users, privileges, sessions as well as assignments of users to roles, permissions to roles and roles to sessions. They also define constraints in RBAC as the specification of restrictions to RBAC states, called

invariants, as well as to state changes, called preconditions. They use this model to specify constraints for RBAC in two ways: (i) by treating them as invariants that should hold at all times, and (ii) by treating them as preconditions for functions such as assigning a role to a user. They define a set of global functions to model all operations performed in an RBAC system, and specify constraints which include: conflicting roles for some users, conflicting roles for sessions of some users, and prerequisite roles for some roles with respect to other users. The following example from [Chen et al. 1995] can be used as an invariant or as a precondition to a user-role assignment, to indicate that assigned roles must not be conflicting with each other:

```
Role set: R = {r1 , r2 , ... , rn }
User set: U = {u1 , u2 , ... , um }
Check-condition: oneelement(R) ∈ role-set(oneelement(U)) →
                 allother(R) ∩ role-set(oneelement(U)) = Ø
```

The two non-deterministic functions, `oneelement` and `allother`, are introduced in the language to replace explicit quantifies. `Oneelement` selects one element from the given set, and `allother` returns a set by taking out one element from its input.

RSL99 [Ahn et al. 1999] is a formal language, which extends the ideas introduced in [Chen et al. 1995] and can be used for specifying separation of duty properties in role-based systems. The language covers both static and dynamic separation of duty constraints, and its grammar is simple, although the expressions are rather complicated and inelegant.

Since time is not defined as part of the state of an RBAC system as defined in [Chen et al. 1995], the two proposed languages described above cannot specify temporal constraints. The specification of temporal constraints on role activations is addressed in the work by Bertino et al. [Bertino et al. 2000], which extends the RBAC models with a temporal model called **TRBAC**. They propose an expression language that can be used to specify two types of temporal constraints: (i) periodic activation and deactivation of roles using periodic expressions, and (ii) specification of temporal dependencies among role activations and deactivations using role triggers.

A role can be activated/deactivated by means of role triggers specified as rules to automatically detect activations/deactivations of roles. Role triggers can also be time-based or external requests to allow administrators to explicitly activate/deactivate a role, and are specified in the form of prioritised event expressions. The following example from [Bertino et al. 2000] shows two periodic expressions (*PE1* and *PE2*) and two role triggers (*RT1* and *RT2*). The periodic expressions state that the role *doctor-on-night-duty* must be active during the night. The role triggers state that the role *nurse-on-night-duty* must be active whenever the role *doctor-on-night-duty* is active. In the example, the symbols *VH* and *H* stand for very high and high respectively and denote priorities for the execution of the rules.

```
(PE1) ([1/1/2000, ∞], Night-time, VH:activate doctor-on-night-duty)
(PE2) ([1/1/2000, ∞], Day-time, VH:deactivate doctor-on-night-duty)
(RT1) (activate doctor-on-night-duty → H:activate nurse-on-night-duty)
(RT2) (deactivate doctor-on-night-duty → H:deactivate nurse-on-night-duty)
```

In general, logic-based approaches to policy specification allow formal reasoning about the specified policies, and enable properties of the specification to be proved, but they are not aimed at human interpretation and do not directly map to an implementation. The use of logic is important in the area of policy refinement whereby high-level abstract policy specifications are refined to formats amenable to direct implementation or analysis. As an example, Michael et al. [Michael et al. 2001] describe a suite of tools, which serve as an expert database management system to automate the process of mapping natural language policy statements into equivalent first-order predicate calculus. Their proposal maps statements like the following: "*A valid password is issued to an authorised user to allow the user to logon. The user must logon to obtain access to the system. Access is granted when a valid password is entered to complete a logon*" into modal logic as shown below, and answers queries about the policies:

```
∀x (password(x) → (∃ s (∀c (character(c) ∧ part_of(c,x)) → member(c,s))) → (∃n (size(s,n)
∧ n ≥ 8) ))))
```

2.2.2 High-level Policy Languages for Security

Although most of the efforts on security policy specification focus on the use of formal logic, some approaches have been proposed for high-level security languages. The **security policy language** (SPL) [Ribeiro et al. 2001a] is an event-driven policy language that supports access-control, history-based and obligation-based policies. SPL is implemented by an event monitor that for each event decides whether to allow, disallow or ignore the event. Events in SPL are synonymous with action calls on target objects, and can be queried to determine the subject who initiated the event, the target on which the event is called, and attribute values of the subject, target and the event itself. SPL supports two types of sets to group the objects on which policies apply: groups and categories. Groups are sets defined by explicit insertion and removal of their elements, and categories are sets defined by classification of entities according to their properties. The building blocks of policies in SPL are constraint rules which can be composed using a specific tri-value algebra with three logic operators: *and*, *or* and *not*. A simple constraint rule is comprised of two logical binary expressions, one to establish the domain of applicability and another to decide on the acceptability of the event. The following extract from [Ribeiro et al. 2001a] shows examples of simple rules and their composition. Note that conflicts between positive and negative authorisation policies are avoided by using the tri-value algebra to prioritise policies when they are combined as demonstrated by the last composite rule of the example. The keyword *ce* in the examples is used to refer to the current event.

```
// Every event on an object owned by the author of the event is allowed
OwnerRule: ce.target.owner = ce.author :: true;

// Payment order approvals cannot be done by the owner of payment order
DutySep: ce.target.type = "paymentOrder" & ce.action.name = "approve"
        :: ce.author != ce.target.owner;

// Implicit deny rule.
deny: true :: false;

// Simple rule conjunction, with default deny value
OwnerRule AND DutySep OR deny;

// DutySep has a higher priority then OwnerRule
DutySep OR (DutySep AND OwnerRule);
```

SPL defines two abstract sets called *PastEvents* and *FutureEvents* to specify history-based policies and a restricted form of obligation policy. The type of obligation supported by SPL is a conditional form of obligation, which is triggered by a pre-condition event:

```
Principal_O must do Action_O if Principal_T has done Action_T
```

Since the above is not enforceable, they transform it into a policy with a dependency on a future event as shown below, which can be supported in a way similar to that of history-based policies:

```
Principal_T cannot do Action_T if Principal_O will not do Action_O
```

SPL obligations are thus additional constraints on the access control system, which can be enforced by security monitors [Ribeiro et al. 2001b], and not obligations for managers or agents to execute specific actions on the occurrence of system events, independent of the access control system.

The notion of a policy is used in SPL to group set definitions and rules together to specify security policies that can be parameterised; policies are defined as classes which allow parameterised instantiation. Instantiation of a policy in SPL also means activation of the policy instance, so no control over the policy life-cycle is provided. Further re-use of specifications is supported through inheritance between policies. A policy can inherit the specifications of another policy and override certain rules or sets. Policy constructs can also be used to model roles, in which case sets in the policy specify the users allowed to play the role and those playing the role. Rules or other nested policies inside a role policy specify the access rights associated with the role. SPL provides the ability to hierarchically compose policies by instantiating them inside other policies, thus enabling the specification of libraries of common security policies that can be used as building blocks for more complex policies. The authors claim that this hierarchical composition also helps restrict the scope of conflicts between policies, however this is not clear as there may be conflicts across policy hierarchies. Note that SPL does not cater for specification of delegation of access rights between subjects, and there is no explicit support for specifying roles. The following example from [Ribeiro et al. 2001a] defines an *InvoiceManagement* policy, which allows members of the *clerks*

team to access objects of type *invoice*. The actual policy that permits the access is specified as *ACL* separately and instantiated within *InvoiceManagement* using the keyword *new*:

```
policy InvoiceManagement
{
    // Clerks would usually be a role but for simplicity here it is a group
    team clerks ;

    // Invoices are all object of type invoice
    collection invoices =
        AllObjects@{ .doctype = "invoice" };

    // In this simple policy clerks can perform every action on invoices
    DoInvoices: new ACL(clerks, invoices, AllActions);
    ?usingACL: DoInvoices;
}
```

Tower [Hitchens et al. 2001] is a language designed to specify RBAC policies. The language supports the basic RBAC model elements, extended with the *target objects* to which access is being controlled. Thus Tower provides structures to define objects, privileges, permissions, users and roles, and allows the definitions of multiple structures within blocks bounded by *begin* and *end* statements to enable reusability of common elements. Note that privileges define a specific access type on an object, permissions are composed of privileges and roles contain a set of permissions. Simple or set variables can be defined within a block to enable reference to system or structure-specific values (e.g. the id of a user or the time of access on an object) across all structures defined in the block. The use of variables enables fine-grained authorisations with the ability to base access control decisions on past events. The following example from [Hitchens et al. 2000] specifies a dynamic separation of duty policy, and demonstrates the definition of some of the structures in Tower. The example considers a class of *cheque* objects, which may be accessed by members of a role *accountant*, but with no user permitted to both issue and authorise the same cheque. The variable *issuing_user* is used to represent the id of the user which issues the check, and is used in the specification of the condition for the separation of duty. The *issuing_user* variable is assigned to be the user who executes the *issue_privilege*, and must not be the user who executes the *authorise_privilege*.

```
begin
        issuing_user* : userid
        issue_privilege := privilege
                issuing_user := user
                {issue}
        end_privilege
        authorise_privilege := privilege
                issuing_user <> user
                {authorise}
        end_privilege
        check_permission := permission
                cheque_class
                privilege {issue_privilege, authorise_privilege}
        end_permission
        accountant := role
                permissions {check_permission}
        end_role
end
```

The specification of privileges in Tower allows conditions to restrict the validity of the privilege, and actions that must be executed when the invocation of any of the methods identified in the privilege is allowed. However, privileges are always specified for a class of objects, which is restrictive because in realistic applications privileges are normally specified for sets of objects grouped together for reasons other than the class or type of the objects. In addition, the need to explicitly define object structures for each object that needs to be referred in the permissions makes the use of the language impractical in large-scale systems where the objects are already described in interface definition languages. In addition, Tower is based on the concept of ownership whereby each object and structure has an owner that has control over that object or structure, making the language unnecessarily complicated. The specification of roles is Tower is also problematic. A role defines which other roles are mutually exclusive to that role, as well as the conditions under which the role can be active in a session. This restricts the use of a role to a specific application; some roles in an organisation may be defined as mutually exclusive within one department but not mutually exclusive within another. The constraints on mutually exclusive roles, role activation and assignment of users to roles must be separated from the specification of a role, and can be specified in the form of meta-rules or meta-policies for sets of roles. Finally, roles in Tower can inherit permissions from other roles and arbitrarily specify which permissions to exclude. This results in a messy system where the inheritance relationships between roles are difficult to maintain.

Recent proposals express access control policies as XML documents as exemplified by **XACML** [OASIS 2001]. XACML is an XML specification for expressing policies for information access over the Internet and is being defined by the Organisation for the Advancement of Structured Information Standards (OASIS) technical committee. The language provides XML with a sophisticated access control mechanism that enables the initiator not only to securely browse XML documents but also to securely update each document element. Similar to existing policy languages, XACML is used to specify an *subject-target-action-condition* oriented policy in the context of a particular XML document. The notion of subject comprises identity, group, and role and the granularity of target objects is as fine as single elements within the document. The language supports roles, which are the same as groups, and are defined as collections of attributes relevant to a principal. XACML includes conditional authorisation policies, as well as policies with external post-conditions to specify actions that must be executed prior to permitting an access. e.g. "*A physician may read any record and write any medical element for which he or she is the designated primary care physician, provided an email notice is sent to the patient or the parent/guardian, in case the patient is under 16*". An example of a policy specified in XACML is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<applicablePolicy xmlns="http://www.oasis-open.org/committees/accessControl/docs/draft-
actc-schema-policy-08.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:rec="medico.com/record" xmlns:saml="hhtp://www.oasis-
open.org/committees/security/docs/draft-sstc-schema-assertion-22"
xsi:schemaLocation="http://www.oasis-open.org/committees/accessControl/docs/draft-actc-
schema-policy-08.xsd" majorVersion="0" minorVersion="8" issuer="medico.com"
policyName="researchers may read medical elements and the patient's date of birth and
gender" issueInstant="2002-01--8">
<!-- -->
     <target
        resourceClassification="medico.com/record/medical.*"
        resourceClassificationTransform="http://www.oasis-open.org/committees/
        accessControl/docs/transforms/regularExpression">
                read
     </target>
        <target
        resourceClassification="medico.com/record/patient/patientDoB.*"
        resourceClassificationTransform="http://www.oasis-open.org/committees/
        accessControl/docs/transforms/regularExpression">
                read
     </target>
        <target
        resourceClassification="medico.com/record/patient/patient/gender.*"
        resourceClassificationTransform="http://www.oasis-open.org/committees/
        accessControl/docs/transforms/regularExpression">
                read
     </target>
     <policy>
        <equal>
                <valueRef attributeName="rec:role"/>
                <value xsi:type="string">researcher</value>
        </equal>
     </policy>
</applicablePolicy>
```

The above policy assumes an XML schema to describe medical records, and specifies that a researcher may read a medical element and the patient's date of birth and gender. Although the granularity of access control with XACML is fine, the policy is rather verbose and not really aimed at human interpretation. In addition, the language model does not include a way of grouping policies. Note that XACML is intended to be used in conjunction with SAML (security assertion and markup language) assertions and messages, and can thus also be applied to certificate-based authorisations. We discuss certificate-based authorisations in the following section. The work on XACML includes an architecture for enforcing policies which extends the IETF policy architecture described in Section 2.3.

LaSCO [Hoagland et al. 1998] is a graphical approach for specifying security constraints on objects, in which a policy consists of two parts: the *domain* (assumptions about the system) and the *requirement* (what is allowed assuming the domain is satisfied). Policies defined in LaSCO have the appearance of conditional statements used to express authorisations between objects in the system and are stated as policy graphs. A policy graph is an annotated directed graph where the annotations are domain and requirement predicates. Nodes in the policy graph represent the sort of objects described by the associated domain predicate. Collectively, the nodes, edges, and domain predicates form the domain of a policy graph. The domain describes when the policy is in effect, i.e. when it applies. The other part of the policy graph is the requirement, which consists of the requirement predicates on each of the nodes and edges. A node requirement predicate is an expression that must be met on the object and constitutes an authorisation policy. LaSCO cannot

specify any form of obligation policies, and there is no way of composing policies or specifying policies for groups of objects apart from those defined for classes of objects. This makes the scope of this approach very limited to satisfy the requirements of security management. In addition graphs are often used in conjunction with a textual version to specify details not easily expressed in the graphical format. In LaSCO this is lacking making the language difficult to use and further restricting its expressiveness. Note however, that a graphical approach to specifying policies is attractive for human users, and is thus an interesting future research direction.

2.2.3 Trust Specification

Applications such as e-commerce and other Internet-enabled services require connectivity between entities that do not know each other. In such situations, the traditional assumptions for establishing and enforcing access control do not hold; subjects of requests can be remote, previously unknown users, making the separation between authentication and access control difficult. A possible solution to this problem is the use of digital certificates or credentials representing statements certified by given entities, which can be used to establish properties of their holder (e.g. identity, accreditation). Access control makes the decision of whether or not a party can execute an access based on properties that the party may have, and can prove by presenting one or more certificates. Such an approach is often called certificate-based authorisation and is adopted for the specification of trust. Trust management frameworks combine authentication with authorisation [Grandison et al. 2000] and are used for applications such as web based labelling, signed email, active networks and e-commerce.

In [Blaze et al. 1998; Blaze et al. 1999], two trust management applications are presented: the **PolicyMaker** and its successor **KeyNote.** Both of these applications are used to answer signed queries of the form "*does a set of requested actions r, supported by credential set C, comply with policy P?"*, where the credentials can be public key certificates with anonymous identity. Both policies and credentials are predicates specified as simple C-like and regular expressions. In this context a policy is a trust assertion that is made by the local system and is unconditionally trusted by the system. Although trust management systems provide an interesting framework for reasoning about trust between unknown parties, assigning authorisations to keys may result in authorisations that are difficult to manage [Samarati et al. 2000]. In addition, providing a common solution to both authentication and access control makes the system more complex.

The **trust policy language** (TPL) by IBM [Herzberg et al. 2000] provides a clearer separation between the authentication of subjects based on certificates and the assignment of authorisations to those subjects which have been successfully authenticated. With TPL, the credentials result in a client being assigned to a role which specifies what the client is permitted to do, where a role is a

group of entities that can represent specific organisational units (e.g. employees, managers, auditors). The assignment of access rights to roles is outside the scope of TPL; the philosophy of the work on TPL is to extend role-based access control mechanisms by mapping unknown users to well defined roles. Although the certificate is intended to be format-independent, the current implementation of the system uses X.509v3 certificates, and defines the language in XML, which makes the syntax rather verbose. Note that unlike KeyNote, TPL permits negative certificates interpreted as suggestions not to trust a user or not to assign a user to a given role. The following example is taken from [Grandison et al. 2000] to demonstrate the use of TPL. In summary, the policy states that a customer of a retailer company is an employee of a department of a partner company. The first group defined is the originating *retailer*. Then, it is stated that entities that have partner certificates, signed by the original retailer, are placed in the group *partners*. The group *department* is defined as any user having a partner certificate signed by the partners group. Finally, the *customer* group consists of anyone that has an employee certificate signed by a member of the departments group who has a rank greater than 3.

```
<POLICY>
     <GROUP NAME=”self”> </GROUP>
     <GROUP NAME=”partners”>
          <RULE>
               <INCLUSION ID=”partner” TYPE=”partner” FROM “self”> </INCLUSION>
          </RULE>
     </GROUP>
     <GROUP NAME=”departments”>
          <RULE>
               <INCLUSION ID=”partner” TYPE=”partner” FROM=”partners”> </INCLUSION>
          </RULE>
     </GROUP>
     <GROUP NAME=”customers”>
          <RULE>
               <INCLUSION ID=”customer” TYPE=”employee” FROM=”departments”> </INCLUSION>
               <FUNCTION>
               <GT>
                 <FIELD ID=”customer” NAME=”rank”></FIELD>
                 <CONST>3</CONST>
               </GT>
               </FUNCTION>
          </RULE>
     </GROUP>
</POLICY>
```

2.2.4 Management Policy Specification

The **policy description language** (PDL) is an event-based language originating at the network computing research department of Bell-Labs [Lobo et al. 1999]. In PDL they use the *event-condition-action* rule paradigm of active databases to define a policy as a function that maps a series of events into a set of actions. The language can be described as a real-time specialised production rule system to define policies. The syntax of PDL is simple and policies are described by a collection of two types of expressions: *policy rules* and *policy defined event propositions*. Policy rules are expressions of the form:

```
event causes action if condition
```

Which reads: If the event occurs under the condition the action is executed. Policy defined event propositions are expressions of the form:

```
event triggers policy-defined-event if condition
```

Which reads: If the event occurs under the condition, the policy-defined-event is triggered. Events can be primitive or complex, and there are two types of primitive events: the policy defined events, which are only generated by the policy defined event propositions, and the system events, which are generated by the environment. Primitive event classes can define attributes, and instances of the classes take actual values for those attributes that can be referenced by other events, actions or conditions within the same rule. Primitive events can be composed to form complex events that enable policies to be enforced under any of the following situations:

- If two events *e1* and *e2* occur simultaneously.
- If an event *e* does not occur.
- If an event *e2* immediately follows an event *e1*.
- If an event *e2* occurs after an event *e1*.

The following example from [Kohli et al. 1999] makes use of some of the different features of the language to define a policy for a service provider network which rejects call requests when there is an excessive number of network signalling timeouts over the calls made (i.e. overload state) until the time-out rate goes down to a reasonable number. The policy has three policy defined event propositions and one policy rule proposition.

```
Events: normal_mode: policy defined event, restricted_mode : policy defined event
        call_made: system event, time_out: system event, power_on: system event

Actions: restrict_calls, accept_all_calls

Policy description:
        // when the system starts the primitive event normal_mode is triggered. i.e. the
        // system starts in normal mode
        power_on triggers normal_mode

        // when in normal_mode, a sequence of call_made or time_out events will trigger
        // restrict_mode if the overload threshold is exceeded. t is the overload ratio
        // of signalling timeouts over the calls made. The ^ sign denotes a sequence of
        // zero or more events.
        normal_mode, ^(call_made | time_out)  triggers restricted_mode
                                              if Count(time_out) > t*Count(call_made)

        restricted_mode causes restrict_calls

        // when in overlaod mode, a sequence of call_made or time_out events will trigger
        // normal_mode if the normal threshold is exceeded. t' is considered to be a
        // reasonable timeout rate
        restricted_mode, ^(call_made | time_out) triggers normal_mode
                                                 if Count(time_out) < t'*Count(call_made)

        // Assumes only one callMade or timeOut event per epoch
        normal_mode causes accept_all_calls
```

Despite its expressiveness, PDL does not support access control policies, nor does it support the composition of policy rules into roles, or other grouping structures. The language has clearly defined semantics and an architecture has been specified for enforcing PDL policies. Work on conflict resolution for policies written in PDL is described in [Chomicki et al. 2000], and extensions to the language to specify workflows for network management can be found in [Kohli et al. 1999]. The language has been used to program Lucent switching products [Virmani et al. 2000] and proves to be powerful in a variety of network operations and management scenarios.

The group working on the International Standards Organisation (ISO) Open Distributed Programming Reference model (ODP-RM) are defining an **enterprise language** as part of the RM-ODP Enterprise Viewpoint [ISO/IEC 1999], which incorporates concepts such as policies and roles within a community. A *community* in RM-ODP terminology is defined as a *configuration* of objects formed to meet an *objective*. The objective is expressed as a contract, which specifies how the objective can be met, and a configuration is a collection of objects with defined relationships between them. The community is defined in terms of the following elements:

- The enterprise objects comprising the community,
- The roles fulfilled by each of those objects and the relationships between them,
- The policies governing the interactions between enterprise objects fulfilling roles,
- The policies governing the creation, usage and deletion of resources,
- The policies governing the configuration of enterprise objects and assignment of roles to enterprise objects,
- The policies relating to the environment contract governing the system.

Policies constrain the behaviour of enterprise objects that fulfil actor roles in communities and are designed to meet the objective of the community. Policy specifications specify what behaviour is allowed or not and often contain prescriptions of what to do in case some rule is violated. Policies in the ODP enterprise language thus cover the concepts of obligation, permission and prohibition.

The ODP enterprise language is an abstract language in the sense that it does not prescribe the use of any particular notation. Recently, there have been a number of attempts to define precise languages that implement the abstract concepts of the enterprise language. These approaches concentrate on using UML to graphically depict the static structure of the enterprise viewpoint language as exemplified by [Steen et al. 2000] (see Figure 2.4), as well as languages to express policies based on those UML models. Steen et al. [Steen et al. 1999; Steen et al. 2000] propose a language to support the enterprise viewpoint where policy statements are specified using the grammar shown below. Each statement applies to a role, the subject of the policy, and represents either a permission, an obligation or a prohibition for that role. The grammar of the language is concise, however it does not allow composition of policies or constraints for groups of policies.

Constraints cannot be specified to restrict the activation/deactivation of roles or the assignment of users and permissions in roles. Note that the Object Constraint Language (OCL) [OMG 1999b] is used to express the logical conditions in the before-, if- and where- clauses defined in the grammar.

```
[R?] A <role> is (permitted | obliged | forbidden) to (do <action> [before <condition>] |
satisfy <condition>)[, if <condition>][, where <condition>][, otherwise see <number>].
```

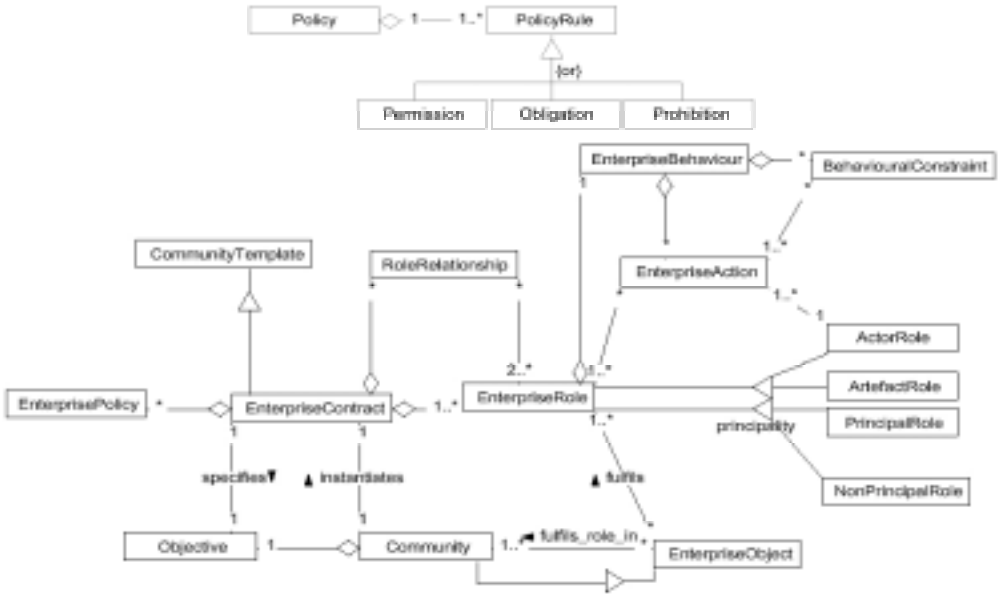


Figure 2.4 A UML meta-model of the enterprise viewpoint language

The authors specify the semantics of the policy language by translating it to Object-Z, an object-oriented extension of the specification language Z. The following examples from [Steen et al. 2000] demonstrate the use of the proposed language.

```
[R1] A Borrower is permitted to do Borrow(item:Item), if(fines < 5*pound).
[R2] A UGBorrower is forbidden to do Borrow(item:Item), where item.isKindOf(Periodical).
[R3] A Borrower is obliged to do Return(item:Item) before (today > dueDate),
               if (loans→exists(loan | loan.item = item)),
               where (dueDate = loans→select(loan | loan.item = item).dueDate),
               otherwise see R4.
```

Policy statements *R1* and *R2* specify a permission and a prohibition respectively. *R1* permits a member of the *Borrower* role to borrow an item if the fines of that borrower are less than 5 pounds. *R2*, forbids an undergraduate student belonging to the *UGBorrower* role to borrow periodicals. *R3* is an obligation specifying that a borrower must return an item by the *dueDate* of that item. *R3* is conditional upon the item to be returned actually being on loan to the borrower, as specified by the *if*-clause. The *where*-clause constrains the logic variable *dueDate* to be equal to the *dueDate* of the loan in question, and the *before*-clause contains a condition upon which the obligation should have been fulfilled. Obligations do not contain explicit specifications of the events upon which the actions must be executed which makes their implementation difficult. Note that the *otherwise*-

clause is an exception mechanism that indicates what will happen when the obligation is violated. The actions to be executed on a violation are specified in another policy (*R4*).

Policy-based management is also applied to *configuration management* and builds on monitoring software to enable automation of network and system administration through the event-condition-action paradigm; policy-based configuration languages associate the occurrence of specified events or conditions, with responses to be carried out by an agent. **Cfengine** is a language-based administration system targeted at BSD and System-5-like operating systems, which might be connected through a TCP/IP network [Burgess 1995]. Cfengine grew out of the need to replace complex shell scripts used for the automation of administration tasks on Unix systems and allows the creation of single, central configuration files which describe how every host on the network should be configured. It uses the idea of classes to group hosts and dissect a distributed environment into overlapping sets. Host-classes are essentially labels which document the attributes of different systems. The following classes are meaningful in the context of a particular host: (i) the identity of the machine, including hostname, address, network, (ii) the operating system and architecture of the host (iii) an abstract user-defined group to which the host belongs (iv) the result of any proposition about the system, including the time or date. Policies are specified for classes of hosts and define a sequence of actions regarding the configuration of a host. The following example demonstrates the use of the language for configuration management [Burgess et al. 2001]:

```
files:
     (linux|solaris).Hr12.OnTheHour.!exception_host::
          /etc/passwd mode=0644 action=fixall inform=true
```

The first line simply defines the name *files* for the action. The second line identifies the class of hosts for which the action is to be executed, followed by the actual command. The command-line specifies that the cfengine agent, which is always the subject of the policy, must search for all password files with an invalid mode, fix them, and inform the administrator. The class membership expression specifies all hosts which are of type *linux* or *solaris*, during the time interval from 12:00am to 12:59am, apart from a host labelled with the class *exception_host*. Note that the second line specifies too much information in a single clause making the interpretation of the policy too complicated. It identifies the target of the policy, i.e. all the hosts falling within the classification, the condition for execution of the policy, which is a time interval, and a trigger which specifies that the action must be executed *on the hour*. Policies are stored in a central repository, accessible to every host, and an active cfengine agent on each host executes the policies which apply only to that host.

Cfengine achieved its original goals through a scripting language suitable for system administrators, which automates common administrative tasks on Unix systems. However, it is still

lacking functionality to enable scalable policy-based configuration management, and its creators admit the need for extensions to enable enterprise-level policy specification [Burgess 2001].

Others, focus on the specification of policies using the full power of a general purpose scripting or interpreted language (eg TCL or Java) which can be loaded into network components or agents to implement policies. Such approaches are often leveraging the mechanisms in the area of active networks [Tennenhouse et al. 1997] to enable the control of resources at a very low level. For example Bos et al. [Bos 1999] use C-programs to specify application policies for resource management in netlets, which are small virtual networks within a larger virtual network. In general for all of these approaches, the security concerns are increased, and malicious or improperly tested code can potentially damage the network. In addition, it is difficult to determine whether two computer programs specifying two different policies are contradictory or conflict with each other in any way.

2.2.5 Network Policy Specification

The area of network policy specification has recently seen a lot of attention both from the research and the commercial communities. We call *network policy* the rules which define the relationship between clients using network resources and the network elements that provide those resources. The main interest in network policies is to manage and control the quality of service (QoS) experienced by networked applications and users, by configuring network elements using policy rules. The most notable work in this area is the Internet Engineering Task Force (IETF) policy model, which considers policies as rules that specify actions to be performed in response to defined conditions:

```
if <condition(s)> then <action(s)>
```

The condition-part of the rule can be a simple or compound expression specified in either conjunctive or disjunctive normal form. The action-part of the rule can be a set of actions that must be executed when the conditions are true. Although this type of policy rule prescribes similar semantics to an obligation of the form event-condition-action, there is no explicit event specification to trigger the execution of the actions. Instead it is assumed that an implicit event such as a particular traffic flow, or a user request will trigger the policy rule. In addition, although the IETF are considering the specification of admission control policies, the above rule-based approach is not suitable for specifying such policies. Simple admission control policies can be specified by using an action to either allow or deny a request if the condition of the policy rule is satisfied. The following are simple examples of the types of rules administrators may want to specify. The first rule assures the bandwidth between two servers that share a database, directory and other

information. The second rule gives high priority to multicast traffic for the corporate management sub-network on Monday nights from 6:00pm to 11:00pm, for important (sports) broadcasts:

```
if ((sourceIPAdress = 192.168.12.17 AND destinationIPAdress = 192.168.24.8) OR
    (sourceIPAdress = 192.168.24.8 AND destinationIPAdress = 192.168.12.17)) then
    set Rate := 400Kbps

if ((sourceIPSubnet = 224.0.0.0/240.0.0.0) AND (timeOfDay = 1800-2300) AND
    (dayofweek = Monday)) then
    set Priority := 5
```

The IETF do not define a specific language to express network policies but rather a generic object-oriented information model for representing policy information following the rule-based approach described above, and early attempts at defining a language [Strassner et al. 1998] have been abandoned. The **policy core information model** (PCIM) [Moore et al. 2001] extends the common information model (CIM) [DMTF 1999a] defined by the DMTF with classes to represent policy information. The CIM defines generic objects such as managed system elements, logical and physical elements, systems, service, users, etc, and provides abstractions and representations of the entities involved in a managed environment including their properties, operation and relationships. The specification of information models is important to enable a common way of specifying policy. However, it is independent of the actual policy specification task, and does not solve the problem of specifying policies. Apart from the PCIM the IETF are defining an information model to represent policies that administer, manage, and control access to network QoS resources for integrated and differentiated services QoS management [Snir et al. 2001]. The philosophy of the IETF is that business policies expressed in high-level languages, combined with the network topology and the QoS methodology to be followed, will be refined to the policy information model, which can then be mapped to a number of different network device configurations. Vendors following the IETF approach are using graphical tools to specify policy in a tabular format and automate the translation to PCIM. We provide an overview of commercial tools in Section 2.4.

Figure 2.5 shows the classes defined in the PCIM and their main associations. Policy rules can be grouped into nested policy groups to define policies that are related in any application specific way, although no mechanism exists for parameterising rules or policy groups. Note that both the actions and conditions can be stored separately in a policy repository and reused in many policy rules. A special type of condition is the time-period over which the policy is valid. The *PolicyTimePeriodCondition* class covers a very complex specification of time constraints.

Policy rules can be associated with a priority value to resolve conflicts between conflicting rules. This approach is not scalable in large networks with a large number of rules specified by a number of different administrators. In addition policy rules can be tagged with one or more roles. A role represents a functional characteristic or capability of a resource to which policies are applied, such as backbone interface, frame relay interface, BGP-capable router, web-server, firewall, etc. The use

of role labels is essentially used as a mechanism for associating policies with the network elements to which the policies apply.

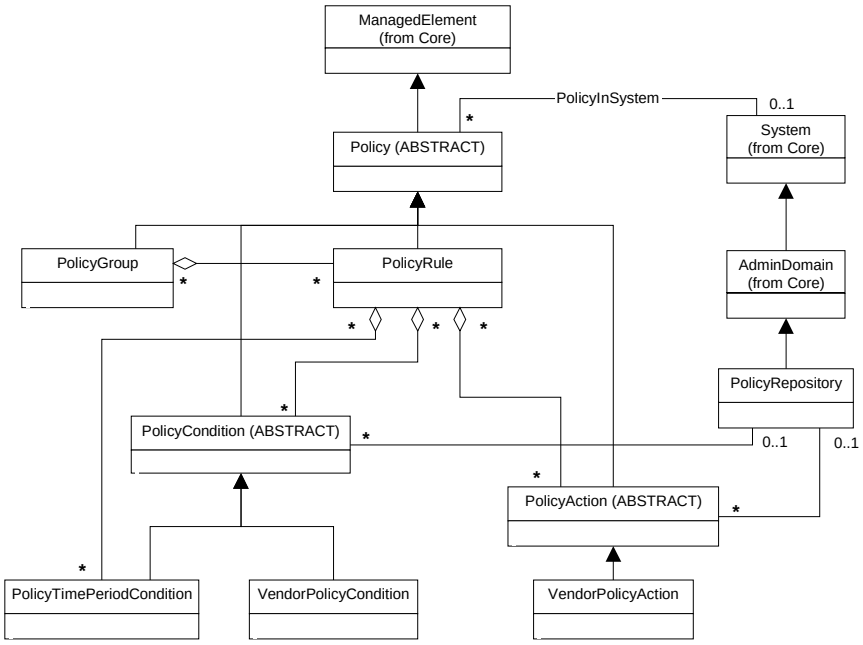


Figure 2.5 IETF policy core information model

An advantage of the information modelling approach followed by the IETF is that the model can be easily mapped to structured specifications such as XML which can then be used for policy analysis as well as distribution of policies across networks. The mapping of CIM to XML is already undertaken within the DMTF [DMTF 1999b]. The IETF define a mapping of the PCIM to a form that can be implemented in a directory that uses LDAP as its access protocol [Strassner et al. 2002].

Other approaches to network policy specification try to extend the IETF rule-based approach to specify traffic control using a concrete language. An example is the **path-based policy language** (PPL) from the Naval postgraduate school described in [Stone et al. 2001]. The language is designed to support both the differentiated as well as the integrated services model and is based on the idea of providing better control over the traffic in a network by constraining the path (i.e. the links) the traffic must take. The rules of the language have the following format:

```
policyID <userID> @{paths} {target} {conditions} [{action_item}]
action_item = [{condition}:] {actions}
```

Action_items in a PPL rule correspond to the *if-condition-then-action* rule of the IETF approach. The informal semantics of the rule is: "*policyID created by <userID> dictates that target class of traffic may use paths only if {conditions} is true after action_items are performed*". The following are examples of PPL rules from [Stone et al. 2001]:

```
Policy1 <net_manager> @ {<1,2,5>} {class = {faculty}} {*} {priority := 1}
Policy2 <Betty> @ {<1,*,5>} {traffic_class = {accounting}} {day != Friday : priority := 5}
```

Policy1 states that the path starting at node 1, traversing to node 2, and ending at node 5 will provide high priority for *faculty* users. *Policy2* uses the wild-card character to specify a partial path. It states that, on all paths from node 1 to node 5, accounting class traffic will be lowered to priority 5 unless it is a Friday. In this policy the *action_items* field is used with temporal information to influence the priority of a class of traffic.

Note that the use of the *userID* is not needed in the specification of the rules, and unnecessarily complicates the grammar. The ID of the creator of a policy, as well as information such as the time of the creation, or other priority labels attached to a rule are better specified as meta-information that could be used for the analysis of policies. PPL does not provide any way of composing policies in groups, and there is no use of roles either.

2.3 Policy Management Architectures

Figure 2.6 shows the policy-based management architecture defined within the IETF framework, which is being used as the basis for other efforts at designing policy architectures, including those by most commercial vendors. *The policy management tool* is expected to provide a graphical user interface to allow administrators to specify the policies that are active in the network, translate the input into an LDAP schema and store them in the policy repository. However, the tool can also be used to determine the association between the policies and the different devices to which the policies are applicable, or monitor the changes in stored policies and inform the relevant policy consumers. The *policy repository* is used to store the policies generated by the management tools. It is assumed that policies are objects stored in an LDAP (Lightweight Directory Access Protocol) directory service.

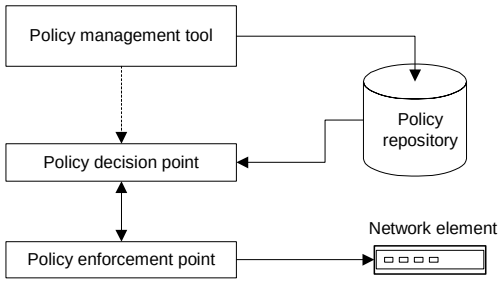


Figure 2.6 IETF policy architecture

A *policy decision point* (PDP), also referred to as a policy consumer, retrieves policies from the policy repository, interprets the policies and sends them to *policy enforcement points* (PEP) (e.g. routers, bridges) to enforce them. A PDP may need to translate the set of rules it receives from the repository to a format that is understood by the corresponding PEP's. It uses the policy role attribute of a policy rule to identify the policy enforcement points it needs to send the rule to. Other

functions of the PDP include receiving policy decision requests from PEPs and returning policy decisions to them. PDPs also send asynchronous policy decisions based on updates or external requests. Policy enforcement involves the PEP applying actions according to the PDP's decision and based on current network conditions. These conditions can be static (source/destination IP address) or dynamic (current bandwidth availability, time of the day). A PEP enforces the policy for example by permitting/forbidding requests or allocating packets from a connection to a particular queue. A PEP and PDP could be combined into a single component. Current work within the IETF concentrates on the protocols to be used between different components with most of the efforts focusing on LDAP for storing policies, and the common open policy service (COPS) protocol for communication between a PDP and a PEP.

Note that there is no explicit event to trigger an IETF policy, so an implicit event such a packet arrival at a router may trigger the search for applicable policies. This is done using the characteristics of the packets, or the roles attached to the device. However, it is only practical for the PEP to query the PDP for a decision on comparatively infrequent packets. The IETF architecture supports the distribution of policies using both: a push-model, from the PDP's to the PEP's, and a pull-model whereby the PEP's request policies based on the implicit events in the system. In addition, the automation of the distribution of policies based on changes in the network topology, changes in the characteristics of the network devices, or the policy rules themselves is also not adequately addressed.

Verma [Verma 2001] provides a detailed description of the concepts and their implementation within the IETF framework including policy validation and translation algorithms, policy distribution mechanisms and policy enforcement point algorithms.

2.3.1 Security Architectures

Architectures for deploying security policies within the role-based and the trust management communities have been reported, with emphasis on authentication techniques to support credential-based authorisation. The **open architecture for secure interworking** (OASIS) originating at the University of Cambridge [Hayton et al. 1998; Hine et al. 2000] is an example of the category of architectures which address the issue of enforcing access control policies based on roles, where the access rights of a principal are grouped in roles to which a principal can be assigned using credentials. The architecture is based on the RDL language described in Section 2.2.1. Others are concentrating on implementing the RBAC models on existing middleware platforms. In [Beznosov et al. 1999], they discuss the issues related to implementing RBAC models on the CORBA security service [OMG 2001], and they conclude that this is possible with minimal extensions to the CORBA security service.

Work is being carried out in relation to the KeyNote management system at the University of Pennsylvania to define the **Strongman** security policy architecture [Keromytis et al. 2001]. The philosophy of the Strongman architecture is illustrated in Figure 2.7. The interesting aspect of the architecture is the compilation of various global high-level policy specifications into a common lower-level *policy interoperability layer* implemented using KeyNote. The common policy layer is used to implement the high-level policies onto a variety of mechanisms and network devices. This idea identifies two important requirements: (i) the need for a common language (i.e. the policy interoperability layer in this architecture) which will be used for the uniform enforcement of policies onto a variety of mechanisms, and (ii) the need to enforce policies specified at a higher level of abstraction than a credential-based authorisation system.

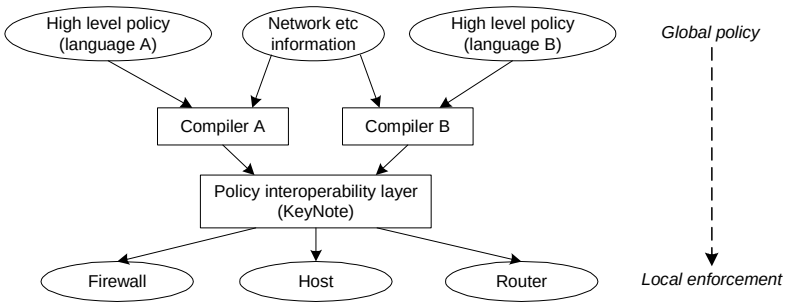


Figure 2.7 The Strongman security architecture

Note that the security architectures briefly described above, largely ignore issues related to the storage of policies in repositories and the distribution of policies to their enforcement components in large-scale distributed systems.

2.4 Tool Support

Verma [Verma et al. 2001] describes a QoS tool used to specify Service Level Agreements (SLAs) and to manipulate SLA related information in a tabular format. The tool transforms high-level policy information into device configurations, and stores them in an LDAP directory. Another tool, presented in [Mont et al. 1999b], focuses solely on template-based refinement of policies from high-level goals.

Existing work within the RBAC community is limited to specifying access control configurations in terms of roles. A centralised tool, presented in [Thomsen et al. 1998], translates access control configuration from the RBAC framework to the target's native security mechanism, which is then transported to the target. Another web-based tool, presented in [Barkley et al. 1998], allows administrators to specify roles, role hierarchies and constraints to implement RBAC for networked servers using Web protocols in order to manage access to an organisation's Web information.

In policy-based networking most of the tool support comes from industry and is based on the IETF policy framework. The majority of the commercial tools are specific to quality of service management, but many also include access control configuration. The list of vendor products is very big but one could list the following major commercial policy-based network management products: Nortel Optivity policy services, Orchestream, HP Openview PolicyXpert, Cisco CiscoAssure policy networking, Allot communications NetPolicy, IPHighway open policy system, Lucent technologies RealNet rules, SolSoft Visual Policy Management for Network Security, Novell ZENworks, Computer Associates Infrastructure Management and eTrust solutions and Tivoli Management Framework product suite. Surveys of commercial products and comparisons are available on the web (see *http://www-dse.doc.ic.ac.uk/Research/policies* for more information).

A common feature in commercial tools is a graphical user interface which typically allows the administrator to visually select a network device or other managed element from a hierarchically arranged tree-view of policy targets, and specify the policies in the form of *if* <*condition*> *then* <*action*> rules for the selected targets. The different products allow the specification of varying degrees of conditions in policy rules including a number of time attributes, source or destination IP addresses, IP type service, TCP and UDP port numbers, as well as higher level user-defined data, and allow the user to permit or deny traffic based on those conditions.

An important effort common to some of the solutions is work towards support of multi-vendor platforms, which is not adequately supported by most of the currently available products. In addition, the different standards protocols are implemented at varying degrees from the different vendors. A lot of the products support COPS as the main communication protocol for policy information between the components of their architecture, while others support HTTP or CLI for the configuration of routers and switches. In addition, not all vendors support LDAP for storing policies although they use directories as a major component of their products both for storing policy rules as well as network and user information, in order to enable scalability and third-party interoperability.

Support for security is also available in many of the commercial products, and includes access control configuration for firewalls and routers, Unix and Windows operating systems, as well as databases or for web-access. Some products such as Tivoli's and Computer Associates' are focusing on enterprise-level management of security for e-commerce applications and support role-based management of user access rights.

2.5 Background Work

The work described in the following chapters relies heavily on concepts, tools and techniques in policy management developed at Imperial College during past years. This section aims to describe the concepts on which this thesis relies. It is by no means extensive or complete in its description, restricting its scope to those concepts that are fundamental in the design of the policy framework presented in this thesis, and necessary for the understanding of what follows.

2.5.1 Domains

In large-scale systems it is not practical to specify policies for individual objects and so there is a need to be able to group objects to which a policy applies [Sloman et al. 1994a]. For example, a bandwidth management policy may apply to all routers within a particular region or of a particular type. An authorisation policy may specify that all members of a department have access to a particular service. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers [Sloman et al. 1994a; Sloman 1994b]. The benefits of the domain-based approach are twofold: *i)* a policy applying to a domain will propagate to its sub-domains thus applying to large numbers of objects and providing scalability and, *ii)* by moving an object from one domain to another its policies will be automatically replaced with those applying to the new domain, without the need to modify the policies or manually manage the association between policy and managed objects.

Membership of a domain is explicit and not defined in terms of a predicate on object attributes. A domain does not encapsulate the objects it contains but merely holds references to object interfaces, and it is thus very similar in concept to a file system directory but may hold references to any type of object, including a person. A domain, which is a member of another domain, is called a sub-domain of the parent domain. A sub-domain is not a subset of the parent domain, in that an object included in a sub-domain is not a direct member of the parent domain, but is an indirect member, c.f. a file in a sub-directory is not a direct member of a parent directory. An object or sub-domain may be a member of multiple parent domains. For example, in Figure 2.8 sub-domain *staff* is member of both *academic* and *employees,* which therefore overlap.

Path names are used to identify domains, e.g., domain *staff* can be referred to as */engineering-dept/academic/staff* or */engineering-dept/employees/staff* as an object may have different local names with multiple parent domains, where '/' is used as a delimiter for domain path names. Policies normally propagate to members of sub-domains, so a policy applying to domain */engineering-dept/academic/resources* will also apply to members of domains */engineering-*

dept/academic/resources/servers and */engineering-dept/academic/resources/printers*. *Domain scope expressions* can be used to combine domains to form a set of objects for applying a policy, using union, intersection and difference operators, e.g., a scope expression *@/engineering-dept/employees/secretary + @/engineering-dept/employees/admin* would apply to members of both domains and *@/engineering-dept/academic ^ @/engineering-dept/employees* applies only to the direct and indirect members of the overlap between the two domains. The '@' symbol selects all non-domain objects in nested domains. In this thesis, the domain concepts are used unchanged apart from the semantics of some of the unary operators (e.g. '@') which are slightly modified from the original semantics given in [Marriott 1997] (see Section 3.2).

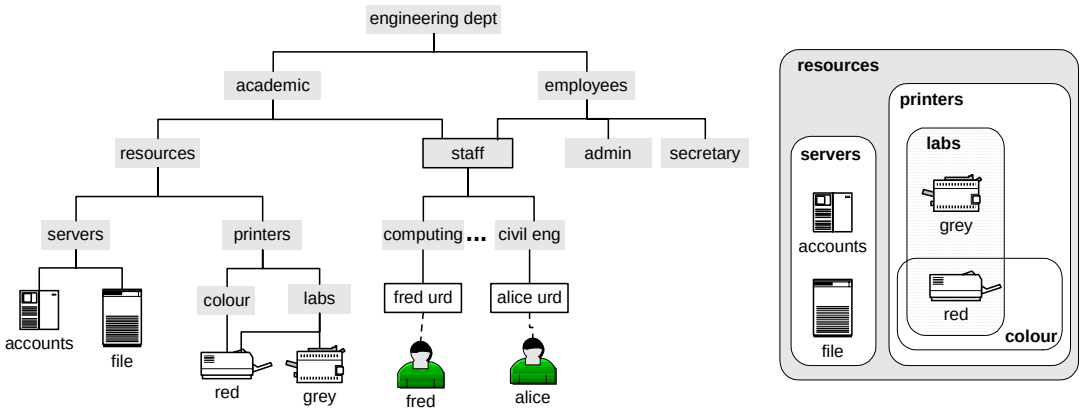


Figure 2.8 Graphical display of a domain structure

Domain Browser

The domain browser [Tonouchi 2001] provides a common user interface for all aspects of an integrated management environment. It can be used to group or select objects for applying policy, to monitor them or to perform management operations, although the current implementation only supports policy management. The domain browser reads data from the domain service and provides a graphical tree-structured view of the data. Administrators can use the domain browser to manage the domain structure, group objects into domains to apply a common policy, modify or create new objects. Objects can represent users, roles, network components or manager agents.

The domain structure can be very large, both in terms of number of objects within a domain as well as depth of the hierarchy. The domain browser implementation adopts a hyperbolic tree-mapping algorithm [Lamping et al. 1995], which allows the display of large numbers of nodes while providing effective navigation of the hierarchy (see Figure 2.9). In addition, the hyperbolic view allows the domain browser to display any part of the tree uniformly. This gives users a better feel of the entire domain structure, making it easier to perceive the context.

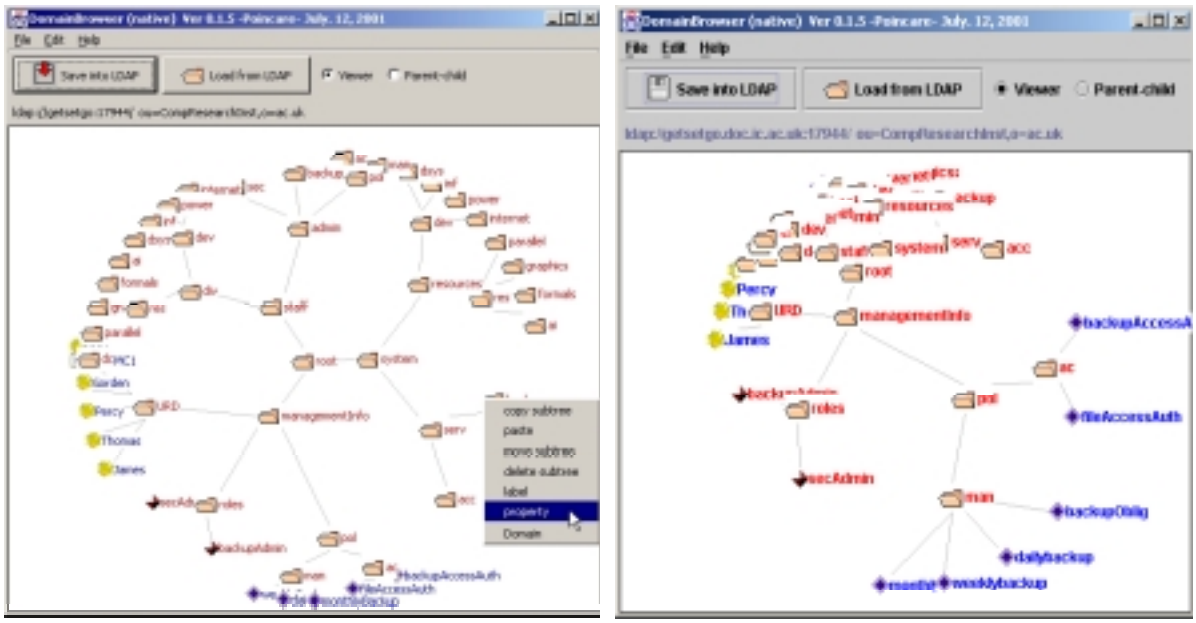


Figure 2.9 Domain browser

Navigation is realised by moving the domain tree around the hyperbolic plane. Objects nearer to the centre of the display are enlarged and come into focus. For example, an administrator can focus on the policies sub-tree of the domain structure by selecting the */managementInfo/pol* sub-tree and dragging it near the centre of the viewer, as shown on the right side of Figure 2.9. Two implementations of the domain browser with the same API are available: one uses a program module developed at Imperial College, and the other uses the Inxight Star Tree Software Development Kit [Inxight Software Inc. 2001]. Although the current version uses LDAP directories as the information repository, the browser implementation is not dependent on LDAP and data can be loaded from other sources as well.

2.5.2 Policy Concepts

Separating policies from the managers which interpret them, permits modifying policies to change the behaviour and strategy of the management system without recoding the managers. The management system can thus adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting the system down. Work at Imperial College concentrates on two types of policies: *authorisation policies* which specify what activities a subject is permitted or forbidden to perform on a set of target objects and *obligation policies* which specify what activities a subject must or must not do to a set of target objects [Sloman 1994b]. Policies within a system are specified for domains of objects. By default a policy propagates and applies to all direct and indirect members of a domain although this propagation can optionally be disabled. Policies establish a relationship between the *subjects* that perform the operations and the *target* objects on which the operations are performed. The subjects interpret obligation policies, and authorisation

policies are enforced by access control components on the target host. Note that subjects can refer to human managers, automated managers or any entity initiating operations within the system. Managers that are subjects for a set of policies may in turn, be managed by other managers according to another set of policies and thus become target managed objects.

Notation for Specifying Policies

In this section we give an overview, of the notation used to specify policies [Marriott et al. 1996; Marriott 1997]. The notation is essentially aimed at specifying policies which are interpreted by automated agents, but can also be used to specify high-level abstract policies or goals that could only be interpreted by humans. The policies are interpreted rather than compiled into the code of agents, so can be changed dynamically. The notation is precise and can be analysed for conflicts using tools, but it is not based on a well-known logic.

Authorisation policies define what activities a subject can perform on a set of target objects and are essentially access control policies to protect resources from unauthorised access. Constraints can be specified to limit the applicability of both authorisation and obligation policies based on time or values of the attributes of the objects to which the policy refers.

```
x1  A+ @/project-managers { defer(); activate() } x:@/tasks/modification_requests
when (x.status == approved)
```

Project managers are authorised to defer or activate modification requests that have been approved. The ';' is used to separate the permitted actions. Note the use of the constraint to limit the scope of applicability of the policy to objects in the target domain with status = approved.

```
x2  A- @/test-engineers { commit(); edit() } /repository/db
when (20:00 < time) or (time < 07:00)
```

Test engineers are forbidden to commit new changes or edit the repository database between the hours of 8 pm and 7 am the following day i.e. a time-based constraint. The ';' is used to separate the forbidden actions. Note, that if there is a default negative authorisation policy, whereby all actions are forbidden unless explicitly authorized, the negative authorization in x2 could be converted into a positive authorisation with a constraint when 07:00 < time< 20:00.

Obligation policies define what activities a manager or agent must or must not perform on a set of target objects. Positive obligation policies are triggered by events.

```
x3 O+ on new_request(mri) @/project1/analysts { investigate(mri); propose_solution(mri) }
/project2/tasks/modification_requests;
```

This positive obligation policy is triggered by an external event signaling that a new modification request has been issued and obliges the analysts to investigate and then propose a solution to the modification request. The ';' is used to separate a sequence of actions in an obligation policy.

```
x4  O+ at 01:00 /archiver { backup () } /repository/db
```

This positive obligation policy is triggered by an internal event – every night at 1 am – for the archiver to backup the repository database.

```
x5 O- n:@/test-engineers { DiscloseTestResults() } @/analysts + @/developers
when n.testing_sequence == in-progress
```

This negative obligation policy specifies that test engineers must not disclose test results to analysts or developers when the testing sequence being performed by that subject is still in progress, i.e., a constraint based on the state of subjects.

Negative obligation policies are not equivalent to negative authorisations. The main difference lies in the fact that obligation policies are interpreted by subjects while authorisation policies are interpreted by access control components on the target host. Thus, negative obligation policies act as subject based filters specifying actions that managers 'must refrain' from performing.

The general format of a policy is given below with optional attributes within brackets (the braces and semicolon are the main syntactic separators). Some attributes of a policy such as trigger, subject, action, target or constraint may be comments (e.g. /* this is a comment */), in which case the policy is considered high-level and not able to be directly interpreted.

```
identifier mode [trigger] subject '{' action '}' target [constraint] [exception] [parent]
[child] [xref] ';'
```

The identifier is a label used to refer to the policy. The mode of the policy distinguishes between positive obligations (O+), negative obligations (O-), positive authorisations (A+) and negative authorisations (A-). The trigger only applies to positive obligation policies. It can specify an internal timer event using an at clause, as in *x4* above, or an external event using an on clause, as in *x3* above, where the *new_request* event passes a parameter (*mri*) to the agent. Examples of external events are a temperature exceeding a threshold or a component failing. These are detected by a monitoring service. Marriott's policy notation specifies only simple events as a generalised monitoring service can be used to combine event sequences to generate simple events [Mansouri-Samani et al. 1997].

The subject of a policy, defined in terms of a domain scope expression, specifies the human or automated managers and agents to which the policies apply and which interpret obligation policies. The target of a policy, also defined in terms of a domain scope expression, specifies the objects on which actions are to be performed.

The actions specify what must be performed for obligations and what is permitted for authorisations. It consists of method invocations or a comment and may list different methods for different object types. Multiple actions in an authorisation policy indicate the set of actions or operations which are permitted or forbidden. Multiple actions in a positive obligation policy imply that they are performed sequentially after the policy is triggered.

The constraint, defined by the when clause, limits the applicability of a policy, e.g. to a particular time period as in policy *x2* above, or making it valid after a particular date (e.g. *when time > 1/June/1999*). In addition, the constraint could be based on attribute values of the subject such (as in policy *x5* above) or target objects. In *x5*, the label *n*, prepended to the subject, is referenced in the constraint to indicate a subject attribute. Constraints must be evaluated every time an obligation

policy is triggered or authorisation policy is checked to see whether the policy still applies as attribute values may change.

An action within an obligation policy may result in an operation on a remote target object. This could fail due to remote system or network failure so an exception mechanism is provided for positive obligations to permit the specification of alternative actions to cater for failures which may arise in any distributed system.

High-level abstract policies can be refined into implementable policies. In order to record this hierarchy, policies automatically contain references to their parent and children policies. In addition, a cross reference (*xref*) from one policy to another can be inserted manually, e.g., so that an obligation policy can indicate the authorisation policies granting permission for its activities.

Meta Policies

Meta-policies are policies about permitted policies, and can be used to specify application-specific constraints on groups of authorisation and obligation policies e.g., the same person must not approve payment and sign the check. There has been some experimentation for specifying meta-policies using Prolog [Lupu 1998]. The following is an example of the separation of duties principle specifying that "*No two policies can allow the same managers (subjects) to authorise a payment and sign a payment check*", written in Prolog.

```
∀ P1,P2 ∈ /policies/accounting
  P1.subjects ∩ P2.subjects ∧
  (authorise ∈ P1.actions) ∧ (sign ∈ P2.actions) ∧
  (payment ∈ P1.targets) ∧ (cheque ∈ P2.targets) ∧
  (P1.mode = P2.mode = A+) ⇒ P1 conflicts_with P2
```

2.5.3 Role-based Management Framework

A role-based framework for management of distributed systems has been defined [Lupu 1998] based on the concepts already presented in this section. This framework takes into account organisational structure to partition policy specification. Organisational structure is often specified in terms of organisational positions such as regional, site or departmental network manager, service administrator, service operator, company vice-president. Specifying organisational policies for people in terms of role-positions rather than persons, permits the assignment of a new person to the position without re-specifying the policies referring to the duties and authorisations of that position. The tasks and responsibilities corresponding to the position are grouped into a role associated with the position (which is essentially a static concept in the organisation). The position could correspond to a manager or a user of a network or services. A role is thus the position, the set of authorisation policies defining the rights for that position and the set of obligation policies defining the duties of that position. These definitions correspond to the concepts of classic role theory,

which postulates that individuals occupy positions inside an organisation and associated with the position are a set of activities (including the required interactions) that constitute the role of that position.

Organisational positions can be represented as domains and a role is considered to be the set of authorisation and obligation policies with the *position domain* as subject (as illustrated in the top of Figure 2.10). A person or automated agent can then be assigned to or removed from the position domain without changing the policies as explained in [Lupu et al. 1997b].

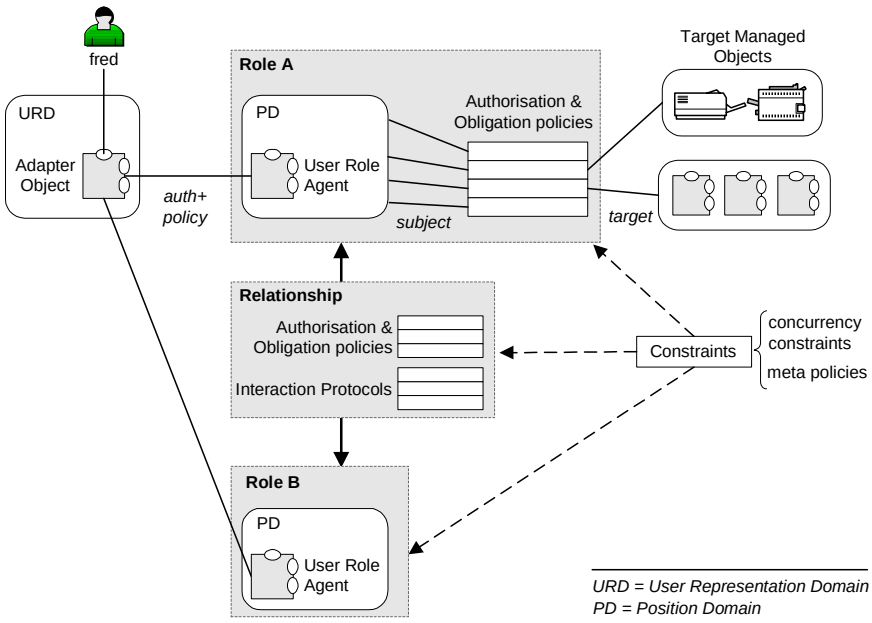


Figure 2.10 The role model

The role-based framework also contains relationships to model the ways roles of an organisation can be related to each other. A relationship may group two or more roles and specifies the rights and duties of the related parties within its scope (i.e. as authorisation and obligation policies), as well as the valid interactions between the managers acting in those roles in order to achieve their tasks, share information and agree on their commitments. Lupu defines a rule-based notation for the specification of interaction protocols between the managers assigned to roles. The notation aims to regulate the exchange of messages and enables the protocols to be changed at runtime without the need to recompile the agents.

The definition of roles and relationships as groups of policies allows the specification of global constraints on the set of policies specified in roles and their relationships. These constraints can be concurrency constraints or a variety of application-specific constraints specified as meta-policies. Lupu suggests object-oriented extensions to the specification of roles and relationships to enable reuse of specifications. Roles and relationships can be specified as classes which contain policy templates instead of actual policy instances. Policy templates are parameterised policies that may

have one or more elements unspecified or abstract. Reuse of role and relationship specifications is also achieved through inheritance. Inheritance allows a role class to be specialised with particular rights and duties and enables incremental refinement of the organisational structure for specific management needs.

The role-based management model summarised above differs from the RBAC models in two important ways [Lupu et al. 1997a]: (i) it introduces obligation policies to model the duties of the managers assigned to roles, which is not considered within RBAC models, (ii) it uses inheritance as an implementation of specialisation of role classes, whereas in RBAC models inheritance is used as a means of reusing permissions. In RBAC, inheritance is between role instances, which limits the ability to reuse role specifications by parameterising roles with the targets of the policies assigned to the role.

2.5.4 Problems

The policy specification described above suffers from problems a number of which are addressed by the work presented in this thesis.

- An important issue is that of providing a uniform language for specifying both basic and role-based policies. The work on role-based management follows a modelling approach and does not include a notation for specifying roles or relationships. However, it suggests some object-oriented extensions to the specification which can be explored further. Note that the use of classes and templates discussed in [Lupu 1998] ignores typing issues, and is rather ad-hoc and inconsistent.
- The notation described above does not provide any way of grouping policies for reusability. There is also a need for shared declarations for groups of policies.
- There was no support for specifying delegation policies discussed in [Yialelis 1996]. A delegation policy should allow the administrators to specify: (i) which access rights can be delegated, (ii) who can delegate these access rights (possible grantors), (iii) to whom the access rights can be delegated (possible grantees) and (iv) special restrictions on delegation, such as time constraints, maximum delegation period, maximum number of delegation hops, etc.
- There were a number of different types of constraints which apply to policy, and at different levels of a specification. A single notation for constraints about a set of policies (meta-policies), constraints on the activities of a policy (specified within a policy) and constraints which apply to roles e.g. a person may not activate role A and role B at the same time, should be achieved.

- The notion of a positive obligation policy allowed only simple events as triggers, because it is assumed a separate event specification tool will be used to generate simple events from complex combinations of events. However, this has proven cumbersome and there is a need to extend the notation to cope with defining event sequence compositions as triggers.
- The described notation has a rather simple use of variables whereby a constraint can select a subset of the subjects or targets to limit the applicability of the policy as in example policies *x1* and *x5* in Section 2.5.2, but a more generalised solution to the use of variables and parameters is needed. For example, a parameter received via an event should be useable within an action or constraint, or a variable which reflects current resource usage within a policy agent should be accessible within a policy.
- The described notation assumes actions are pre-loaded into policy agents and are similar to internal methods invoked from the policy level. An obligation policy interpreter would have actions predefined in order to tailor it to a particular application e.g. security, fault or configuration management. However, if an action is specified in an interpreted language such as Java, it could also be dynamically loaded into an agent in order to change or extend its functionality. An action script could specify conditional execution of sub-actions so that the execution of a sub-action could be dependent on the result of a previous one.

2.6 Conclusions

The area of policy specification for distributed systems management received considerable attention in many different areas of management. Although many approaches exist that address security and various management issues, there is a lack of policy languages that adequately cover both security and management policy specification, with built-in scalability features to enable enterprise-level management and re-use of policy specifications in different situations.

The policy specification languages can be divided into two main categories: those concentrating on security specification with emphasis on role-based access control, and those specifying the actions that must be executed in response to events which we term management policies. Security specification derives from work on formal security models and thus a lot of the approaches in this area are formal logic languages concentrating on proving properties of the security system. Although formal specifications are particularly useful in the area of security because they allow reasoning about the specified policies to enable the detection of conflicts or inconsistencies, they are generally non-intuitive and they cannot be directly translated into an implementation. High-level security policy languages concentrate more on providing the end-user with a tool for expressing policies in an environment-independent way. Recent work on trust specification

combines authorisation with authentication based on the certificates of users instead of their identities.

Management policy specification covers the areas of configuration management, network management and enterprise modelling and follows an *event-condition-action* paradigm. Most of the work on network policy management is influenced by the *if-condition-then-action* rule-based approach advocated by the IETF, which concentrates on QoS specification in IP networks. A lot of vendors are implementing policy-based management systems following the IETF framework where policies are specified using graphical tools, translated into an LDAP schema and stored in distributed policy repositories.

A lot of the work on policy-based management was triggered by the work done at Imperial College since the early 1990s. We have provided a more detailed description of this work separately from the rest of the survey, and concentrated on the concepts and techniques on which this thesis relies. We identified issues that remain to be resolved within the past work at Imperial College, and which motivated many of the ideas presented in the following chapters.

Chapter 3

Basic Policy Constructs

This chapter describes the basic features of the Ponder policy specification language. A description of the language can be found in [Damianou et al. 2001], and [Damianou et al. 2000b] is a reference guide. Ponder is declarative, and borrows features from the object-oriented world. The language is flexible, expressive and extensible to cover the wide range of requirements implied by the current distributed systems paradigms as identified in the list presented in Section 1.2. This and the next two chapters will concentrate on the policy specification language and its formal semantics, which is the centre of our management architecture. In this chapter we present only the basic policy features. Structures used to compose basic policies, and additional features of the language are presented in the next chapter.

We present the language syntax through simple examples of its use. We use the following conventions to present the syntax in this and the following chapter: Everything in **bold** is a language keyword in the figures presenting the syntax, including symbols. Choices are enclosed in round brackets ( ) separated by |, optional elements are specified with square brackets [ ] and repetition is specified with braces { }. Some features of the language are left out at this stage in order to make the discussion clearer. The complete syntax of the language is written in SableCC [Gagnon 1998] and is available in Appendix B. SableCC can be used to specify LALR(1) grammars which are a subset of LR(1) grammars and thus a subset of the context-free grammars that can be specified using BNF. The concrete syntax of the language is also described using EBNF in [Damianou et al. 2000b].

3.1 Information Model

Before describing the various elements of the language and its syntax, we first define the model we will use to represent the information that is relevant to our management framework. This includes policies as well as objects which participate in the management process either as targets or as subjects of policies. Figure 3.1 shows a partial class diagram for the information model we define in our framework. The diagram is partial because composite policy classes are not included; these will be presented in Chapter 4. We define everything as a managed object (*ManagedObject*). This

includes domains, policies and enforcement components. A domain may include any number of managed objects, for which the domain is the parent. Basic policies are defined over sets of objects formed by applying set operations, such as union, intersection and difference to the objects within domains. Subjects and targets of policies are defined in terms of domains, and this is indicated with the dependency line in the figure. Enforcement components are responsible for the enforcement of policies in the runtime management system. For authorisation policies these components are access controllers and play the role of a reference monitor at the targets' host. Policy management components are automated components responsible for enforcing subject based (obligation and refrain) policies. Basic policies are distributed to the enforcement components, hence the usage path line between the enforcement component class and the basic policy class. Details of the enforcement architecture will be presented in Chapter 7.

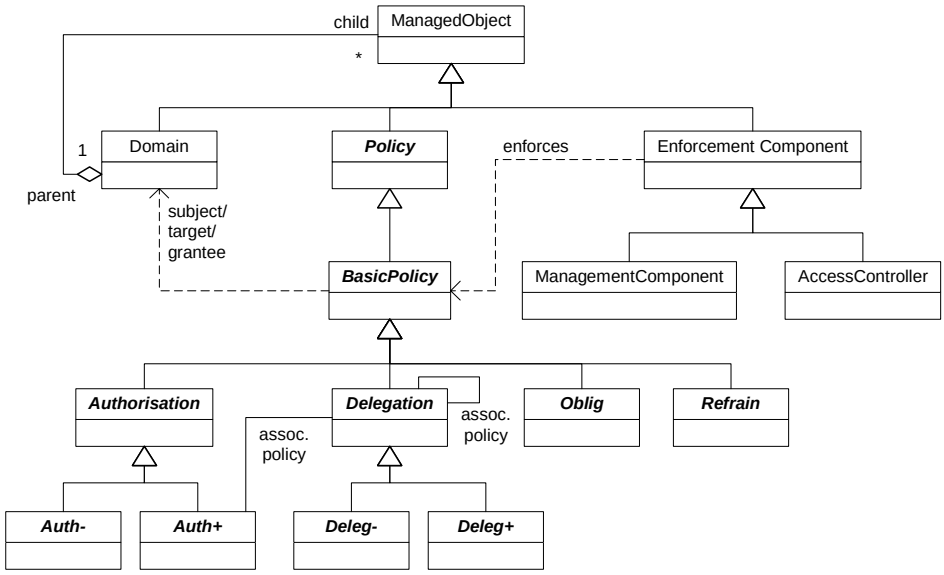


Figure 3.1 Basic policy object class hierarchy

The various subclasses of the policy class shown in the figure will be defined in the discussion of the basic policy features of the language that follows. Note that all of the policy classes are abstract. Users implicitly define their own concrete subclasses when they define a new basic policy (e.g. a new positive authorisation policy). Basic policies can be defined as parameterised types which can then be instantiated, in which case these are actually subclasses of the corresponding base class of the information model. As a shortcut a user can also define a single policy instance directly. In that case, this corresponds to the instantiation of an unnamed policy type with no parameters. We define the semantics of the language in Chapter 5.

We assume that all policies relate to objects with interfaces defined in terms of methods using an interface definition language. We use the term subject to refer to users, principals or automated manager components, which have management responsibility. A subject accesses target objects (resources or service providers), by invoking methods visible on the target's interface. The

granularity of protection for access control in Ponder is thus an interface method. Authorisations refer to methods in contrast to elementary access. References to both subject and target objects are stored within domains maintained by a domain service. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers and were described in Section 2.5.1. This facilitates policy specification for large-scale systems with millions of objects.

3.2 Domain Scope Expressions

Domain scope expressions (DSE) (introduced in [Yialelis 1996]) are used to combine domains to form a set of objects for applying a policy to (i.e. for the specification of subjects/targets of policies). We modify DSEs and extend them with domain-object library calls, to enhance their usage. Explanation of domains and their use in partitioning the objects in the system for management purposes has been described earlier in Section 2.5.1. The set of objects (i.e. the domain scope expression) to which a policy applies is evaluated each time that the policy is interpreted because domain membership can change dynamically. Note: in practice, implementation optimisations are used to minimise run-time evaluation. We cover details of the deployment architecture in Chapter 7.

Figure 3.2 shows the syntax we adopt for domain scope expressions in this thesis, and in Table 3.1 we explain the different domain scope expression operators. The set union, difference and intersection operators have equal precedence and are evaluated left to right. The unary operators '*' and '@' have higher precedence. Note that the semantics of the unary operators have been modified from the original semantics given in [Marriott 1997].

```
domain_scope_expression =
     domain_object                                                    |
     { domain_object }                                                |
     * [int_value] domain_object                                      |
     @ [int_value] domain_object                                      |
     ( domain_scope_expression )                                      |
     domain_scope_expression (+ | - | ^) domain_scope_expression     ;

domain_object := (identifier | path) (.  actionCall) | (-> featureCall) } ;
```

Figure 3.2 Domain scope expressions syntax

featureCall is an action call on collections defined in the OCL version 3 specification. It is included in domain scope expressions to allow the selection of subsets for subject and target specifications, as we will show in examples presented in this chapter.

Domain scope expressions are composed of domain-objects, which can be:

- A path

- A name defined within scope, of type domain or set, that is assigned a domain path
- A domain library call on a name of type domain declared within scope; such a call evaluates to a domain. E.g. *myDomain.get("b/c")* evaluates to the domain b/c relative to the domain already assigned to the name *myDomain*.

Syntax	Explanation
D	Returns all non-domain members of the domain-object *d* and all distinct non-domain members of all nested sub-domains recursively traversed all levels down the domain structure.
@d *@nd*	If *d* is a domain, returns a set that contains all non-domain members of the domain. The integer constant *n* specifies that the domain structure is to be traversed *n* levels down, e.g. n = 1 specifies only direct members, whereas n = 2 would include distinct members of the sub-domains of *d* also. If *n* is omitted, all nested sub-domains are recursively traversed. If *d* is a non-domain object, returns a set that contains the non-domain object.
**d* **nd*	Returns a set that contains all non-domain and all domain members of the domain *d*, including the domain itself. The integer constant *n* specifies that the domain structure is to be traversed *n* levels down. If *n* is omitted, all nested sub-domains are recursively traversed.
{c}	Returns a set that contains the object *c*.
a+b	Returns a set that contains all distinct members of *a* and *b* (Set Union).
a^b	Returns a set that contains only members that are in both *a* and in *b* (Set Intersection)
a-b	Returns a set that contains members of *a* that are not also in *b* (Set difference)

Table 3.1 Domain scope expressions

3.3 Access Control Policies

Access control is concerned with limiting the activity of legitimate users who have been successfully authenticated [Abrams 1993; Sandhu et al. 1994]. Our emphasis has been on non-discretionary access control (as defined in [Abrams 1993]), where administrators have the authority to specify security policies that are enforced by the access control system. Delegation and propagation of authority are permitted only within the scope defined by the security policy. However, this does not exclude the use of our language to specify discretionary or mandatory security policies. Ponder supports access control by providing authorisation, delegation, and information filtering policies as described below. We use the term *access control policies* to refer to all of these types of policies. We assume that policies relate to objects with interfaces defined in terms of methods using an interface definition language. The granularity of protection in Ponder is thus an interface method in contrast to elementary access (e.g. read, write, append).

3.3.1 Authorisation Policies

Access rights (often called permissions or privileges) in Ponder are specified using authorisation policies. Authorisation policies define what activities a member of the subject domain can perform on the set of objects in the target domain in terms of interface method calls. A positive

authorisation policy defines the actions that subjects are permitted to perform on target objects. A negative authorisation policy specifies the actions that subjects are forbidden to perform on target objects. Authorisation policies are implemented on the target host by an access control enforcement component (an access controller), traditionally called a reference monitor.

```
inst ( auth+ | auth– ) policyName {
    subject   [<type>] domain-Scope-Expression ;
    target    [<type>] domain-Scope-Expression ;
    action             action-list ;
    [when              constraint-Expression ; ]
}
```

Figure 3.3 Authorisation Policy Syntax

The syntax of an authorisation policy is shown in Figure 3.3. Constraints are optional in all types of policies and can be specified to limit the applicability of policies based on time or values of the attributes of the objects to which the policy refers. Constraints are discussed in detail in Section 3.3.2. Elements of a policy can be specified in any order. Note that the subject and target elements can optionally include the interface specification reference within the specified domain-scope-expression on which the policy applies. This can be used to check that the objects do support the specified operations or to locate the interface specification. The name of a policy can be specified as a path, thus identifying the domain into which the policy will be stored.

```
inst auth+ switchProfileOps  {
    subject            /NetworkAdmin ;
    target <ProfileT>  /Nregion/switches ;
    action             load(), remove(), enable(), disable() ;
}
```

Members of the NetworkAdmin domain are authorised to load, remove, enable or disable objects of type ProfileT in the Nregion/switches domain.

```
inst auth– /negativeAuth/testRouters  {
    subject /testEngineers/trainee ;
    action  performance_test() ;
    target  /routers ;
}
```

Trainee test engineers are forbidden to perform performance tests on routers. The policy is stored within the /negativeAuth domain.

The above examples show direct declaration of policy instances using the keyword *inst*. The language provides reuse by supporting the definition of policy types to which any policy element can be passed as formal parameter. Multiple instances can then be created and tailored for the specific environment by passing actual parameters. Figure 3.4 shows the syntax for authorisation policy types and instantiations.

```
type ( auth+ | auth– ) policyType  ( formalParameters )  {
    subject   [<type>] domain-Scope-Expression ;
    target    [<type>] domain-Scope-Expression ;
    action             action-list ;
    [when              constraint-Expression ; ]
}

inst ( auth+ | auth– ) policyName = policyType( actualParameters );
```

Figure 3.4 Authorisation Types and Instantiations

The authorisation policy *switchProfileOps* shown previously can be specified as a type with the subject and target given as parameters as shown in the following example.

```
type auth+ ProfileOpsT (subject s, target <ProfileT> t) {
    action load(), remove(), enable(), disable() ;
}

inst auth+ switchProfileOps = ProfileOpsT(/NetworkAdmins, /Nregion/switches);
inst auth+ routersProfileOps = ProfileOpsT(/QoSAdmins, /Nregion/routers);
```

The two instances allow members of /NetworkAdmins and /QoSAdmins to execute the actions on profile objects within the /Nregion/switches and /Nregion/routers domains respectively.

The *ProfileOpsT* authorisation policy type specifies the subject and target elements of the policy as formal parameters. This is a shortcut for specifying them in the body of the policy type. The following type specification is the same as *ProfileOpsT*.

```
type auth+ ProfileOps2T (set s, set t) {
    subject           s;
    target <ProfileT> t;
    action            load(), remove(), enable(), disable() ;
}
```

The set type is used for sets of objects and is type compatible with a DSE since DSEs always result in sets of objects when evaluated at runtime.

It can be argued that the specification of negative authorisation policies complicates the enforcement of authorisation in a system. However, there are reasons to support the provision for negative authorisation policies. Administrators often express high-level access control in terms of both positive and negative policies; retaining the natural way people express policies is important and provides greater flexibility. Negative authorisation policies can also be used to temporarily remove access rights from subjects if the need arises. In addition, many systems support negative access rights (e.g., Windows NT/2000). The existence of both positive and negative authorisation policies in a system may result in conflicts. Although this adds the need to analyse policies for conflict detection, this kind of conflict may indicate potentially unforeseen problems with the specification. For a discussion on conflicts between policies see [Lupu et al. 1999]. The usefulness of negative access rights for discretionary access control policies, has been acknowledged by other researchers [Samarati et al. 2000].

3.3.2 Basic Policy Constraints

An important element of each policy is the set of conditions under which the policy is valid. This information must be explicit in the specification of the policy. A subset of the Object Constraint Language (OCL) [OMG 1999b] is used to specify constraints in Ponder. OCL is simple to understand and use, and it is declarative – each OCL expression is conceptually atomic and so the state of the objects in the system cannot change during evaluation. Basic policy constraints limit the applicability of a policy and are expressed in terms of a predicate, which must evaluate to true for

the policy to apply. Policy constraints can be considered as conjunctions of basic constraints, which can be either time or state based. The analysis of a set of policies can then be substantially improved since time-based constraints can be compared for possible overlap and state based constraints can be either simultaneously satisfied or mutually exclusive if they relate to states of the same system component. We separate the different types of constraints based on:

- Subject/target state – constraints based on the object state as reflected in terms of attributes at the object interface.
- Action/event parameters – constraints based on event parameter values in obligations or action parameter values in authorisations or refrains. We look at obligation and refrain policies in Section 3.4.
- Time – constraints which specify the validity periods for the policy. A time library object is provided with the language to specify time constraints.

In the specification of constraints, *Time* is a predefined object on which operations such as *between*, *before* or *after* can be invoked related to the current time. The policy compiler can resolve the different types of constraints at compile time and separate the constraints in order to aid in the analysability of policies. More information on the runtime representation of policy objects and the implementation of the compiler are described later (Chapter 6).

```
inst auth- testRouters {
    subject   s =/testEngineers;
    action    performance_test();
    target    /routers;
    when      s.role = "trainee";
}
```

TestEngineers cannot execute performance tests on routers if they are trainee testEngineers. The role attribute of the subject is used in the constraint.

```
inst auth+ videoConf1 {
    subject   /Agroup + /Bgroup;
    target    USAStaff – NYgroup;
    action    VideoConf(BW, Priority);
    when      Time.between("1600", "1800") and (Priority > 2);
}
```

Members of Agroup plus Bgroup can set up a video conference with USA staff except the New York group. The constraint of the policy is composite. The time-based constraint limits the policy to apply between 4:00pm and 6:00pm and the action constraint specifies that the policy is valid only if the priority parameter (the 2nd parameter of the action) is greater than 2.

3.3.3 Information Filtering

Filtering policies are needed to transform the values of the input parameters in an action and the information returned from the action. For example, a location service might only permit access to detailed location information, such as a person is in a specific room, to users within the department. External users can only determine whether a person is at work or not. Some databases support similar concepts of ‘views’ onto selective information within records – for example a payroll clerk

is only permitted to read personnel records of employees below a particular grade. Positive authorisation policies may include filters to transform input parameters associated with their actions, based on attributes of the subject or target or on system parameters (e.g., time). In many cases it is not practical to provide different operations as a means of selecting the information. Although these are a form of authorisation policy they differ from the normal ones in that it is not possible for an external authorisation agent to make an access control decision based on whether or not an operation, specified at the interface to the target object, is permitted. Essentially the operation has to be performed and then a decision made on whether to allow results to be returned to the subject or whether the results need to be transformed. Filters can only be applied to positive authorisation actions.

```
actionName { filter }
filter = [ if condition ] {  { (
       in parameterName = expression ;  |
       result = expression ; ) }  }
```

Figure 3.5 Filters on Positive Authorisation Actions

Every action can be associated with a number of filter expressions (see Figure 3.5). Each filter contains an optional condition under which the filter is applied. If the condition evaluates to true, then the transformations (the assignment statements in the body of the filter) are executed. The *in* keyword is used to indicate an input parameter of the action on which the filter is specified; *result* is used to transform the return value of the action.

```
inst auth+ filter1  {
     subject  /Agroup + /Bgroup ;
     target   USAStaff – NYgroup ;
     action   VideoConf(BW, Priority)
                      if (Time.after("1900")) {in BW=3; in Priority = 1; }
                      { in BW=2 ; in Priority=3 ; }     // default filter
}
```

Members of Agroup plus Bgroup can set up a video conference with USA staff except the New York group. If the time is later than 7:00pm then the video conference takes parameters: bandwidth = 3 Mb/s, priority = 1. Otherwise the filter restricts the parameters to bandwidth = 2 Mb/s, priority = 3.

The following is a more elaborate example. Consider a hypothetical class-diagram of the information stored in a departmental server shown in Figure 3.6. The getEmp(ssn) method returns an *Employee* object given its ssn-number. Assume there is an authorisation policy authorising subjects to execute the method *getEmp(ssn)* on objects of type *Department* on the departmental file server. Depending on the subject of the authorisation, there is a filter that allows the subject to see only part of the information returned:

- The general manager can see all of the information.
- The departmental manager cannot see the agenda of the employee.
- Another fellow employee cannot see the salary, his agenda and the budget of the projects to which the employee is assigned.

- A person outside the organisation can see only the name, project names and meeting topics of the employee.

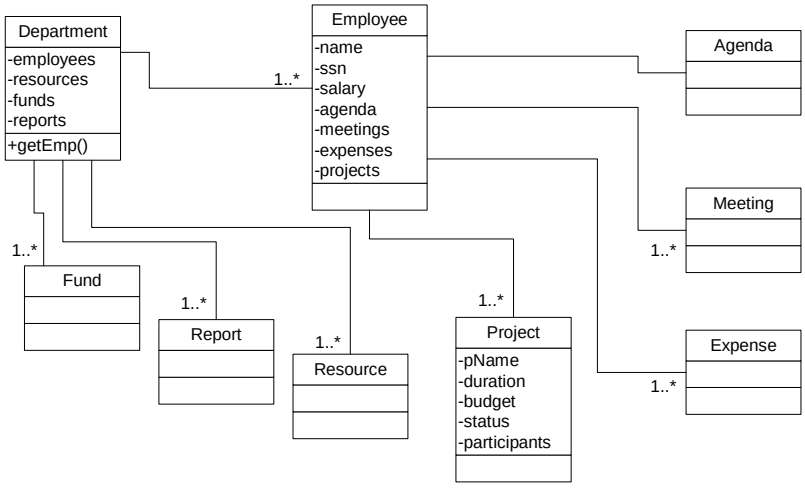


Figure 3.6 Partial departmental information class diagram

Here are the authorisation policies to specify this.

```
inst auth+ GMgetEmployeeAuth {
     subject  General_Manager;
     target   DeptFile_Server;
     action   getEmp(ssn);
}

inst auth+ DMEmployeeAuth {
     subject  Dept_Manager;
     target   DeptFile_Server;
     action   getEmp(ssn) {result = result->reject(agenda);};
}

inst auth+ employeeAuth {
     domain   e = /employees;
     domain   other = /external;

     subject  e + other;
     target   DeptFile_Server;

     action getEmp(ssn)
              if (subject = e) {
                    result = reject(result, salary, agenda, projects.budget);
              }
              if (subject <> e) {
                    result = select(result, name, project.pname, meeting.topic);
              }; // getEmp
} // employeeAuth
```

Note the use of the functions *reject* and *select* which reject or select respectively, certain attributes from their first parameter, in this case the result of the action execution. We assume that these actions are available as part of a standard library.

3.3.4 Delegation Policies

Delegation is often used in access control systems to cater for the temporary transfer of access rights. However the ability of a user to delegate access rights to another must be tightly controlled by security policies. This requirement is critical in systems allowing cascaded delegation of access

rights. A delegation policy permits subjects to grant privileges, which they possess (due to an existing authorisation policy), to other subjects called grantees to perform an action on their behalf e.g., passing read rights to a printer spooler in order to print a file. Delegation in Ponder does not transfer access rights from grantors to grantees; grantors continue to retain their access rights after a delegation is performed.

```
inst deleg+  ( associated-policy-name  ) policyName  {
     grantee    [<type>]  domain-Scope-Expression ;
     [ subject  [<type>]  domain-Scope-Expression ; ]
     [ target   [<type>]  domain-Scope-Expression ; ]
     [ action             action-list ; ]
     [ when               constraint-Expression ; ]
     [ valid              constraint-Expression ; ]
     [ hops               int-value ; ]  }
```

Figure 3.7 Delegation policy syntax

A delegation policy is always associated with an authorisation policy, which specifies the access rights that can be delegated. Negative delegation policies forbid delegation of certain actions. Note that delegation policies are not meant to be used for assignment of rights by security administrators.

Figure 3.7 shows the syntax of a positive delegation policy. The only required part in the body of the policy is the *grantee*. The rest of the parts (subject, target, action) must be subsets of those in the associated authorisation policy; if not specified they default to those of that policy.

Constraining Delegation

Delegation constraints specify restrictions on when the delegation performed is valid, or on when a cascaded delegation is valid. Only positive delegation policies contain delegation constraints; they make no sense in negative delegation policies. The syntax for a negative delegation policy is thus the same as that for a positive policy, without the *valid* and *hops* clauses, which are used to specify delegation constraints.

Delegation constraints are:

- Time restrictions to specify the duration or the period over which the delegation should be valid before it is revoked.
- Any arbitrary constraint based on system attributes or subject/target/grantee or action attributes.
- Maximum number of cascading delegations allowed (maximum number of delegation hops or levels)

The first two types of constraints are specified in the valid attribute of the delegation policy whereas the maximum number of cascading delegations allowed is specified in the hops attribute. See the example that follows.

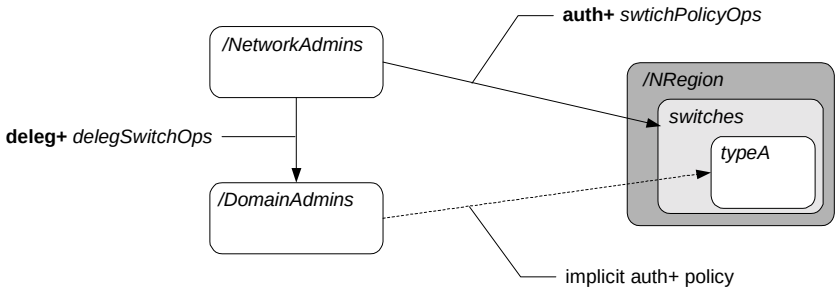


Figure 3.8 Delegation policy example

```
inst deleg+ (switchProfileOps) delegSwitchOps  {
     grantee    /DomainAdmin ;
     target     /Nregion/switches/typeA ;
     action     enable(), disable();
     valid      Time.duration(24) ;
}
```

The above delegation policy is associated with the *switchProfileOps* auth+ policy from Section 3.3.1. It states that the subject of that authorisation policy (NetworkAdmin), which is implicit in this policy, can delegate the enable and disable actions on policies from the domain /Nregion/switches/typeA to grantees in the domain /DomainAdmin. Note how the policy restricts the target to a subset of the switchProfileOps policy target (See Figure 3.8). The valid-clause, specifies that the delegation is only valid for 24 hours from the time of creation; after that it must be revoked.

A delegation policy specifies the authority to delegate, it does not control the actual delegation and revocation of access rights. Delegation policies map to authorisation policies and can be implemented as authorisations. We formalise the mapping of delegation to authorisation policies in Chapter 5.

Cascaded Delegation

During runtime, cascaded delegation is allowed provided that both the grantor and the grantee are in the grantee scope of the delegation policy. This is demonstrated in Figure 3.9; an authorisation policy allows objects in the *subject scope* to execute actions on objects in the *target scope*. A delegation policy allows objects in the *subject scope* to delegate the access rights assumed from that authorisation policy to objects in the *grantee scope*.

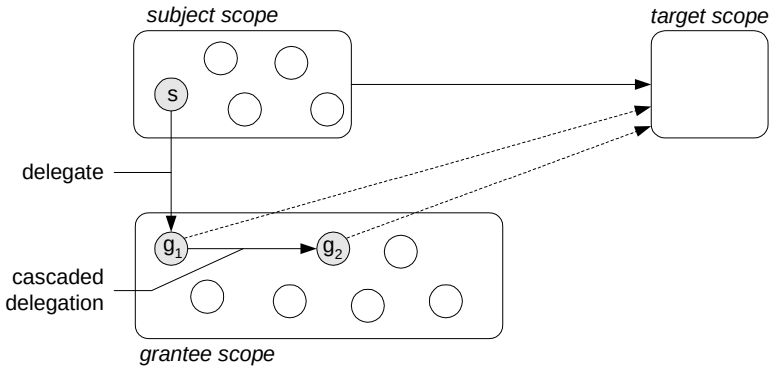


Figure 3.9 Cascaded delegation

When the subject *s* executes the delegation of access rights to an object g_1, g_1 is automatically given the right to also execute the delegated actions on objects in the *target scope*. Furthermore, g_1 can delegate these access rights to another object g_2 within the *grantee scope* only. We call this

cascaded delegation, and each cascaded execution of the delegate action counts as one delegation hop.

Cascaded delegation can also take place at specification time. This is the case when a delegation policy is passed as a parameter to another delegation policy. This specifies that any grantee who has been delegated some access rights based on the first delegation policy, can delegate them further to another grantee in a second grantee scope as specified by the second delegation policy.

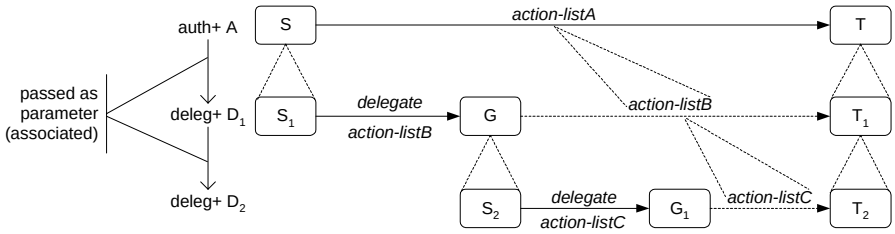


Figure 3.10 Relation between authorisation and delegation policies

Figure 3.10 summarises the relation between authorisation and delegation policies and their elements. The top of the figure shows a positive authorisation policy A, with subject S target T and a list of actions action-listA, the actions S can execute on T. This is called the associated policy and is passed as a parameter to a positive delegation policy D_1 which allows objects in S_1 to delegate actions within the action-listB on objects in T_1, to objects in G. An actual delegation of access rights during runtime would then create an implicit authorisation policy authorising G to execute actions within action-listB to target objects T_1. Note that the following are true: $S_1 \subseteq S$, $T_1 \subseteq T$, action-listB $\subseteq$ action-listA. Finally, D_1 can be used as the associated policy for a second delegation policy D_2 authorising objects in S_2 to delegate actions within the action-listC on objects in T_2, to objects in G_1. The following relations are also true: $S_2 \subseteq G$, action-listC $\subseteq$ action-listB.

An action being delegated may have a filter attached to it, which will be executed when the action is invoked by the grantee on the target. Delegation policies are not allowed to override filters on delegated actions. Also, the authorisation policy from which the delegation derives access rights may already have a constraint which limits its applicability. This constraint is taken into account when the delegation action is executed. The grantee can only execute the action if the original constraint of the authorisation policy from which the access rights were derived, evaluates to true. In Section 5.4 we demonstrate the enforcement of delegation policies by mapping them to authorisation policies, and show how the constraints are taken into account.

Examples

We use two slightly more extended examples to demonstrate the concepts of delegation policies. Consider the following hypothetical domain structure.

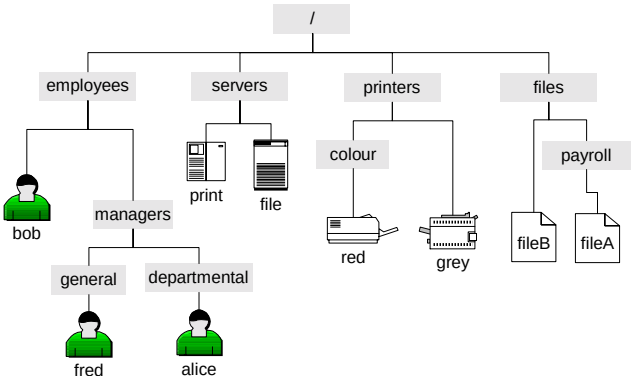


Figure 3.11 A hypothetical domain structure

Suppose that the following authorisation policies are in place:

```
domain /policies/delegation;

type auth+ fileAccess (subject S, target files) {
     action   read, write;
} // fileAccess

inst auth+ managerFileAccess = fileAccess(/Employees/Managers, /Files/PayrollFiles);
inst auth+ employeeFileAccess = fileAccess(/Employees-Employees/Managers,
                                           /Files-Files/PayrollFiles);

type auth+ printAccess (subject S, target printer) {
     action print;
} // printAccess

domain man = /Employees/Managers;

inst auth+ GMprintAccess = printAccess(man/GeneralManagers, Printers/ColorPrinters);
inst auth+ employeePrintAccess = printAccess(/Employees, /Printers-Printers/ColorPrinters);

inst auth+ fileServerAccess {
     subject  /Employees;
     target   Servers/FileServer;
     action   *;       // all actions
} // fileServerAccess

inst auth+ printServerAccess {
     subject  Employees;
     target   Servers/PrintServer;
     action   *;       // all actions
} // printServerAccess
```

In the above examples notice the use of the domain statements. The first changes the **current working domain**. Any relative domain paths specified after this statement are consideredrelative to the current working domain. So the policies following will all be stored under: /policies/delegation. The default current working domain is the root /. The second domain statement declares a constant called *man* and assigns it a path. This constant can then be used in the specification. More on declaring constants for reusing specifications will be described in Section 3.5. Here are the actual delegation policies:

```
inst deleg- invalidDeleg1 (managerFileAccess) {
     subject /Employees/Managers/DeptManagers ;
     grantee /Employees - /Employees/Managers ;
}
```

The above delegation policy specifies that departmental managers are not allowed to delegate the access rights specified by the managerFileAccess policy to employees that are not managers.

```
inst deleg- invalidDeleg2 (managerFileAccess) {
     subject  /Employees/Managers/GeneralManagers ;
     grantee  /Employees - /Employees/Managers;
     action   write;
}
```

The above delegation policy specifies that general managers are not authorised to delegate the write access right specified by the managerFileAccess policy.

```
inst deleg+ colorPrintDeleg (GMprintAccess) {
     subject  /Employees/Managers/GeneralManagers;
     grantee  /Employees/Managers/DeptManagers;
     action   print;
     when     Time.between("18:00:00", "07:00:00");
     hops     1        // do not allow cascading
}
```

Finally, this last delegation policy specifies that general managers are authorised to delegate the print access right specified by the GMprintAccess, to departmental managers. Note the use of the maximum delegation-hop constraint specified at the end of the policy following the 'hops' keyword. Since the maximum number of cascading hops allow is 1, this disallows cascaded delegation for this policy.

The following scenario (see Figure 3.12) is based on the hypothetical domain structure of Figure 3.11. The scenario is deliberately made more complicated than needed for in real situations just to demonstrate different aspects of the delegation policy. In order for the *FileServer* to be permitted to access the requested file, it must be delegated the access rights from the subject that requires the access to the file. The same is true for the *PrintServer*. In order for it to be able to print to a particular printer, it must be delegated the access right by the user requesting the print.

Now consider the following: A general manager (*fred*) wants to print a payroll file (*fileA*) on a color printer (*red*). *Fred* first needs to delegate the access right to the *PrintServer* to print on *ColorPrinters*, the right to access the *FileServer* and request a read on payroll *FileA*, and the right to access payroll files. The *PrintServer* then needs to further delegate the right to read *PayrollFiles* to the *FileServer* in order for the file server to be able to read *FileA*.

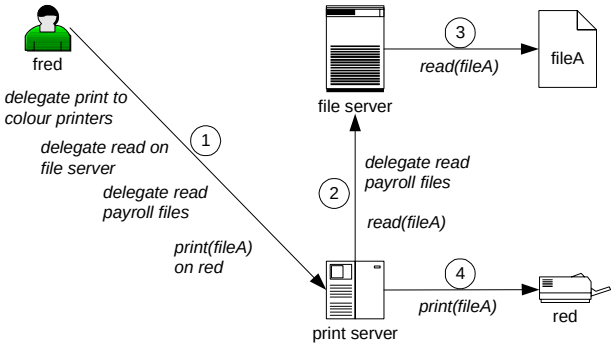


Figure 3.12 Delegation actions involved in printing a payroll file on a colour printer

The following delegation policies must then be in place in order for *Fred* to be able to print *FileA* on *red*.

```
type deleg+ GMtoPrintServerT(authPol)(action actionToDelegate) {
     subject  /Employees/Managers/GeneralManagers;
     grantee  /Servers/PrintServer;
     action   actionToDelegate;
 }
```

```
inst  deleg+ GMtoPrint1 = GMtoPrintServerT(GMprintAccess)(print);
      deleg+ GMtoPrint2 = GMtoPrintServerT(fileServerAccess)(read);
      deleg+ GMtoPrint3 = GMtoPrintServerT(managerFileAccess)(read);
```

The first delegation policy (GMtoPrint1) states that a general manager can delegate the right to print to colour printers coming from the GMprintAccess authorisation policy. The second (GMtoPrint2), that it can call the action read on the file server, and the third (GMtoPrint3) that it can read payroll files.

```
inst deleg+ printStoFileS(GMtoPrint3) {
     subject  /Servers/PrintServer;
     grantee  /Servers/FileServer;
     action   read;
}
```

The last delegation policy (printStoFileS) states that the print server can delegate the right to read payroll files to the file server. On the attempt to do so, the access control system would check that the print server has already been delegated this access right. The GMtoPrint3 delegation policy only states that a general manager is authorised to delegate to the print server the referenced access right; it does not automatically mean that the print server has that right.

3.4 Subject-based Policies

Access control is only one aspect of a policy specification system. A second aspect involves the management of subjects in the system. We are interested in the specification of policies to control the operation of automated subjects in the system; we do not deal with human users. In the following subsections we describe two additional types of policy. Obligations used to specify what subjects must do and refrains, a form of subject-based access control, to restrict the execution of these actions within the subject itself.

3.4.1 Obligation Policies

Obligation policies specify the actions that must be performed by managers within the system when certain events occur and provide the ability to respond to changing circumstances. For example, security management policies specify what actions must be specified when security violations occur and who must execute those actions; what auditing and logging activities must be performed, when and by whom. Obligation policies could relate to management of QoS, storage systems, software configuration etc.

Obligation policies can also be considered in a more general context as a constrained form of programming network elements and end-user agents. Active networks, mobile agents and management by delegation are techniques for transferring executable content to the automated agents, thus enhancing and customising agents' capabilities. These techniques offer limited or no control over the execution of the transferred code. Obligation policies are complementary to these approaches as they allow to specify what actions must be performed in response to events but are not dependent on the means used to transfer the code which implements the actions to the managers.

Obligation policies are event-triggered and define the activities automated manager components in subject domains must perform on objects in the target domains. Event-condition-action rules prove to be a very flexible approach to specifying management policy as exemplified by PDL, a language implemented and used in Lucent switching products [Lobo et al. 1999; Virmani et al. 2000]. In order to carry out an obligation, a manager must know when to perform the action, thus implementable obligation policies must explicitly specify the event on which the actions must be performed. Events can be simple, i.e. an internal timer event, or an external event notified by monitoring service components e.g. a temperature exceeding a threshold or a component failing. Composite events can be specified using event composition operators.

```
inst oblig policyName {
     on                    event-specification ;
     subject    [<type>]   domain-Scope-Expression ;
     [ target   [<type>]   domain-Scope-Expression ; ]
     do                    obligation-action-list ;
     [ catch               exception-specification ; ]
     [ when                constraint-Expression ; ]
}
```

Figure 3.13 Obligation policy syntax

The syntax of obligation policies is shown in Figure 3.13. Note the required event specification following the *on* keyword. Table 3.2 specifies the event composition operators that can be used in event expressions. All event operators have equal precedence and evaluation is strictly left to right.

Operator	Explanation
e1 && e2	Occurs when *both* e_1 and e_2 occur irrespective of their order
e + time-period	Occurs a specified period of time after the occurrence of event e
{e_1 ; e_2} ! e_3	Occurs when e_1 occurs followed by e_2 with no interleaving e_3
e_1 | e_2	Occurs when either e_1 or e_2 occurs irrespective of their order
e_1 -> e_2	Occurs when e_1 occurs before e_2
n * e	Occurs when e occurs n times, where n is an integer value

Table 3.2 Event composition operators

The target element is optional as obligation actions may be internal to the subject, whereas authorisation actions always relate to a target object. The action can be prefixed with the name of the object on which the action is called, as actions may be on the target, internal to the subject or part of the subject's interface. If no prefix is specified, the action is assumed to be internal to the subject or part of the subject's interface by default. Concurrency operators specifying whether actions should be executed sequentially or in parallel are used to separate the actions in an obligation policy. The optional catch-clause specifies an exception that is executed if the actions fail to execute for some reason. The concurrency operators for obligation policy actions are given in the following table (Table 3.3). All concurrency operators have equal precedence and evaluation is strictly left to right. Parenthesis can be used to change the default precedence.

Operator	Explanation
a_1 -> a_2	a_2 must follow a_1. If any of the actions fails or is not allowed by a refrain policy, the execution stops.
a_1 || a_2	a_1 and a_2 may be performed concurrently. Execution continues when either has finished.
a_1 && a_2	a_1 and a_2 may be performed concurrently. Execution continues when both have finished. If any of the actions fails or is not allowed by a refrain policy, the execution stops.
a_1 | a_2	a_1 is performed. If it fails or is not allowed by a refrain policy, a_2 is performed. If a_1 succeeds, execution stops.

Table 3.3 Action concurrency operators

A formal treatment of event composition operators, and the action concurrency operators, is presented in Section 5.5. The following are a few simple examples to demonstrate some of the features of obligation policies.

```
inst oblig loginFailure {
     on             3*loginfail(userid) ;
     subject        s = /NRegion/SecAdmin ;
     target <userT> t = /NRegion/users->select(t1 | t1.getId() = userid) ;
     do             t.disable() -> s.log(userid) ;
}
```

This policy is triggered by 3 consecutive loginfail events with the same userid. This is an example of a simple event composition. The NRegion security administrator (SecAdmin) disables the user with userid in the /NRegion/users domain and then logs the failed userid by means of a local operation performed in the SecAdmin object. The '->' operator is used to separate a sequence of actions in an obligation policy. Names are assigned to both the subject and the target. They can then be reused within the policy. In this example we use them to prefix the actions in order to indicate whether the action is on the interface of the target or local to the subject.

```
type oblig printFail (string msg, QueueMan qMan)  {
     on       printfail(jobid, userid, filename);
     subject  s = printManager;
     target   ms = /servers/mailServer;
     do       ms.mailto(userid, filename+msg) || s.putInQueue(qMan, jobid);
 }
```

Types external to the policy specification can be specified assuming the corresponding specifications are accessible from a type repository. This is demonstrated with the above policy. The printFail obligation type accepts two parameters one of which is an external type called QueueMan. This is an interface specification of a printer queue manager object. The qman parameter is then used as a parameter in the call to putInQueue which is local to the printManager. The use of the || concurrency operator allows the actions to be performed in parallel.

The following examples demonstrate the use of more complicated domain scope expressions based on OCL collection operations to select the subjects/targets involved in the action execution of obligation policies.

```
type oblig printJob (set S, domain T, int maxpages) {
     on       print(job, sender);
     subject  S ^ {sender};
     target   T->select(t | t.state = 'idle');
     do       print(job) -> sender.mail("job re-directed");
     when     job.pages > maxpages;
}

inst oblig backupFiles {
     domain   d = backupAdmins/;
     on       Timer.at("20:00:00");
     subject  s = d->select(s1|);
     target   /logServer;
     do       backup() -> s.log();
}
```

In the printJob policy, the sender (of the print job), executes the job on targets (printers within domain T) which are idle. The second obligation policy (backupFiles) obliges backup administrators (backupAdmins) to backup files located on the logServer every day at 8:00pm. However, we only want one of the administrators to execute the backup. The select operation on the subject domain selects only one object from the set of subject objects. This is indicated by an empty select expression. A select operation of the form: S->select(s1,s2|) would select two objects from the set S, whereas a non-empty expression after the bar would select those subject objects from which the expression evaluates to true. Note that basic policies can define constants that can be reused in the specification. The backupFiles policy defines a domain constant d, which is then used in the specification of the subject of the policy.

Scripted Actions

A script is an externally-defined code object that can be imported into a Ponder specification from a domain. An obligation action can be defined as a script using any suitable scripting language to specify a complex sequence of activities or procedures with conditional branching. Scripts provide the flexibility of including complex actions which cannot be expressed as single object method invocations and can contain conditional statements supported by the scripting language used. For example, a script could be defined to update software on all computers in a target domain as an atomic transaction which rolls back to the old version if any one of the updates fail. An action script could specify conditional execution of sub-actions so that the execution of a sub-action could be dependent on the result of a previous one. This is similar to the conditional policy rules defined in the IETF notation, which limits the condition to an ORed set of ANDed conditions or an ANDed set of ORed conditions [Moore et al. 2001]. Scripts are implemented as objects and stored in domains. Thus authorisation policies can be specified to control access to the scripts as shown in the following example:

```
inst auth+ domainManagement1 {
     subject  /domainAdmin;
     action   execute;
     target   /scripts/domainMove(A,B,x);
     when     A=”/users/gold” and B=”/users/silver”;
}

inst oblig domainMove {
     on       offensiveRequest(user);
     subject  /domainAdmin;
     do       /scripts/domainMove(”/users/gold”,”/users/silver”,user);
}
```

The authorisation policy permits domain administrators to execute the script object ‘domainMove’ when the first two parameters to it are “/users/gold” and “/users/silver”. This script moves an object x from domain A to domain B. Since this is considered to be a security sensitive operation only domain administrators are permitted to execute it, and only to downgrade gold service users to silver service users. The obligation policy specifies that domain administrators must move a user from the gold service domain (“/users/gold”) to the silver service domain (“/users/silver”) when that user has requested access to a web-page which is considered offensive.

If an interpreted language such as Java is used to program scripts, then the scripts could be updated using mobile code mechanisms to change the functionality of automated manager agents, although this suffers from all the usual security vulnerabilities of mobile code [Chess 1998]. Extending the functionality of a manager agent is an operation on the agent’s management interface and can be restricted by authorisation policies.

Scripts are primarily used as obligation policy actions, but they can also be invoked as actions in positive authorisation action-filters, in the when-clause (i.e. the constraint) of any basic policy, or as exceptions in obligation policies and meta-policies (see Section 4.5).

3.4.2 Refrain Policies

Refrain policies define the actions that subjects must refrain from performing (must not execute) on target objects even though they may actually be permitted to do so. In other words, refrain policies act as restraints on the actions that subjects perform. They have a similar syntax to negative authorisation policies, but are enforced by subjects rather than target access controllers. Refrains are used for situations where negative authorisation policies are inappropriate because the targets are not trusted to enforce the policies (e.g., they may not wish to be protected from the subject), or because it may not be possible to enforce an access control policy; e.g. the decision depends on attributes and state values of the subject performing the action. In addition, an action specified in an obligation policy might be internal to the subject as part of its interface, or a script action loaded into the agent implementing the subject. Calls on those actions can only be restricted using refrain policies because they do not involve method calls on target objects, making it impossible to control their execution using authorisation policies. The syntax of refrain policies (Figure 3.14) is the same as that of negative authorisation policies.

```
inst refrain policyName {
    subject   [<type>] domain-Scope-Expression ;
    target    [<type>] domain-Scope-Expression ;
    action             action-list ;
    [when              constraint-Expression ; ]
}
```

Figure 3.14 Refrain policy syntax

The following are simple examples of using refrain policies.

```
inst refrain testingRes {
     subject  s=/test-engineers ;
     target   /analysts + /developers ;
     action   discloseTestResults() ;
     when     s.testing_sequence = "in-progress" ;
}
```

This refrain policy specifies that test engineers must not disclose test results to analysts or developers when the testing sequence being performed by that subject is still in progress, i.e., a constraint based on the state of subjects. Analysts and developers would probably not object to receiving the results and so this policy is not a good candidate for a negative authorisation.

```
inst refrain loginFailureR {
     subject        s = /NRegion/SecAdmin ;
     target <userT> t = /NRegion/users ;
     action         t.disable() ;
     when           /NRegion/users/priviledged->exists(u | u.getId() = t.getId());
}
```

The loginFailureR refrain policy relates to the loginFailure obligation in Section 3.4.1. It instructs the subjects (security administrators) not to disable targets (users) which are part of the /NRegion/users/priviledged domain. i.e. who are priviledged.

3.5 Common Elements Specification

Elements that are common in a policy specification can be defined separately and reused. These include: events, constraints and a variety of constants. Although the examples in this section show the use of common elements in single policies, definition and reuse of event, constant and constraint specifications will be useful in larger specifications of groups of policies (see Chapter 4).

3.5.1 Event Definitions

Events in Ponder are used to trigger obligation policies. It is convenient to be able to define events separately, and re-use them in multiple obligation policies. The definition of an event may be parameterised. Event parameters map onto the event attributes, which define new names within the scope of the policy object where the event is specified. These names can then be referenced within the policy (see example that follows).

```
event timerA = Timer.at("2001:12:15", "22:17:00");
event timerB = Timer.every(“24 hours”, "07:20:00");
event circuitFailure(h,x,y) = (envAlarm(h) -> rFailure(x,y));

inst oblig resetCircuit {
     subject  /brEngineer ;
     on       circuitFailure(h,x,y) ;
     do       resetCircuit() ;
     target   /brCircuits->get(h) ;
}
```

In the above example, a Timer object for generating time-based events is used. The first event occurs at a particular date (15 Dec. 2001) and time (22:15:00), the second event occurs every 24 hours at 07:20. The third event circuitFailure(h,x,y) demonstrates the use of parameters in the definition of an event. The named event receives three parameters (h,x,y) that can be referenced in the obligation policy that uses this event. The first parameter corresponds to the parameter of the envAlarm(h) while the second and third to the two parameters of rFailure(x,y). The two events that are used in the event expression are assigned to the new event. You can see how the first parameter is used in the specification of the target in the obligation policy resetCircuit. Similarly, the timerA and timerB events can be used to trigger other obligation policies as required.

3.5.2 Constraint Definitions

Constraints used to limit the applicability of basic policies i.e. as part of the when-clause of a policy (Section 3.3.2) can also be defined separately, named and reused. Similar to events, defined constraints can be parameterised.

```
constraint active(s) = s.isActive() and s.isEnabled();
constraint workHours = Time.between(“08:00:00”, “16:00:00”);

type oblig serviceReset(subject s, target t) {
     on      e ;
     do      t.reset() ;
     when    active(s) and workHours;
}
```

In the above example, two constraints are specified, which are both used in the specification of the constraint on the obligation policy serviceReset. The first constraint takes a parameter s, which is used in its specification. The second constraint workHours, is a time constraint, and is valid only between 8:00am and 4:00pm.

3.5.3 Constant Definitions

Constants can be defined in Ponder as shown in Figure 3.15. A type identifier can be used to indicate the user-defined type of a constant if it is not one of the predefined types (int, real, string, boolean). *User-defined* types are policy types. *External* types can be used to define constants which are not part of a policy specification. These are the IDL types of objects in the managed system. The *set* type defines a domain scope expression, which can be used to specify subject, target and grantee elements in basic policies. A set can be followed by the definition of the type of the objects in the set. This is usually the IDL type of subjects and targets.

```
constant definition =
     int                  {identifier = expression ;}                |
     real                 {identifier = expression ;}                |
     string               {identifier = expression ;}                |
     boolean              {identifier = expression ;}                |
     set [<type-name>]    {identifier = domain_scope_expression ;}   |
     user type-name       {identifier = expression ;}                |
     extern type-name     {identifier = expression ;}                ;
```

Figure 3.15 Syntax for constant definitions

The following are a few examples of defining constants.

```
int y = 5;
string x = managerX.getName();
string str1 = "this is a string";
set targetSet1 = /subnetA/routers;
set <EdgeRouter> targetSet2 = /subnetB; // All objects of type EdgeRouter from /subnetB
user myRoleType myRole1 = /branchA/roles/role1;
extern Router router1 = /routers/router1; // Give a name to a router object
```

Any of the constants described above, including the constraints and events described earlier, can be used to parameterise policy types.

3.5.4 External Specifications

External specifications are used to embed non-Ponder text into a Ponder specification. Unlike comments which are un-named and ignored by the policy compiler, external specifications are named and preserved by the compiler and runtime system. Such specifications can be accessed by external tools either at compile-time and/or run-time. External specifications are typically used to develop Ponder variants/extensions or attach non-Ponder definitions, code, scripts, performance and protocol requirements, structured documentation etc. to a Ponder specification. Their enforcement is implementation dependent. External specifications are enclosed within three angle brackets and preceded by the keyword *spec*, as shown in the following example.

```
inst auth+ net_config {
     subject  netOp/;
     action   setStrategy ;
     target   qEdgeRtr/ ;

     spec refs <<<
          related net_config2, net_config3;
          parent  config
```

```
            child   router_config
    >>> ; // end refs
}
```

In this example, an external specification named refs, associated with an authorisation policy specifies references to related obligation policies for which it is required as well as a parent policy from which it is refined and child policies which are derived from it. An analysis tool can extract the specification, parse it and interpret it accordingly.

3.6 Security Policy Examples

We present some additional examples to indicate the expressiveness of the language for security policies. Although our focus has been on non-discretionary access control, the language can be used to specify a variety of security policies. We revisit the issue of expressiveness in Chapter 4, after we have presented the composite features of the language which allow role-based policies to be specified.

Closed/Open Policies

A *closed policy* allows an access if there exists a positive authorisation for it, and denies it otherwise. Similarly, an *open policy* denies an access if there exists a negative authorisation for it and allows it otherwise. The simple case of classical closed/open policies can be simulated in our language by allowing only positive/negative authorisation policies to be specified. This can be implemented using simple authorisation policies to control the modality of the policies that can be added in the system. A closed policy can be implemented with the positive authorisation policy shown below which only allows positive authorisation policies to be specified. We implement an open policy with a single positive authorisation policy, which allows any action by any subject on any target (within a sub-tree of the domain hierarchy on which the open policy is to be enforced) to act as the default case. A negative policy can then be specified to prevent the specification of any additional positive authorisation policies involving those subjects/targets. Only negative authorisations will be allowed.

```
inst auth+ closedAuth1 {
    subject  s = /securityAdmins;
    target   as = /services/authorisationService;
    action   as.addAuth(a);
    when     a.getModality() = “+”;
}
```

The closedAuth1 policy permits security administrators to add authorisation policies in the system (by calling the addAuth method of the authorisation service) if the modality of the authorisation policy to be added is positive.

Authorisation for Classes of Objects

In general we advocate specifying policy in terms of instances rather than classes of objects, because we believe the idea of specifying authorisation in terms of classes or types of objects, is restrictive and infeasible in large-scale systems with millions of instances of a class. In practical applications policies are specified for sets of objects not related by class; objects are normally

grouped because of geographical distribution, for the convenience of human managers or for other application-specific reasons. However, it may be useful to define that the policy will only apply to instances of a particular class in a target domain. Knowing the class of the target object enables the interface specification to be located for checking that the policy actions correspond to methods in the interface. Consider the examples below:

```
type auth+ PrintAccessT (subject s, target t, string endTime) {
     action   print() ;
     when     Time.between("09:00:00", endTime);
}

inst auth+ staffPrintAccess = new PrintAccessT(/staff, /allPrinters, "22:00:00");
inst auth+ studentPrintAccess = new PrintAccessT(/students, /bwPrinters, "18:00:00");
```

In the above example, we define a positive authorisation policy type (PrintAccessT). This policy type is parameterised with the subject and target of the policy, and with a string indicating a time parameter to the time constraint of the policy. The string parameter is used to specify the endTime in the expression: Time.between("09:00:00", endTime) which constrains the validity time for the policy. The first instance (staffPrintAccess) creates a positive authorisation policy which authorises members of the staff domain to print to all printers (objects within the /allPrinters domain) between 09:00am and 10:00pm. The second instance (studentPrintAccess) creates a positive authorisation policy which authorises members of the students domain to print to black and white printers (objects within the /bwPrinters domain) between 9:00am and 6:00pm.

```
type auth+ PrintAccessForClassT (subject s, string endTime, type) {
     target   <type> /bwPrinters + /colorPrinters;
     action           print();
     when             Time.between("09:00:00", endTime);
}

inst auth+ staffPrintAccess = new PrintAccessT(/staff, "22:00:00", laserPrinters);
inst auth+ studentPrintAccess = new PrintAccessT(/students, "18:00:00", inkjPrinters);
```

The example shown above creates two positive authorisation instances with the similar semantics as in the previous example. However in this case, we use the ability of restricting the type of target objects to achieve the parameterisation of the target object. Target objects have different types (belong to different classes of objects) instead of being grouped into different domains. The authorisation type is not parameterised with the target of the policy. The target is the union of the printers in the bwPrinters domain and the colorPrinters domain. However, the class of target objects to which the policy applies is a parameter. Note the specification of the type (class) of objects following the target keyword. The first instance (staffPrintAccess) creates a positive authorisation policy which authorises members of the staff domain to print to laser printers between 09:00am and 10:00pm. The second instance (studentPrintAccess) creates a positive authorisation policy which authorises members of the students domain to print to inkjet printers between 9:00am and 6:00pm.

Ownership

The central idea of DAC is that the owner of an object, who is usually its creator, has discretionary authority over who else can access that object. The owner can grant and revoke access rights for other users to that object [Sandhu et al. 1994]. Although we consider the idea of ownership as problematic, we show how it can be simulated in Ponder. The main problem with using ownership as the basis for allowing access to objects, is the fact that users often do not have access to the data they create. In addition, in large-scale systems it is often impractical to allow the individual users which create objects to have full control of who has access to that object. Ownership complicates access control management in large systems and compromises security as users may abuse or misuse their power. In our language the access rights correspond to authorisation policies, which the owner can create in the system to assign permissions to the objects he/she owns. Constraints based on target object attributes (i.e. the creator/owner attribute) can be used to control access to

existing objects as shown in the example below. Delegation policies can also be used to allow the owner of a domain of objects to delegate access rights on these objects to other subjects.

```
inst auth+ ownerAuth1 {
     subject  s = /users/financeDept;
     target   f = /file/payroll;
     action   f.delete(), f.read();
     when     f.getOwner() = s;
}
```

In this case we assume a default negative authorisation policy, whereby everything is forbidden unless explicitly authorised. The ownerAuth1 policy authorises a user in the finance department to delete and read payroll files only if that user is the owner of the file.

Dynamic Separation of Duty

In dynamic separation of duty all members of a group are authorised to perform potentially conflicting actions but after performing one action they cannot perform a conflicting one. This can be implemented as constraints relating to attributes of the subject and target objects. Static separation of duty is a more restricted form of separation of duty in which certain sets of accesses cannot be allowed for the same subject. This involves constraints over groups of policies to restrict the specification of conflicting policies. We handle this with meta-policies which are presented in the next Chapter 4. We also revisit dynamic separation of duty in the context of role-based management in that next chapter.

```
inst auth+ sepDuty1 {
     subject  s = /accountants ;
     action   approvePayment ;
     target   t = /cheques ;
     when     s.id <> t.issuerID ;
}

inst auth+ sepDuty2 {
     subject  s = /accountants ;
     action   issue ;
     target   t = /cheques ;
     when     s.id <> t.approverID ;
}
```

The same user from the accountants domain cannot both issue and approve payment of the same cheque. This assumes that the identity of the issuer/approver can be stored as an attribute of the cheque object.

Derived Authorisations

Derivation rules are sometimes used to express dependencies among authorisations, and allow the derivation of new authorisations on the basis of existing ones and their validity [Jajodia et al. 1997; Bertino et al. 1998]. The use of derivation rules increases the complexity of access control enforcement, and complicates the specification of policies making it difficult to identify at a given time the access rights of a subject in the system. Although we do not introduce explicit derivation rules in Ponder, we can still specify such rules as part of the constraint of an authorisation policy:

```
inst auth+ A1 {
   subject /staff ;
   action  read ;
   target  /files ;
   when    Time.after("18:00:00") ;
}
```

```
inst auth+ A2 {
     subject /staff/technical ;
     action  write ;
     target  /files/technical ;
     when    Time.date() > "01/01/2002" and
             AuthSevice.exists(/staff,read,/files); }
```

The policy A2 specifies a dependency on A1: technical staff are authorised to write technical files if all staff have been granted the right to read any file and if the date is greater than 01/01/2002. In the constraint of A2 we assume access to an authorisation service with an interface method called *exists*, that can be used to check whether the given subject has a specific access right on a given target.

From the above two authorisation policies, we can derive the following authorisation:

```
inst auth+ derivedAuth {
     subject  /staff/techical ;
     action   write ;
     target   /files/techincal ;
     when     Time.after("18:00:00") and Time.date() > "01/01/2001";
}
```

Although in the above example the derived authorisation policy can be inferred with syntactic analysis of the policy specification, this will often not be the case. Access control will require the evaluation of all authorisation policies at runtime in order to derive new authorisations. In addition, the insertion of new policies or the deletion of policies may change the derived authorisations. In Ponder we avoid specifying dependencies between authorisation policies.

Backing Policies

Backing policies, introduced in [Rowley 1998], are usually needed in security sensitive situations where a subject requires the backing of a number of other principals in order to perform an action e.g. a chairman must have the backing of the majority of the board members in order to call an extraordinary meeting. In Ponder we can use authorisation and obligation policies to specify backing assuming that the backing condition can be specified and monitored by the underlying monitoring service, and then specified as an event to trigger obligation policies.

```
inst auth+ b1 {
     subject chairman;
     action  CallExtraMeeting();
     target  shareholders;
}
```

```
inst oblig b2 {
     on      ExtraMeeting;
     subject chairman;
     do      CallExtraMeeting();
     target  shareholders;
}
```

```
inst oblig b3 {
     on      (NoMembers/2+1)*votes(yes);
     subject trusted_agent;
     do      t.enable(policies/b1)->ExtraMeeting();
     target  t = /PolicyService;
}
```

For the chairman example, we need an authorisation policy (b1) authorising the chairman to call an extraordinary meeting and an obligation (b2) triggered by an event generated after a majority of yes vote events have been received. The authorisation policy is enabled only when a trusted agent enables it (in b3). The trusted agent obligation policy is triggered by the same backing event.

We acknowledge the fact that arbitrary backing policies probably require a separate scripting language to specify the backing condition.

Lattice-based Policies

In this subsection, we demonstrate how we can specify lattice-based policies, and in particular the Bell-LaPadula model within our framework. The Bell-LaPadula model assigns a security level to subjects and targets from a totally ordered lattice of security levels. Subjects can read a target

object only if their level dominates that of the target, and they can write an object only if the object's level dominates that of the subject. For a precise definition of the Bell-LaPadula model see Section 2.1.1. Here is an informal solution:

We start by considering the Bell-LaPadula model with just labels (not categories). In this case, a subject is permitted to write target objects whose label is greater than or equal to the label of the subject. On the other hand, a subject can read target objects whose label is less than or equal to that of the subject. Here is how a 3-label example would map to the domain structure. We divide the domains in two sub-trees, one for subjects and the other for target objects as shown in Figure 3.16. Each level in the tree corresponds to a label starting from the minimum label.

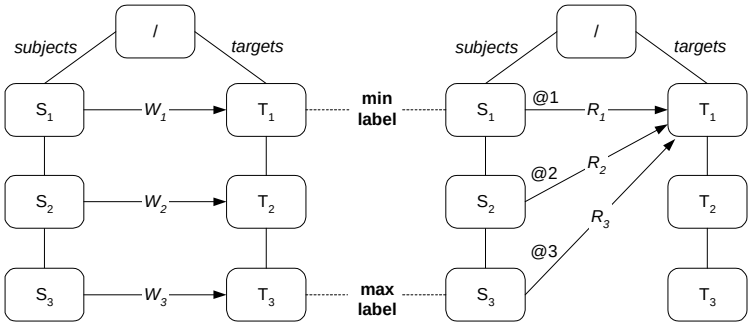


Figure 3.16 Mapping a label-only Bell-LaPadula policy to a domain structure

The arrows labelled W and R between the subject and the target domains show the write and read policies that can be specified using this domain structure. Here are policies W_1 and R_1. The rest are similar.

```
inst auth+ W1 {
     subject  /S1 ;
     action   write ;
     target   /T1 ;
}
```

```
inst auth+ R1 {
     subject  S1 ;
     action   read ;
     target   @1/T1 ;
}
```

The domain scope expressions provided by the grammar can be used to restrict the propagation of policies to sub-domains to the desired level as in R_1. Adding the categories to model the complete version of Bell-LaPadula just increases the number of domains that need to be created. The way to model this follows the same approach. Consider the above case of the three labels, with the addition of the set of categories: K = {k_1, k_2}, from which the following subsets are possible: {k_1}, {k_2}, {k_1,k_2}. We "divide" the domains at each level (of the previous solution) into separate domains for each subset of the set of categories as shown in Figure 3.17. Each domain is shown at the corresponding level as before and the text within it shows the subset of the set of categories to which it corresponds.

The arrows on the figure correspond to policies representing read, append and write access. Based on the Bell-LaPadula model, domain $S_{3.3}$ can have read access on the lightly shaded target domains as shown by the R_1 and R_2 policies, since their label (classification) is lower than or equal to that of

domain $S_{3.3}$, and their categories are proper subsets of domain $S_{3.3}$ categories. W_1 shows the write policy of $S_{3.3}$ and A_1 shows the append policy for $S_{1.2}$. All other policies, for write and append, can be realised in exactly the same way, stopping the propagation to sub-domains where necessary.

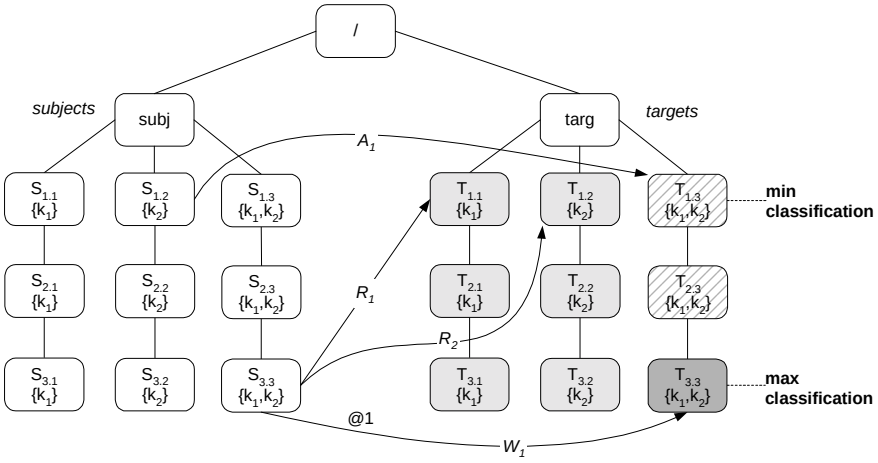


Figure 3.17 Mapping a Bell-LaPadula policy to a domain structure

It is possible to formalise the above solution. An important remark is that the above process of modelling the Bell-LaPadula security model using policies and domains, given a set of labels (classifications) and a set of categories, could be automated. However, although we can translate sensitivity levels into domain structures, any changes such as adding a category or a level, requires re-computing the domain structure.

Note: We see no value in specifying lattice-based policies using Ponder as this proves to be an inefficient and lengthy process. In addition, there is extensive research for approaches and techniques that can be used and are more suitable for mandatory policy implementation.

3.7 Conclusions

In this chapter we have presented the basic policy types of a language called Ponder, which was designed with the requirements described in Section 1.2 in mind. The basic policy types that are supported by the language are the following:

- *Authorisation policies* specify the list of actions that subjects are permitted or prohibited to perform on targets in the system. The language allows support for negative authorisation policies to explicitly forbid access. Positive authorisation policies can be extended with filters to transform the input values to action calls and their return values when these actions are allowed. All basic policies are defined over sets of objects formed by applying set operations, such as union, intersection and difference to the objects within domains. Domains can include domains (sub-domains) and set operations can be restricted to apply only to the top-level members of a domain, or applied recursively to any desired level,

including all nested levels of a domain. Constraints are used to restrict the applicability of policies.

- *Delegation policies* specify the permission to delegate access rights implied by authorisation policies to members of a grantee scope. The language supports constraints on delegation policies, negative delegation policies and cascaded delegation. Authorisation and delegation policies constitute the access control policies, which can be specified in the proposed policy language. They are designed to protect target objects and are conceptually enforced by each target.
- *Obligation policies* are event-triggered condition-action rules, which define the actions subjects (human or automated manager components) must perform, usually in relation to objects in a target domain.
- *Refrain policies* are a form of subject-based access control policy, which restricts the actions that subjects should execute, and are used where negative authorisations are not appropriate or cannot be specified.

All policies can be specified as parameterised *types* corresponding to classes in an object-oriented language. Policy *instances* can be created from the user-defined policy types and tailored to specific situations. Events, constraints and other constants can be defined and reused within policy specifications to enhance reusability. The language allows specifications external to the policy language to be embedded in a policy text.

We have presented simple examples of using the language in order to demonstrate its expressiveness, and evaluate its applicability in specifying certain policies. Examples can identify changes that need to be made in the language grammar, or policies for which the language is not suitable. More specifically, it became clear that the use of Ponder to specify lattice-based information flow policies as typified by the Bell-LaPadula model results in a complex specification (see Section 3.6). In general, the use of policy languages to specify lattice-based policies is inappropriate because lattice-based models assume a fixed set of policies to regulate the access of subjects to targets. However, we have presented examples of how Ponder policies can be used to specify other security policies available in the literature including the classical open (close) policies, dynamic separation of duty policies, backing policies where the execution of an action must be backed up by a number of subjects, and DAC policies where authorisations are based on the presence of other authorisations in the system. Future work will need to concentrate more on the evaluation of the language using large-scale scenarios from real-life applications.

Chapter 4

Composite Policy Features

4.1 Introduction

There is a need to group a set of related policy specifications within a syntactic scope with shared declarations in order to simplify the policy specification task for large distributed systems. This is a common concept in many programming environments and is the main motivation behind composite policy types in Ponder. Ponder composite policies facilitate policy management in large, complex enterprises. They provide the ability to group policies and compose them to reflect organisational structure, preserve the natural way system administrators operate or simply provide reusability of common definitions. This simplifies the task of policy administrators, and adds role-based management features to the language. At run-time, the set of policies defined in a composite policy, together with any constraints applying to the composite policy would be stored within a domain.

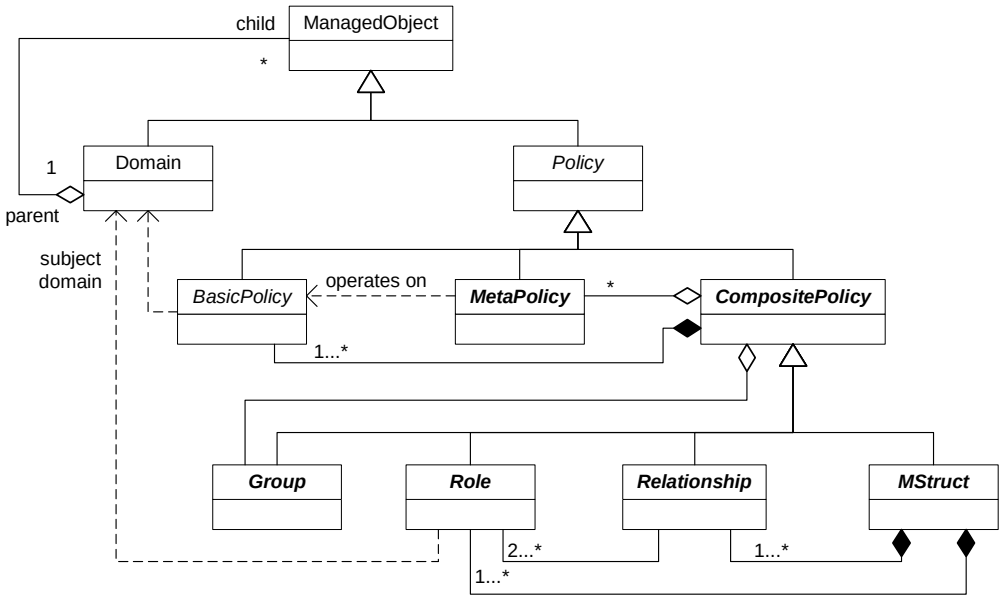


Figure 4.1 Composite policy object class hierarchy

In Figure 4.1 we extend the information model with composite policy classes. The figure includes only those classes which are necessary to illustrate the relation of the new classes to those described in the previous chapter. Four types of composite policies will be presented in detail in

this chapter: groups, roles, relationships and management structures. A composite policy includes one or more basic policies, nested groups and possibly meta-policies to specify application specific constraints on the policies specified within the scope of the composite policy. A role is always associated with a domain that specifies the common subject for all the policies inside the role. Relationships are specified for two or more roles, and management structures (*MStruct*) group related role-relationship configurations to model organisational units.

4.2 Groups

A group is a packaging construct to group related policies together for the purposes of policy organisation and reusability and is a common concept in most programming languages. There are many different potential criteria for grouping policies together – policies may reference the same targets, relate to the same department or apply to the same application. Figure 4.2 shows the syntax for a group instance, and a group type. A group can contain zero or more basic policies, nested groups and/or meta-policies in any order. A meta-policy specifies constraints on the policies within the scope of the group, and will be discussed later in Section 4.5.

```
inst group groupName {
      { common-element-specification}
      { basic-policy-definition }
      { group-definition }
      { meta-policy-definition }
}
```

```
type group groupName(formal-parameters) {
      { common-element-specification}
      { basic-policy-definition }
      { group-definition }
      { meta-policy-definition }
}
```

Figure 4.2 Group construct syntax

Reusability can be achieved by specifying groups as types, parameterised with any policy element or system attribute, and then instantiating them multiple times. As an example, policies related to the login process can be grouped together since they would always be instantiated together.

```
inst group loginGroup {

    inst auth+ staffLoginAuth {
        subject /dept/users/staff ;
        target  /dept/computers/research;
        action  login;
    }

    inst oblig loginActions {
        subject s = /dept/computers/loginAgent ;
        on      loginevent (userid, computerid) ;
        target  t = computerid ^ {/dept/computers/} ;
        do      s.log (userid, computerid) -> t.loadEnvironment (userid) ;
    }

    inst oblig loginFailure {
        on             3*loginfail(userid) ;
        subject        s = /NRegion/SecAdmin ;
        target <userT> t = /NRegion/users->select(t1 | t1.getId() = userid) ;
        do             t.disable() -> s.log(userid) ;
    }

}
```

The login group policies authorise staff to access computers in the research domain, log login attempts, load the users environment on the computer and deal with login failures.

4.3 Roles

Roles provide a semantic grouping of policies with a common subject, generally pertaining to a position within an organisation such as department manager, project manager, analyst or ward-nurse. Specifying organisational policies for human managers in terms of manager positions rather than persons permits the assignment of a new person to the manager position without re-specifying the policies referring to the duties and access rights of that position (see Section 2.5.3). A role can also specify the policies that apply to an automated component acting as a subject in the system, or to a network device such as a router.

We represent organisational positions with domains, which we call subject domains, and associate them with roles. A role is thus the set of authorisation, obligation, refrain and delegation policies with the subject domain of the role as their subject.

```
inst role roleName {                          type role roleName(formal-parameters) {
    { common-element-specification}                 { common-element-specification}
    { basic-policy-definition }                     { basic-policy-definition }
    { group-definition }                            { group-definition }
    { meta-policy-definition }                      { meta-policy-definition }
} [ @ subject-domain ]                        }
```

Figure 4.3 Role construct syntax

Roles (Figure 4.3) can include any number of basic-policies, groups or meta-policies. The subject domain of the role can be optionally specified following the '@' sign for role instances, and must be pre-created. If it is not specified then a subject domain with the same name as the role is created by default when the role instance is created.

```
type role /mgmtInfo/roles/ServiceEngineer (CallsDB callsDb) {
    inst oblig serviceComplaint {
        on       customerComplaint(mobileNo) ;
        do       t.checkSubscriberInfo(mobileNo, userid) ->
                 t.checkPhoneCallList(mobileNo) -> investigate_complaint(userId);
        target   t = callsDb ;  // calls register
    }

    inst oblig deactivateAccount { . . . }
    inst auth+ serviceActionsAuth {  . . .  }
    // other policies
}

inst role /mgmtInfo/roles/Area1ServiceEng =
                        mgmtInfo/roles/ServiceEngineer(Area1CallsDB)@/SD/area1ServiceEng;
```

The role type ServiceEngineer models a role in a mobile telecommunications service, which is responsible for responding to customer complaints and service requests. The role type is parameterised with the calls database, a database of subscribers in the system and their calls. The obligation policy serviceComplaint is triggered by a customerComplaint event with the mobile number of the customer given as an event attribute. On this event, the subject of the role must execute a sequence of actions on the calls-database in order check the information of the subscriber whose mobile-number was passed in through the complaint event, check the phone list and then investigate the complaint. Note that the obligation policy does not specify a subject as all policies within the role have the same implicit subject. The role is instantiated with the calls-database for area 1 to create the role instance Area1ServiceEng. The domain /mgmtInfo/roles is

used to store both the role type and the role instance definitions. The subject domain of the role instance is /SD/area1ServiceEng. Users assigned to the role must be included in this domain (see Section 7.6.2 for a description of user-role assignments).

4.3.1 Type Specialisation

Ponder allows specialisation of policy types through the mechanism of inheritance. When a type extends another, it inherits all of its policies, may add new policies and overrides policies with the same name. Inheritance is only defined for composite policy types. We present it here in terms of the role construct but it can also be used for groups, as well as relationships and management structures which will be presented in Section 4.4.

```
type role roleTypeName (formalParameters) extends parentRoleType [(actualparameters)]
                                                  [{,parentRoleType [(actualparameters)]}] {
       role-body
}
```

Figure 4.4 Inheritance syntax

The type that extends some other base type, can pass parameters to the base type with the *extends-clause* in order to parameterise the base type. The language does not currently support polymorphism or dynamic binding. In this thesis we define multiple inheritance only for role policy types. The problem with multiple inheritance is that of multiple policies with the same name coming from different base-types [Lupu 1998]. This can be solved in two ways:

- The policy compiler can warn the policy writer of this situation, so that the policy writer can choose not to inherit one of the two base types or change the names of the policies in the base-types if possible.
- By prefixing the names of the policies with the name of the type from which they are inherited. This is a common way of resolving similar name-conflicts from multiple inheritance in object-oriented languages.

We show an example of the use of inheritance to extend a role type below:

```
domain /mgmtInfo/roles;

type role MSServEngineer (CallsDB vlr, SqlDB eqRegistry) extends ServiceEngineer(cdb)  {
    inst oblig maintainProblems {
         on      MSfailure(equipmentId) ;     // MS = Mobile Station
         do      updateRecord(equipmentId) ;
         target  eqRegistry                   // Equipment identity registry
    }
}
```

The MSServEngineer (MobileStation Service Engineer) role extends the ServiceEngineer role specified in the previous example. It inherits the policies of the parent role and adds an obligation policy that updates the equipment's record in the equipment identity registry (the target) when the mobile station signals a failure (the event).

Role Hierarchies

Role and organisational hierarchies can be specified using specialisation. The role-hierarchy in Figure 4.5 can be specified in Ponder by extending roles as shown in the following example.

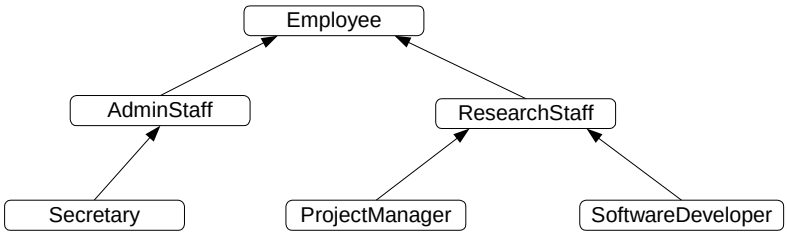


Figure 4.5 A role hierarchy

```
type role EmployeeT(…) { … }
type role AdminStaffT(…)       extends Employee { … }
type role ResearchStaffT(…)    extends Employee { … }
type role SecretaryT(…)        extends AdminStaff { … }
type role SoftDeveloperT(…)    extends ResearchStaff { … }
type role ProjectManagerT(…)   extends ResearchStaff { … }
```

4.4 Role Relationships and Management Structures

Managers acting in organisational positions (roles) interact with each other. A relationship groups the policies defining the rights and duties of roles towards each other. It can also include policies related to resources that are shared by the roles. It thus provides an abstraction for defining policies that are not part of the role specifications, but are part of the interaction between the roles.

```
inst rel relationshipName {
      { common-element-specification }
      { basic-policy-definition }
      { group-definition }
      { role-definition }
      { meta-policy-definition }
}
```

```
type rel relationshipName(formal-parameters) {
      { common-element-specification}
      { basic-policy-definition }
      { group-definition }
      { role-definition }
      { meta-policy-definition }
}
```

Figure 4.6 Relationship construct syntax

The syntax of a relationship (Figure 4.6) is very similar to that of a role. In addition, a relationship includes the definitions of the roles participating in the relationship. Alternatively, these roles can also be passed as parameters to relationship types. An example of this is shown below:

```
type rel ReportingT (ProjectManagerT pm, SecretaryT secr) {
    inst oblig reportWeekly {
        on       Timer.day (“monday”) ;
        subject  secr ;
        target   pm ;
        do       mailReport() ;
    }
    // other policies or roles participating in the relationship
}
```

The ReportingT relationship type is specified between a ProjectManager role type and a Secretary role type. The obligation policy reportWeekly specifies that the subject of the SecretaryT role must mail a report to the subject of the ProjectManagerT role every Monday. The use of roles in place of subjects and targets implicitly refers to the subject of the corresponding role.

Relationships were introduced in [Lupu 1998] and a syntax for interaction protocol specification was also proposed to define the interactions between the managers, assigned to the roles of a relationship, in terms of the permitted sequences of messages that the managers can exchange. This is a very important part of a relationship specification which we do not cover in this thesis. Future

work will need to revisit this issue and extend the grammar with interaction protocols, which should be part of the relationship syntax.

4.4.1 Management Structures

Many large organisations are structured into units such as branch offices, departments, and hospital wards, which have a similar configuration of roles and policies. Ponder supports the notion of management structures to define a configuration in terms of instances of roles, relationships and nested management structures relating to organisational units. For example a management structure type would be used to define a branch in a bank or a department in a university and then instantiated for particular branches or departments.

```
inst mstruct manStructName {
      { common-element-specification }
      { basic-policy-definition }
      { group-definition }
      { role-definition }
      { relationship-definition }
      { meta-policy-definition }
}
```

```
type mstruct manStructName(formal-parameters) {
      { common-element-specification}
      { basic-policy-definition }
      { group-definition }
      { role-definition }
      { relationship-definition }
      { meta-policy-definition }
}
```

Figure 4.7 Management structure syntax

A management structure is a composite policy containing definitions of roles, relationships and other nested management structures. Basic policies, which refer to the roles and relationships defined in the management structure, as well as constraints in the form of meta-policies can also be included (Figure 4.8). We consider a management structure as being different to the concept of a community as defined in the RM-ODP Enterprise Viewpoint [ISO/IEC 1999]. A community in RM-ODP terminology is a semantic concept defined as “a configuration of objects formed to meet an objective”, where the objective is expressed as a “contract which specifies how the objective can be met”. Management structures are structuring mechanisms used to compose policy specifications based on organisational or network structure, in order to cope with large-scale policy specifications. We thus refrain from specifying relationships or interaction-protocols between management structures, and we only allow them to be composed using syntactical inclusion.

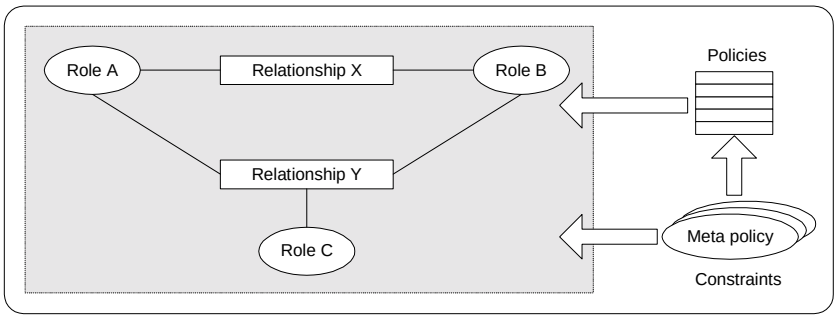


Figure 4.8 Management structure components

Next, we present an example to demonstrate the use of roles, relationships and their configurations in management structures.

4.4.2 Example: Security Quality Assurance in SLA Management

A *service level agreement* (SLA) implies certain guarantees for the customer of that SLA. The content of these guarantees can roughly be broken down into the following areas:

- Stability: availability, guarantee of failsafe operations, meantime between failure
- Performance: response times, guaranteed capacity
- Security: protection against any form of system attacks
- Support: hotline availability times, etc.

In this section we borrow the ideas of a case study presented in [Hegering et al. 1999] which involves the security area of quality assurance in SLA management for networked environments. We deal with a telecommunications network that is constructed from interconnected networks of digital switching nodes (SNs). The network is divided into regional networks, which consist of a number of local network sites each of which is a digital switching network.

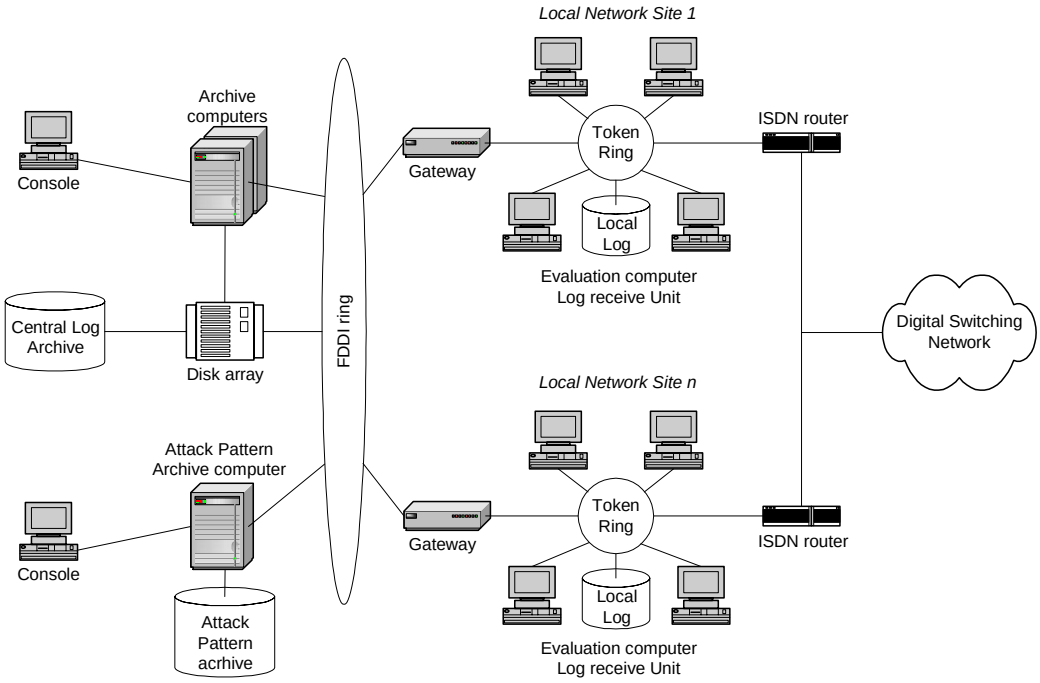


Figure 4.9 Security quality assurance system architecture

Figure 4.9 illustrates the operation of the security quality monitoring and assurance system for one of the regions of the telecommunications network. The goal is to monitor all operations executed on the switching nodes of the digital switching network, in order to be able to detect and trace any security attacks. The switching nodes of the digital switching network generate specialised log data, which is stored in log-files. Experts, who are familiar with the structure of the log data

generated from the various SNs, use an editor tool to create a uniform structured log-file from the raw log-file generated for each SN initially. Log-data collection is implemented by several log receive units, which collect the data via ISDN routers. The log receive units that are part of the same site are interconnected through token rings. Local log databases are used to store the data at each of the site networks. The data is then transferred over an FDDI ring and archived centrally. An evaluation manager for the region is responsible for specifying the so-called attack patterns used to analyse the archived log data, and stored centrally. The attack patterns are compared to the data logs in order to identify any security attacks. The function of analysing the log data takes place at each of the site networks as well as at the regional network level.

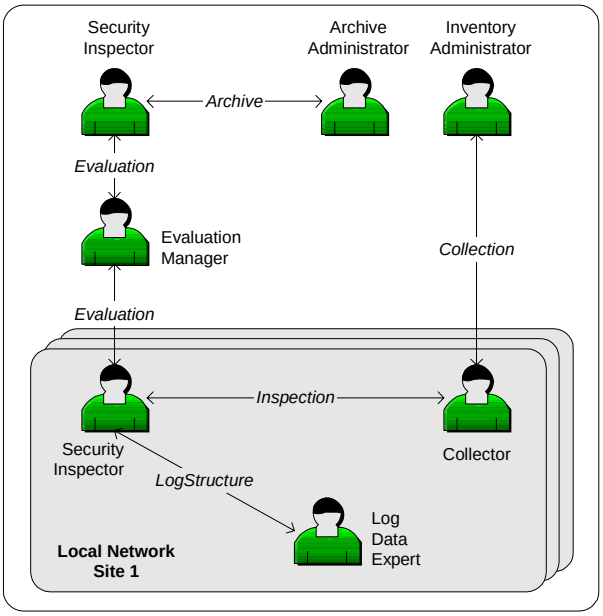


Figure 4.10 Roles, relationships and management structures for a single TN region

Access to the various archiving systems, log databases, and tools is restricted only to authorised users assigned to the appropriate roles. Figure 4.10 illustrates the various roles involved in the operation of the system, and identifies the relationships that exist between them to enable coordination and cooperation.

- **Security Inspector**: authorised to access and use the log evaluator tool to check the log data. A security inspector role is defined for each of the local network sites, as well as for the regional domain network. Security inspectors interact with the evaluation managers to notify them of detected security attacks.
- **Evaluation Manager**: responsible for specifying the attack patterns using the attack pattern editor. A second function of this role is to plan the measures to take when attacks are detected. The evaluation manager interacts with the security inspectors to receive information about detected attacks.

- **Archive Administrator**: responsible for the archiving of the data in the central log repository. It may interact with the regional security inspector, in case of problems or anomalies with the archiving system.
- **Inventory Administrator**: responsible for the provision of log data from the various domain networks.
- **Collector**: this role exists for each of the local network sites. Occupants of this role are authorised to access the log receive units to collect data received from the SNs. It interacts with the security inspector at the local network site level, through an inspection relationship involving the collection of data.
- **Log Data Expert**: each local network site has more than one as required. The data expert is authorised to use the structure editor tool to create a structured format of the raw log data received from the various SNs.

Figure 4.10 demonstrates the formation of management structures out of the roles and relationships described above. A management structure type is defined to model the configuration of the roles and relationships in a site network and then instantiated for each of the sites with the appropriate parameters. A second management structure is used to model the configuration of roles and relationships for a regional network. This outer management structure includes the instances of the first management structure for each of the site networks on the region. In the following we outline the policy specification for the management structures assuming that there are three site networks for a regional network. We do not present the individual policies that are part of the roles and relationships. Roles contain the appropriate authorisation policies to enable the occupant of the role to access the parts of the network and the tools required. Obligations that are not part of relationships (i.e. which are not obligations of a role occupant towards another) are also specified within the roles as needed. Relationships include those obligations of a role towards another (e.g. reporting functions on certain events like time-periods), as well as any authorisations of one role towards another (e.g. what methods the occupant of a role can execute on the occupant of another role in order to request functions or information).

```
// The management structure which models the security quality assurance system part
// for a single site network. The structure is parameterised with the domain
// representing the site, and a reference to the log database of the network

type mstruct siteStructureT(domain site, LocalLogDB logDB) {
      // define role types
      type role secutiryInspectorT(domain site, LocalLogDB logDB) { … }
      type role collectorT(domain site, LocalLogDB logDB) { … }
      type role logDataExpertT(domain site, LocalLogDB logDB, SwitchNetworkType snType) {…}

      // define relationships types
      type rel inspectionT(role secInsp, role col) { … }
      type rel logStructureT(role secInsp, role logExprt) { … }

      // create instances for the roles
      inst role securityInspector = securityInspectorT(site, logDB);
      inst role collector = collectorT(site, logDB);
      inst role logDataExpert1 = logDataExpertT(site, logDB, SiemensEWSD);
```

```
      inst role logDataExpert2 = logDataExpertT(site, logDB, AlcatelS12);

      // create instances for the relationships
      inst rel inspection = inspectionT(securityInspector, collector);
      inst rel logStructure1 = logStructureT(securityInspector, logDataExpert1);
      inst rel logStructure2 = logStructureT(securityInspector, logDataExpert2);
}
```

Each of the roles in the siteStructureT management structure type, is parameterised with the site domain and the reference to the log database used for the local data logs. These parameters are passed to the management structure instantiation and are then used in the instantiation of the roles inside the management structure. They then act as the target for policies inside the roles. Note that the logDataExpertT role type also requires the type of the switch nodes which will be monitored. Some of the obligation policy actions inside the role may depend on the type of node from which the logs are received. Since there are two log data expert roles, two logStructure relationships are instantiated between the security inspector and the log experts.

```
// The management structure which models the security quality assurance system part
// for a single region of the telecommunications network. The structure is
// parameterised with the domain representing the region

type mstruct siteQualityAssuranceT(domain region) {
      // paths to the three site domain networks of this site
      domain site1 = region.getDomain("/site1");
      domain site2 = region.getDomain("/site2");
      domain site3 = region.getDomain("/site3");

      // define role types
      type role archiveAdminT(DB centralLog) { …     }
      type role inventoryAdminT(DB centralLog, domain net1, domain net2, domain net3) { … }
      type role evaluationManagerT(DB attackPatternDB) { … }

      // define relationships types
      type rel evaluationT(role secInsp, role evalMan) { … }
      type rel archiveT(role secInsp, role archAdmin) { … }
      type rel collectionT(role invAdmin, role col) { … }

      // create instances for the roles
      inst role siteSecInspector = domainStructureT.securityInspectorT(site.get("logDB"));
      inst role archiveAdmin = archiveAdminT(site.get("centralLog"));
      inst role evalManager = evaluationManagerT(site.get("patternLog"));
      inst role inventoryAdmin = inventoryAdminT(site.get("centralLog"),
                                                 region1, region2, region3);

      // create instances for the relationships which exist between roles
      // at the level of the site network
      inst rel siteEvaluation = evaluationT(siteSecInspector, evalManager);
      inst rel archive = archiveT(siteSecInspector, archiveAdmin);

      // createthe instances for the site management structures
      // Each of the management structures is parameterised with the site part
      // relative to the region parameter which is a formal parameter of the current
      // management structure
      inst mstruct site1Struct = domainStructureT(site1, site1.get("logDB"));
      inst mstruct site2Struct = domainStructureT(site2, site2.get("logDB"));
      inst mstruct site3Struct = domainStructureT(site3, site3.get("logDB"));

      // create the relationships between the roles which involve those of
      // the inner management structures
      inst rel evaluation1 = evaluationT(site1Struct.securityInspector, evalManager);
      inst rel evaluation2 = evaluationT(site2Struct.securityInspector, evalManager);
      inst rel evaluation3 = evaluationT(site3Struct.securityInspector, evalManager);

      inst rel collection1 = collectionT(inventoryAdmin, site1Struct.collector);
      inst rel collection2 = collectionT(inventoryAdmin, site2Struct.collector);
      inst rel collection3 = collectionT(inventoryAdmin, site3Struct.collector);
}
```

Notice that the regional security inspector for the region management structure is instantiated from the role definition of the security inspector (securityInspectorT) defined in siteStructureT.

4.5 Meta Policies

The validity of a policy may depend on other policies existing or running in the system within the same scope or context. These conditions are usually impossible or impractical to specify as part of each policy and therefore need to be specified as part of a group of policies. Meta-policies specify constraints over a set of policies, on the permitted types of policies or their policy elements. These constraints apply to policies within a specific scope, and limit the permitted policies in the system, or disallow the simultaneous execution of conflicting policies. Meta-policies can be defined within a composite-policy to apply to all policies within the scope of the composite policy. Alternatively they may apply to all policies within a domain sub-tree. The syntax of a meta-policy is based on the syntax of the object constraint language (OCL). The body of a meta-policy specifies the constraint as a series of OCL expressions separated by semicolons. The expressions can be boolean or navigation expressions. If any of the boolean expressions evaluates to `true,` execution stops, and the action following the *raises*-clause is executed. This way, a series of related constraints can be specified within the same meta-policy. Note that the result of an OCL expression can be named so that it can be passed to the exception action as a parameter (see examples), or reused in subsequent constraint expressions.

A meta-policy can also be used to specify concurrency constraints on the mandatory sequencing, and permitted parallelism of activities, or to forbid the overlap of certain activities. When used for concurrency constraint specification, a meta-policy does not specify the raises-clause, and its body consists of a series of concurrency constraint expressions. A concurrency constraint expression specifies a sequence of activities separated with concurrency constraints. The concurrency constraints and their semantics are the same as those used in obligation policy actions (see Section 3.4.1). An activity is either:

- An action in an obligation policy
- An obligation policy

In the latter case, this implies that all the actions of the obligation policy are subject to the concurrency constraint specified (See examples that follow).

```
inst meta metaPolName raises exception [ ( parameters ) ] {
        { OCL-expression | [identifier] = OCL-expression } |
        { concurrency-expression }
}
concurrency-expression = activity (-> | | | || | &&) concurrency-expression
activity = [path .] identifier {. identifier}
```

Figure 4.11 Meta-policy syntax

The examples in the following subsection indicate how meta-policies can be used to specify application dependent constraints on groups of policies.

4.5.1 Constraint Policy Examples

We have presented examples in Chapter 3 to show how Ponder can be used to represent many of the security policies available in the literature. We now show some additional examples of policies that can be expressed in our language related to specifying constraints on sets of policies specified in composite structures, to complement those presented in Section 3.6.

Constraints on Elements of Policies

The following example shows a meta-policy used within a role to specify a simple constraint on the policies within the role; in this case, a constraint on the instantiation of the role: the number of patients for which the nurse is responsible must be less than 10.

```
type role nurseT(set <patient> p) {
      inst auth+ mealSchedule {
            target  p;
            action  updateMealSchedule;
      }

      inst oblig administer {
            target  p;
            on      Time.at("08:00:00");
            do      administerDrugs() -> checkTemperature();
      }

      inst meta maxNoOfPatients raises errorInPatients(p) {
            p->size < 10;
      }
}
```

Static Separation of Duty

The following examples show how to specify a static separation of duties to prevent the same person from being authorised to perform actions marked as conflicting.

```
inst meta budgetDutyConflict raises conflictInBudget(z) {
       [z] = this.policies -> select (pa, pb |
              pa.subject -> intersection (pb.subject)->notEmpty    and
              pa.action -> exists (act | act.name = "submit")      and
              pb.action -> exists (act | act.name = "approve")     and
              pb.target -> intersection (pa.target)->oclIsKindOf (budget))
       z -> notEmpty ;
}
```

This metapolicy prevents a conflict of duty in which the same person both approves and submits a budget. It searches for policies with the same subject acting on a target budget in which there is an action submit and approve.

We specify a generic meta-policy type, shown below, that can be used to create various instances of the static separation of duty principle involving two actions.

```
type meta dutyConflictT(act1, act2, tarType) raises conflictSepD(z) {
      [z] = this.policies->select(pa, pb |
               pa.subject->intersection(pb.subject)->notEmpty      and
               pa.action->exists(act | act.name = act1)            and
               pb.action->exists(act | act.name = act2)            and
               pb.target->intersection(pa.target)-> oclIsKindOf(tarType)) ;

      z -> notEmpty;
}

inst  meta dc = dutyConflictT("execute", "authorise", "payment");
      meta bwDc = dutyConflictT("addBandwidth", "use", "service");
```

```
inst oblig notifyConflict {
      subject policyService;
      on      conflictSepD(z);
      do      policyService.notify(manager);
}
```

Two actions and a target type are passed as parameters to the meta-policy. Within its body, the meta-policy checks all pairs of policies in its scope, for possible conflicts. If there exists a pair of policies with common subjects, who have actions act1 and act2 respectively in their action entry, and whose target intersection is of the given tarType, then there is a conflict and the conflict action conflictSepD(z) is called. This action takes the set of pairs of policies resulting in conflict (the result of the OCL expression) as a parameter, so that it can act on them. In order to check the type of the target intersection we use the oclIsKindOf method defined in OCL.

The following example is another instance of a static separation of duty, which involves the assignment of users to roles.

```
type meta roleAssignment(role r1, role r2) raises incompatibleRoles(r1, r2, users) {
       [users] = r1.subjectDomain -> intersection (r2.subjectDomain)
       users->notEmpty
}

inst meta accountingRoles = roleAssignment(/roles/Accountant, /roles/FinanceDirector);
```

The roles Accountant and FinanceDirector in a specific organisation are marked as conflicting, so no user can be assigned to both. The meta policy type checks that the subject domains of the two roles have no common elements.

Self-Management

"*There should be no policy authorising a manager to retract policies for which he is the subject"*, from [Lupu 1998]. This happens within a single authorisation policy with overlapping subjects and targets. A meta-policy can be used to specify this as follows:

```
inst meta selfManagement1 raises selfMngmntConflict(pol) {
  [pol] = this.authorisations -> select (p | p.action->exists (a |
              a.name = "retract" and a.parameter -> exists (p1 |
                     p1.oclType.name = "policy" and p1.subject = p.subject))) ;
  pol->notEmpty ;
}
```

The body of the policy contains two OCL expressions. The first one operates on the set of authorisations in the meta policy container (a composite policy), referred to by "this". It selects all policies (p) with the following characteristics: the action set of p contains an action named "retract", and whose parameters include a policy object with the same subject as the subject of policy p. The second OCL expression is a boolean expression; it returns true if the pol variable, which is returned from the first OCL expression is not empty. If the result of this last expression is true, the exception specified in the raises-clause executes. It receives the pol set with the conflicting policies as a parameter

Prerequisite Roles

Prerequisite roles is a type of constraint identified in [Sandhu et al. 1996] whereby a user can be assigned to role A only if the user is already assigned to role B. This can be translated into: user U can only become a member of the subject domain of role A if it is already a member of the subject domain of role B. Here is how it can be specified as a meta policy:

```
type meta prerequisiteRoles(role r1, role r2) raises assignmenError(r1, r2, users) {
       [users] = r1.subjectDomain - (r2.subjectDomain);
       users->notEmpty
}

inst meta Roles = roleAssignment(/roles/HeadOfDepartment, /roles/Professor);
```

The roles Professor and HeadOfDepartment in a university setting are prerequisites. A user cannot be assigned to the head of the department role unless that user is already assigned to the professor role. The meta policy type defined checks that the subject domain of the HeadOfDepartment role contains any elements (users) not in the subject domain of the

Professor role. If that set is not empty then the resulting set of users have been erroneously assigned to the headOfDepartment role. We assume an attribute called "subjectDomain" which can be accessed on each role object to get the subject domain of the role.

User-Role Assignment Cardinality Constraints

This kind of constraint restricts the number of users assigned to a role [Sandhu et al. 1996]. It can be specified inside a role if we know at specification time, the maximum number of users that can be assigned for that role. Otherwise it can be specified outside the role.

```
type meta maxUsersT(role r, int max) raises maxUserLimit(r) {
       [users] = r.subjectDomain;
       users->size > max;
}

inst meta maxUsers1 = maxUsersT(/roles/SecurityAuditor, 1);
```

The example above shows a single meta-policy instance created from a generic meta-policy type, which handles the user-role assignment cardinality constraints. The example specifies that only one user can be assigned to the SecurityAuditor role.

Concurrency Constraints

This following example, demonstrates the use of a meta-policy to specify concurrency constraints which involve a set of policies. The *paymentConcurrency* meta-policy in the example, specifies two concurrency constraints which involve individual actions between different policies.

```
inst role accountant {

      inst oblig paymentPol {
            on      paymentRequest(p);
            target  t = Payments_registry;
            do      t.registerPayment(p) || t.updateRecords(p);
      }

      inst oblig chequeIssuePol {
            on      paymentTransactionInit(t);
            target  db = backupDB;
            do      issueCheque() || db.backupRecords(t);
      }

      inst meta paymentConcurrency {
            // must register payment before issuing the cheque
            paymentPol.registerPayment -> chequeIssuePol.issueCheque;

            // cannot update and backup records at the same time
            (paymentPol.updateRecords -> chequeIssuePol.backupRecords) |
            (chequeIssuePol.backupRecords -> paymentPol.updateRecords)
      }
}
```

History-based Access Control

History-based access control policies [Acharya et al. 1998] can be expressed in our language assuming the existence of an event-history monitoring system, which will be used by the policies to access information about events that happened in the past. Examples of such policies include the following:

Joint action based authorisation policies: "*Three out of five users which possess a certain role must vote in-favour, for a subject to be permitted to execute an action*". [Varadharajan et al. 1996].

This is a form of backing policy which was presented in Section 3.6, which can be specified using a combination of authorisation and obligation policies. In Ponder this type of policy can also be specified as a constraint on the authorisation policy because it involves a single access right:

```
inst role doctor {
      inst auth+ admitPatient {
            target  t = /servers/patientRecords;
            action  admitPatient(x);
            when    votingServer.getAdmit("yes", x) > doctor.subjectDomain->size() / 2;
      }
      …
}
```

The above example allows an occupant of the doctor role to admit a patient to the hospital if more than half of the other doctors (i.e. the other users which possess the doctor role) also agree.

Limiting resource usage: "*At most 5 disk partitions can be used for back-up activities*". [Lupu 1998]. Again, this policy does not restrict the specification of policies in the system, but rather the execution of actions based on object attribute values.

```
type role backupAdminT(BackupControllerT backupController) {
     inst auth+ backupRestrict {
              action    backup ;
              target    t = /server/backup ;
              when      backupController.partitionsUsed(t) < 5 ;
     }
     …
}
```

A backup administrator is permitted to perform a backup operation on a certain backup server, only if the number of partitions used on that server is less than 5. The backupController server object provides information about the backing process and is passed as a parameter to the role.

Based on previous actions: "*A program can open local files for reading only if it has not opened a socket*". This is similar to a dynamic separation of duty policy, and is thus specified as a constraint on authorisation policies. Examples of this have been presented in Section 3.6.

Closed/Open Policies as Meta-Policies

One way of modelling the classical closed and open policies was described in Section 3.6. However, meta-policies can also be used to specify a closed or open policy similar to the way integrity rules are used in ASL [Jajodia et al. 1997]. The following example demonstrates how a simple meta-policy can be used to customise the access control decision mechanism based on the existence of positive or negative authorisation policies.

```
type meta closedPolicyType(domain path, s, t, a) raises accept() {
  path -> exists (p | p.type == "auth+" and
                  p.subject->exists(s) and p.target->exists(t) and p.action->exists(a)) ;
}

type meta openPolicyType(domain path, s, t, a) raises accept() {
  path -> exists (p | p.type == "auth-"  and
        p.subject->exists(s) and p.target->exists(t) and p.action->exists(a)) != true ;
}
```

The first meta-policy *closedPolicyType* raises an *accept* exception if the set of authorisations under the given path contains at least one positive authorisation policy which allows the execution of action a from subject s on target t. The *openPolicyType* raises an *accept* exception if the set of authorisation policies under the given path do not contain a negative authorisation policy which disallows the execution of action a from subject s on target t. Note that the path can be the root of the policy hierarchy to apply the closed (open) policy on all domains in the system. The two meta-policies

can then be distributed to the relevant access control agents which will instantiate them and interpret them every time an access decision is requested.

4.6 Additional Language Features

The Ponder framework is self-managed in that policies and other constructs such as roles and relationships are implemented as objects stored within domains. Authorisation policies can therefore be used to specify who is permitted to add, delete or edit policies, as has been demonstrated in some of the examples presented. Furthermore, obligation policies can be used to specify what actions must be performed on policy objects when certain events occur. For example, obligation policies can be specified to enable new policies or disable existing ones in order to adapt to new circumstances such as failures, emergency conditions, etc. In addition, human managers or automated components that are subjects for a set of policies may in turn, be managed by other managers based on a different set of policies, and thus become targets. We thus avoid the use of *administrative roles* as proposed in the RBAC models [Sandhu et al. 1996] to manage a set of roles. Roles and other composite policies are themselves objects and can be managed by the policies specified in other roles.

The class hierarchy of the language (see Appendix A) allows new policy types that may be identified in the future to be defined as sub-classes of existing policy types. This includes both basic policies (Figure 3.1) and composite policies (Figure 4.1) and makes it easier to *extend* the language, a design goal identified in Section 1.2. In addition, the model provides a convenient means of translating policies to structured representation languages such as XML. The XML representation can then be used for viewing policy information with standard browsers or as a means of exchanging policies between different managers or administrative domains. The DMTF have already engaged in the specification of a mapping of CIM to XML [DMTF 1999b].

Import statements can be used to import definitions such as constants, constraints and events, from external Ponder specifications stored in domains, into the current specification. This allows reuse of common specifications and minimises errors that arise due to multiple definitions. The following example shows how an event specification can be reused.

```
inst group /groups/groupA  {
      event e(userId) = 3*loginfail(userid) ;
      // other common specifications & basic-policies  }

inst group groupB {
      import /groups/groupA ;

      inst oblig FlexibleLoginFailure {
            on      e(userId)|loginTimeOut(userId);
            subject s = /NRegion/SecAdmin ;
            target  t = /NRegion/users ^ {userid} ;
            do      s.log(userid);
      }
}
```

GroupB imports the specification groupA from the /groups domain (where it is stored), and reuses the specification of the event e(userId) defined within loginFailure. The event of the new obligation policy is now 3 consecutive loginfail events or a loginTimeOut event, which is triggered when the user takes too long to enter the password after the prompt.

4.6.1 Example Composite Policy Specification

The example below demonstrates the structure of a policy specification. Import and domain statements can be placed anywhere within the specification. We deliberately show how this small example can be specified in different files, to demonstrate the use of import statements. Note that type and instance definitions can be nested. The example is an extract from Chapter 8 in [Lupu 1998]. In this example a help-desk role type (*helpDeskT*) is defined for a cellular GSM network company. Suppose that the network is divided into regions and each region is further subdivided into branches. Each region has a database called EIR (Equipment Identity Database) for the equipment of the region. Each branch has a database called HLR (Home Location Register) for the subscribers to the network.

The *helpDeskT* role includes an obligation policy (*customer_complaints*) to handle customer complaints; a group *hlr_managementT* specifying policies that relate to the management of an HLR database for a branch; a group *billing_and_abnormal* that contains policies related to cases of unpaid bills, stolen equipment etc. The first group is created as a type and then instantiated for the various HLR databases corresponding to each branch. The authorisation policies that authorise the access to the HLR and EIR databases are not specified directly within the role. They are instead specified as a group *HD_authorisationsT* outside the role. This could be the case if there is a need to reuse those authorisations in other roles or anywhere else within the policy specification. The role *helpDeskT* then imports the *HD_authorisationsT* group, and instantiates it for the different HLR and EIR databases to which it needs access.

File 1

```
domain /policies/groups/types;        // set the current working domain

type group HD_authorisationsT (set hd, HLR_type hlr, EIR_type eir) {

    inst auth+ HD_auth_HLR {
          subject hd;
          target  hlr;
          action  add_new_customer(), update_record(), traceHomeSubscriberInHLR();
    }

    inst auth+ HD_auth_EIR {
          subject hd;
          target  eir;
          action  blacklistEquipment();
    }
} // HD_authorisationsT
```

File 2

```
domain /tr/rr/rc/HD;                  // set the current working domain

type role helpDeskT(EIR_type eir) {

    import /policies/groups/types/HD_authorisationsT; //import the HD_authorisationsT group

    inst oblig customer_complaints {
```

```
        on customer_complaint(complaint);
        do helpDeskT.investigate_complaint(complaint);
    }

    type group hlr_managementT(HLR_type hlr) {

        inst oblig record_update {
            on      new_service_subscription(x);
            do      updateRecord(x.customer, x.service);
            target  hlr;
        }

        inst oblig consistency_loss {
            on unrecognised_customer_in_HLR(imsi);
            do hlr_managementT.checkRecord(imsi);
        }
    } // hlr_managementT

    inst  group hlr_managementBrA = hlr_managementT(hlr_branchA);
          group hlr_managementBrB = hlr_managementT(hlr_branchB);

    inst group billing_and_abnormal {

        inst oblig notify_subscriber {
            on      unpaid_bills(imsi);
            do      notifySubscriber(imsi);
            target  emailServer;
        }

        inst oblig stolen_equipment {
            on      reported_stolen(imei);
            do      blackListEquipment(imei);
            target  eir;
        }
    } // billing_and_abnormal

    inst  group hlr_auth1 = HD_authorisationsT(this.pd, hlr_branchA, eir);
          group hlr_auth2 = HD_authorisationsT(this.pd, hlr_branchB, eir);
} // helpDeskT

domain roles/HelpDesk;        // change the current working domain

inst  role helpDeskRegionA = helpDeskT(eir_regionA) @ pd/HD/HD1;
      role helpDeskRegionB = helpDeskT(eir_regionB) @ pd/HD/HD2;
```

Figure 4.12 Example policy specification

4.7 Conclusions

In this chapter we have presented features of the language which allow composition of the basic policies presented in Chapter 3. The reasons for composing basic policies are twofold: Reusability of related policy specifications, and modelling of organisational concepts such as roles, relationships and management structures. The composite policy types are:

- *Group*: A syntactic scope used to group related policies together.
- *Role*: A semantic grouping of policies, which have the same subject. Roles provide the means of grouping policies related to a position in an organisation such as a staff member, customer support manager or Chief Executive Officer (CEO). A role can also group policies relating to a specific automated agent such as one that registers new users or adaptively manages Quality of Service in a network.
- *Relationship*: A grouping of policies pertaining to the interactions between roles.

- *Management structure*: A configuration of roles and relationships into organisational units.

We introduced simple inheritance for the composite policy types and demonstrated its use to model organisational role hierarchies.

We revisited the issue of expressiveness by presenting a series of application specific constraints that can be specified for groups of policies using meta-policies. Meta-policies are constraints over a set of policies, either policies within a policy grouping or policies within a domain, and prove to be very powerful in specifying a large range or application specific constraints.

Self-management and extensibility are two of the important features of the language. Policies can be specified to control operations on other policies and domains. New types of policy that may be identified in the future can be added to the language object model to extend the language. Examples of new types of policies include: (i) *freedom policies* to permit a subject to override obligation policies e.g., an operator can override an obligation policy if she thinks there is a safety reason for doing so. These are more important for human-based policies than for automated systems and are often specified in safety-critical systems. (ii) *Resource allocation policies* to specify limits on the resources to be allocated to a session or to a service e.g., no more than 10% of available bandwidth can be allocated to a single request.

Further work remains to be done in the area of interaction protocol specification to complete the definition of relationships. However, the significance of roles, relationships and management structures in modelling organisational policies and structuring large-scale policy specifications has been demonstrated using examples.

Information about the policy language and its applications have already been published [Damianou et al. 2000a; Lupu et al. 2000a; Damianou et al. 2000b; Lupu et al. 2000b; Damianou et al. 2001; Dulay et al. 2001a; Dulay et al. 2001b].

Chapter 5

A Structural Operational Semantics

5.1 Introduction

We give an operational semantics to the policy language presented in the previous chapters by exploring the object-oriented nature of the language. The formal semantics is divided based on the different phases of the execution of a Ponder specification (also referred to as a program) as demonstrated in Figure 5.1. A policy is specified, syntactically and semantically analysed and type checked, and then stored in the domain system as a policy object. The stored policy objects can then be distributed to the components responsible for their enforcement. The runtime execution of a policy takes place in the scope of the enforcement component (i.e. management components and access controllers) in which the policy runs, and involves accessing the domain system.

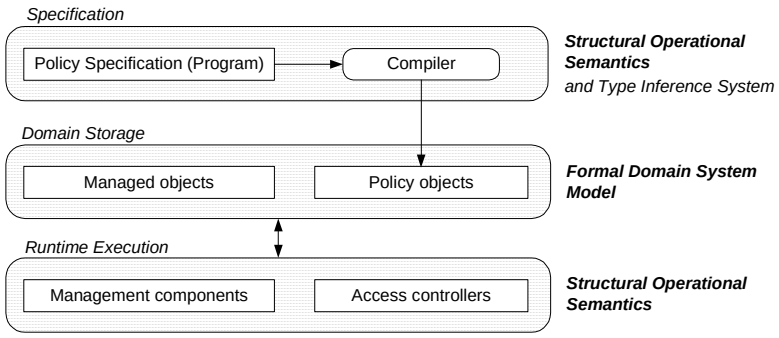


Figure 5.1 Overall semantics system

In this chapter we do not specify the static semantics of the language (i.e. a type inference system) to perform compile-time type checking for Ponder programs; we leave this for future work. We use a structural operational semantics approach to specify the dynamic semantics of policy specifications. With the dynamic semantics we address the execution of policies, including the evaluation of expressions, and the execution of commands in the policy system. The operational semantics also cover the specification of policy types and their instantiation. We formalise the domain system using Alloy [Jackson 2000], an object modelling notation. Alloy is suitable for describing structural properties but not dynamic interactions between objects, and is particularly suitable for describing hierarchical systems such as file systems, graphs etc. We have found it clearer to specify the semantics of the domain system using Alloy. However, a structural

operational approach is more suitable for the dynamic semantics of Ponder. Structural operational semantics is an operational method specifying semantics based on syntactic transformations of programs. This enables us to describe the execution of Ponder policies based on the syntax of the language, making it easier to relate the specification of the semantics to an actual implementation of Ponder. Our approach is inspired by that used in [Drossopoulou et al. 1998] which specifies the operational semantics and a type inference system for a large subset of Java. We choose *small-step* semantics [Hennessy 1990] because we believe that this makes the specification of the semantics more intuitive, and enables a detailed description of the execution for features such as authorisation filters and concurrency constraints.

Where necessary, we specify some restrictions on the syntax of Ponder, which simplify the specifications, but do not compromise or change the semantics of the language in any way. A straightforward transformation turns a Ponder program to the syntax used in this chapter. We distinguish between these restrictions and the features of the language which we do not cover, and indicate this in the text. The semantics is specified for the abstract syntax of Ponder which is included in Appendix C. For clarity reasons we omit some of rules of the operational semantics from the discussion in this chapter. The complete set of rules can also be found in Appendix C.

5.2 Overall Structure of the Operational Semantics

$$\text{Configuration} ::= \langle \text{Ponder term, state, store} \rangle \cup \langle \text{state, store} \rangle \cup \text{grant} \cup \text{deny}$$

$$\longrightarrow : \text{Configuration} \rightarrow \text{Configuration}$$

$$\text{state} ::= (\text{Ident} \rightarrow \text{Value})^{*} \cup (\text{RefValue} \rightarrow \text{PolicyObject})^{*}$$

$$\text{store} ::= (\text{Path} \rightarrow \text{PolicyObject})^{*} \cup (\text{Path} \rightarrow \text{TypeDef})^{*}$$

$$\text{PolicyObject} ::= \ll(\text{LabelName} = \text{EExpr})^{*}, \text{state}=(\text{“enabled”}|\text{”disabled”})\gg^{\text{Type}}$$

Figure 5.2 Configurations and transition rules

The operational semantics is based on a transition system, which maps configurations to new configurations, for a given Ponder program. *Configurations* consist of a Ponder term, a state and a store or just a state and a store. The *grant* and *deny* are special terminal configurations and are introduced later. The *state* is defined as a flat structure; it consists of mappings from identifiers to primitive values or to references, and from references to objects which can be runtime objects or policies. We eliminate block structure and local variables in order to avoid the use of additional structures such as program counters and higher order functions in specifying the semantics. The *store* is the repository for policy objects and policy types, as well as for other managed objects in the system. It maps domain paths to either policy objects or policy type definitions. We use the term path to refer to a domain path in this chapter.

A *policy object* is represented as a sequence of labels and associated expressions, corresponding to the elements of a policy object (i.e. subject, target, action etc.). Apart from the standard policy elements, an additional label called *state* is added to hold the state of the object (i.e. enabled or disabled). See the abstract syntax in Appendix C for a specification of the types of expressions that are assigned to the labels of a policy object (i.e. element expressions EExpr). A policy object is annotated with the full path of the type from which it is instantiated. A policy type definition is retained in its textual format as specified in the abstract syntax, and stored as is.

A rule has the format shown below; it consists of a sequence of propositions (P_i) above the horizontal line and a transition relation below the line, which holds only if all propositions P_i are true. Note that the propositions can also be transitions. Rules are assigned names for reference purposes.

$$\frac{P_1 \ \ldots \ P_n}{\langle config_i \rangle \longrightarrow \langle config_f \rangle} \quad \text{(rule name)}$$

In some of the rules, we use propositions using the existential quantifiers ∃ and ∀. These have the following format:

- $\exists\ x \in s\ (p_1\ \&\&\ p_2\ \ldots\&\&\ p_n)$, is true if propositions p_i are all true for at least one element in the finite set s.
- $\forall\ x \in s\ (p_1\ \&\&\ p_2\ \ldots\&\&\ p_n)$, is true if propositions p_i are all true for all elements in the finite set s.

We sometimes use the following transition relation: $Config_0 \longrightarrow^* Config_k$, where $Config_0$ and $Config_k$ are configurations as defined above, to indicate a finite number of applications of the $\longrightarrow$ transition relation, which takes us from $Config_0$ to $Config_k$. More formally $\longrightarrow^*$ is defined as follows: If $Config_0,\ Config_1,\ \ldots,\ Config_k$, is a finite sequence of configurations satisfying: $Config_i \longrightarrow Config_{i+1}$ for $0 \le i < k,\ k \ge 0$ then $Config_0 \longrightarrow^* Config_k$.

For the execution of obligation policies, which involve events, we introduce a second transition system to include the event histories associated with the obligation policies (Section 5.5). Refrain policies are evaluated using that transition system too, as they are seen as filters on the execution of obligation policies within the context of management components.

5.2.1 Lookup Functions, State, Store and Object Operations

We define the following operations on states, stores and policy objects. For a program P, a state σ, a store Δ and a policy object $pol = \ll e_1 = EExpr_1, \ldots, e_n = EExpr_n \gg^{Type}$ we define:

New state, $\sigma_1 = \sigma[r_1 \mapsto Obj]$, such that:
$\sigma_1(r_1) = Obj,\ \sigma_1(r_2) = \sigma(r_2)$ for $r_2 \neq r_1$

New state, $\sigma_1 = \sigma[z \mapsto val]$, such that:
$\sigma_1(z) = val,\ \sigma_1(z_1) = \sigma(z_1)$ for $z_1 \neq z$

To access the element e of an object stored at reference r_i, in state σ:
$\sigma(r_i, e) = \sigma(r_i)(e)$

To access a policy element e:
$pol(e) = EExpr_i$ if $e = e_i$, $pol(e) =$ Undefined, otherwise
e.g. $pol(subject) =$ Dse

New policy object, $pol_1 = pol[e \mapsto val]$, new state, $\sigma_1 = \sigma[r_i, e \mapsto val]$:
$pol_1(e) = val,\ pol_1(e_1) = pol(e_1)$ if $e \neq e_1$, $\sigma_1 = \sigma(r_i \mapsto \sigma(r_i)[e \mapsto val])$

To access a policy object $pol = \ll e_1 = EExpr_1, \ldots, e_n = EExpr_n \gg^{Type}$ stored in *path* in store Δ
$\Delta(path) = pol$, $\Delta(path_1) =$ undefined if $\Delta(path_1) = p$ and p is not a policy object.

New store, $\Delta_1 = \Delta[path \mapsto pol]$, such that:
$\Delta_1(path) = pol,\ \Delta_1(path_1) = \Delta(path_1)$ for $path_1 \neq path$

New store, $\Delta_1 = \Delta[path \mapsto typeDef]$, such that:
$\Delta_1(path) = typeDef,\ \Delta_1(path_1) = \Delta(path_1)$ for $path_1 \neq path$

New policy object, $pol_1 = pol[e \mapsto val]$, new store, $\Delta_1 = \Delta[path \mapsto pol_1]$:
$\Delta_1(path) = pol_1,\ \Delta_1(path_1) = \Delta(path_1)$ for $path_1 \neq path$

The following functions will be useful for the operational semantics:

- Policies(Δ, policyType) returns all policies within the store Δ which are of type: policyType $\in$ {auth+, auth-, deleg+, deleg-, oblig, refrain}.
- Object(Δ, path) returns the runtime object (i.e. managed object reference) stored in the store at the given path.

5.2.2 Modelling Runtime Commands

In order to describe the dynamic semantics of Ponder programs we model additional commands (called *runtime commands*), which are directly related to the execution of Ponder policies. We view these commands as part of an extended version of Ponder, called $Ponder_R$ in order to make the specification of the semantics less complicated: $Ponder_R$ ::= Ponder + RuntimeCommands. The transition system can then be used to specify their semantics in relation to other Ponder terms. We describe the commands we want to model informally; a formal description can be found in Appendix C:

- enable(path) enables a policy stored in path in the store.
- disable(path) disables a policy stored in path in the store.
- exec(s, t, a, $(v)^*$) models the execution of action a with parameters v_i, from subject s on target t.

- `exec(s, set`$_t$`, a, (v)`*`)` models the execution of action `a` with parameters v_i, from subject `s` on all targets in target set `set`$_t$.
- `exec(set`$_s$`, set`$_t$`, a, (v)`*`)` models the execution of action `a` with parameters v_i, from all subjects in set `set`$_s$ on all targets in target set `set`$_t$.
- `execFilter(s, t, a, (v)`*`, filter)` is the same as the `exec` command. The difference is that the filter expression given by the last argument, must be applied to the result of the method execution.
- `applyFilter(s, t, a, (v)`*`, filter)` is used only by the access control system to indicate that the filter expression given by the last argument, must be applied to the parameter values of the execution of method `a` from `s` to `t`.
- `field(t, f)` hides the details of accessing field `f` on runtime object `t`.
- `allows(pol, s, t, a, (v)`*`)` is used to indicate whether the given policy (an auth+) allows the execution of action `a` with parameters v_i from subject `s` on target `t`. The result is false if the policy doesn't explicitly allow this action, or true if it does. If the policy also has a filter expression associated with the action, then an `execFilter` command is returned instead if the policy explicitly allows the action execution.
- `disallows(pol, s, t, a)` is used to indicate whether the given policy (an auth-) disallows the execution of action a from subject s on target t. The result is false if the policy doesn't explicitly disallow the action execution or true otherwise.
- `delegate(s, g, (a)`*`)` executes the delegation of the actions given in the last parameter, from the subject to the given grantee object.

Note that *subject* and *target* are actual references to runtime objects stored in Δ. We simplify the policy life-cycle (introduced in [Marriott 1997]) and only model the enabling and disabling of Ponder policies as shown in Figure 5.3. In other words we consider the policies as being enabled once they are distributed to their enforcement components. When disabled, a policy is also retracted from the enforcement components responsible for enforcing it. This simplifies the specification of the semantics. We describe the policy life-cycle management in more detail in Chapter 7.

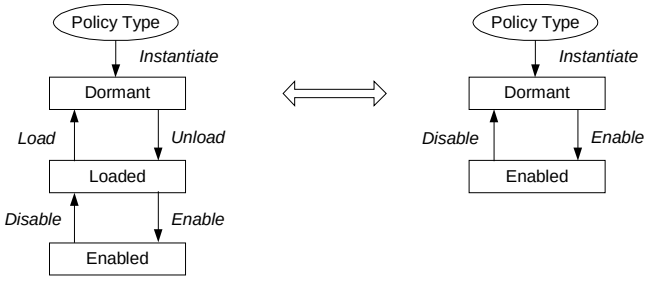


Figure 5.3 Policy life-cycle

5.3 Authorisation Policy Semantics

We do not place any special restrictions on the access control system. We consider a classical model in which access control is enforced by a reference monitor, which mediates every attempted access by a user (or program executing on behalf of the user) to objects in the system [Sandhu et al. 1994]. Figure 5.4 from [Sandhu et al. 1994] shows the enforcement of access control within a system at a logical level. Ponder is not concerned with authentication which ideally exists separately from access control in a security system. In line with Ponder terminology we use the term *subject* to refer to principals and users (or programs running on behalf of users), which attempt to access objects in the system. We call the objects being accessed managed objects, *target* objects or simply targets, and refer to a reference monitor as an *access controller*. One could assume a single access controller for the entire system, or any number of distributed access controllers each one responsible for a specific set of targets. The access control database is the repository of authorisation policy objects, i.e. the store Δ.

Administration is an important part of an access control system. This has two aspects: The first is the specification of policies to control access to already existing policies. This is handled implicitly by the fact that policies are objects and as such can be managed objects as well. The second aspect is the management of the policy life-cycle which is addressed in the semantics (see Section 5.3.4).

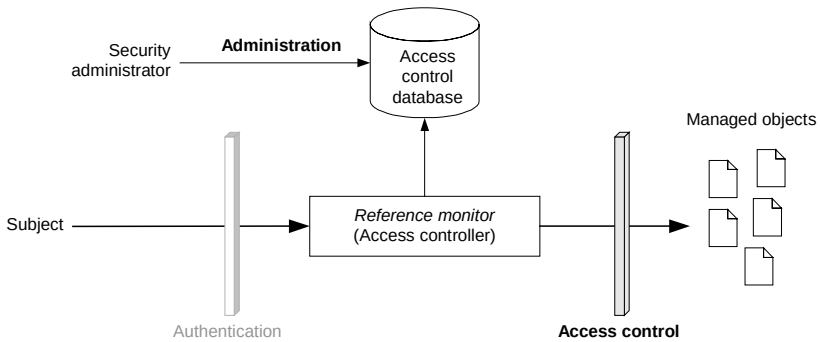


Figure 5.4 Access control model

5.3.1 Program Execution, Types and Instantiation

We simplify the syntax of the policy language, in order to simplify the task of specifying the semantics, as follows. Note that the full Ponder can be translated to the subset defined by these restrictions:

- We allow constants to be specified only at the outer level in the syntax of the language. Not within the scope of basic or composite policies.
- Although the elements of a policy can be specified in any order, we require that they are specified in a fixed order here.

- We require that the constraint is not omitted; if it is, then *true* is specified instead.
- We require that paths are always absolute paths.
- We require that a policy instance is always instantiated from a policy type.
- We assume that an identifier is never assigned to the subject or the target domain scope expression. If it is, then we replace that with a constant definition of type *set*.

  `subject id = Dse` becomes: `set id = Dse; subject Dse`
- We convert the specification of the type of the subject/target into a constraint as follows:

  `subject <router> /path` becomes: `subject /path; when subject.oclistypeof(router);`
- We do not allow a subject/target to be specified as part of the formal parameters of a type. A set can be specified instead and then assigned to the subject/target element of the policy.
- We assume that a filter always has a condition associated with it. If not, *true* is used instead. Filters can cause ambiguity if there are more than one policies with an overlap of subjects, targets and actions, with the same filter on the same action. In that case, the application of filters on the execution of the action is non-deterministic, and depends on which policy is evaluated first. In this specification we assume that there are no two positive authorisation policies with an overlap of subjects, targets and actions and with the same filter on the same action. This can be provided using analysis of the policies at specification time to prevent the definition of the second policy.

We do not handle the following:

- We change the definition of constants which are *user* or *external* types to be just paths instead of expressions.
- We omit prefixes to the actions to specify a specific object within the subject/target domain instead of all objects
- We omit the *import* statement; it is not needed since all the paths are absolute.
- We omit the domain statement used to declare the current working domain; it is not needed since all paths are absolute.
- We do not handle the definition of events and constraints.

The execution of a given Ponder program proceeds sequentially one statement at a time as indicated by rules (program1), (program2) shown in Appendix C. A type definition is stored in the store Δ under the specified path; this is described in the (type auth+) rule shown below. An instantiation retrieves the type from the store Δ, evaluates the expressions passed as parameters to the instantiation, replaces the evaluated values for those parameters in the body of the type, and stores the result as a policy object in the store at the appropriate path. The state of the policy object is set to *disabled*. Note that the term $t[t_1/x]$ has the meaning of replacing the variable x by the term t_1 in the term t. This is used in the (inst auth+) rule to replace the parameter identifiers with

the actual values of the parameters in the body of the authorisation policy. The same rules apply for negative authorisation, obligation and refrain policies.

$$\frac{\Delta_1 = \Delta[path/t \mapsto \texttt{type auth+ } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bauth+\ \}]}{\langle \texttt{type auth+ } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bauth+\ \}, \sigma, \Delta\rangle \longrightarrow \langle \sigma, \Delta_1\rangle}\ \text{(type auth+)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ \Delta(path/t) = \texttt{type auth+ } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bauth+\ \} \\ \langle e_i, \sigma, \Delta\rangle \longrightarrow^* \langle val_i, \sigma_1, \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ Bauth+_1 = Bauth+[z_1/x_1, \ldots, z_n/x_n][state \mapsto disabled] \\ \sigma_3 = \sigma_2[r_1 \mapsto \ll Bauth+_1 \gg^{path/t}] \\ \Delta_1 = \Delta[path_1/a \mapsto \ll Bauth+_1 \gg^{path/t}] \end{array}}{\langle \texttt{inst auth+ } path_1/a = path/t(e_1, \ldots, e_n), \sigma, \Delta\rangle \longrightarrow \langle r_1, \sigma_3, \Delta_1\rangle}\ \text{(inst auth+)}$$

5.3.2 Action Execution, Access Control Decision

The following rules show how an access controller makes a decision to grant or deny an action execution. We assume that there is only one (global) store for the policies, and introduce two explicit configurations to represent the *grant* and *deny* states of the system. When the system reaches the grant configuration on the execution of an action, this means that the execution is granted. Similarly, the deny configuration means that the execution is denied. These are called *terminal* configurations; in other words we do not provide any further details on the execution of these system states.

If there exists a positive authorisation policy in the store which allows the action execution, and no negative policy which disallows it, then the action execution is granted as described in the (exec grant) rule. We use the execution of the command `allows` to determine whether a positive authorisation allows a specific action execution. Similarly, we evaluate the execution of the `disallows` command to determine if a negative authorisation policy disallows an action execution. We describe the execution of these two commands in more detail later in this section. If there exists a filter associated with the authorisation policy which allows the action to execute, then the access controller evaluates to an `applyFilter` command instead of a simple grant as shown in (exec grant filter). The execution of the `applyFilter` is described later, and indicates that the action is granted, but the specified filter must be applied to the parameters of the execution.

$$\frac{\begin{array}{l} \exists\ pol_1 \in Policies(\Delta, \texttt{auth+})\ (\langle \texttt{allows}(pol_1, s, t, a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle \texttt{true}\ , \sigma, \Delta\rangle) \\ \forall\ pol_2 \in Policies(\Delta, \texttt{auth-})\ (\langle \texttt{disallows}(pol_2, s, t, a), \sigma, \Delta\rangle \longrightarrow \langle \texttt{false}\ , \sigma, \Delta\rangle) \end{array}}{\langle \texttt{exec}(s, t, a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \texttt{grant}}\ \text{(exec grant)}$$

$$\frac{\begin{array}{l} \exists\ pol_1 \in Policies(\Delta, \texttt{auth+})\ (\langle \texttt{allows}(pol_1, s, t, a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \\ \langle \texttt{applyFilter}(s, t, a, v_1 \ldots v_n, filterExpr), \sigma, \Delta\rangle) \\ \forall\ pol_2 \in Policies(\Delta, \texttt{auth-})\ (\langle \texttt{disallows}(pol_2, s, t, a), \sigma, \Delta\rangle \longrightarrow \langle \texttt{false}\ , \sigma, \Delta\rangle) \end{array}}{\langle \texttt{exec}(s, t, a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle \texttt{applyFilter}(s, t, a, v_1 \ldots v_n, filterExpr), \sigma, \Delta\rangle}\ \text{(exec grant filter)}$$

If there exists a negative authorisation policy in the store that disallows the action execution, then the execution of the action evaluates to deny (exec deny). The same is true if no policy is specified for the action; in other words an action execution is denied by default as shown in (exec default).

$$\frac{\exists\ pol \in Policies(\Delta, auth\text{-})\ (\langle disallows(pol, s, t, a), \sigma, \Delta\rangle \longrightarrow \langle true\ , \sigma, \Delta\rangle)}{\langle exec(s,t,a,v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow deny}\ (\text{exec deny})$$

$$\frac{}{\langle exec(s,t,a,v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow deny}\ (\text{exec default})$$

Applying Authorisation Filters

The following two rules show the execution of the filters i.e. the execution of the `applyFilter` command. The filters are evaluated in order as shown by the first rule, and execution stops at the first filter whose condition evaluates to true. The second rule shows that when the condition expression (i.e. $expr_c$) of a filter evaluates to true, the expressions inside the filter are applied to the parameter values of the action execution. The evaluation of that filter returns an `execFilter` command, because the filters must be applied later to the result of the execution. The `execFilter` command indicates that the action execution is granted, and is not elaborated any further in this transition system as shown by the rule (auth exec filter done). We will see how it affects execution of the `exec` command in the transition system for obligation policies (Section 5.5), where the filters must be applied to the result of the execution.

$$\frac{filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n; \text{ result } = expr_r\}\quad \langle expr_c, \sigma, \Delta\rangle \longrightarrow \langle false, \sigma_1, \Delta\rangle}{\langle applyFilter(s, t, ba, v_1 \ldots v_n, filter_1 \ldots filter_k), \sigma, \Delta\rangle \longrightarrow \langle applyFilter(s, t, ba, v_1 \ldots v_n, filter_2 \ldots filter_k), \sigma_5, \Delta\rangle}\ (\text{auth exec filter next})$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n; \text{ result } = expr_r\} \\ \langle expr_c, \sigma_1, \Delta\rangle \longrightarrow \langle true, \sigma_2, \Delta\rangle \\ \sigma_3 = \sigma_2[p_1 \mapsto v_1] \ldots [p_n \mapsto v_n] \\ expr'_i = expr_i[v_i/p_i] \\ \langle expr'_I, \sigma_3, \Delta\rangle \longrightarrow^* \langle v'_i, \sigma'_3, \Delta\rangle \text{ for } I \in \{1 \ldots n\} \end{array}}{\langle applyFilter(s, t, a, v_1 \ldots v_n, filter_1 \ldots filter_k), \sigma, \Delta\rangle \longrightarrow \langle execFilter(s,t,a,v'_1 \ldots v'_n, filterExpr), \sigma_5, \Delta\rangle}\ (\text{auth exec filter})$$

$$\frac{}{\langle execFilter(s,t,a,v_1 \ldots v_n, filterExpr)\ \sigma, \Delta\rangle \longrightarrow \langle \sigma, \Delta\rangle}\ (\text{auth exec filter done})$$

We now describe in more detail the execution of the command `allows` which determines whether an auth+ policy allows an action execution, and that of `disallows` which determines whether an auth- policy disallows an action execution. The evaluation of both of these commands was used in the rules described above to make an access control decision. Their execution involves evaluation of the subject and target domain scope expressions, as well as evaluation of the policy constraints. We introduce the helper function `filter(a, pol)`, to return the filter expression for a given action a in a given positive authorisation policy `pol`.

The allows command

If there is no filter associated with the action of an `allows` command, it evaluates to either true or false. If there is a filter then it evaluates either to an `applyFilter` command if the action is allowed, or to false. The rule (allows true) specifies that a policy `pol`, allows the execution of action `a` from `s` to `t` if all of the following are true: (i) the policy is a positive authorisation, (ii) it is enabled, (iii) `s` belongs to the subject set of the policy, (iv) `t` belongs to the target set of the policy, (v) `a` belongs to the set of actions of the policy, and (vi) the constraint of the policy evaluates to true. Note that in this rule, there is no filter associated with the action as specified by the last proposition above the line. If there was a filter associated with action a then the execution would evaluate to an `applyFilter` command. This is described by the rule (allows true filter) found in Appendix C.

$$\frac{\begin{array}{l} pol \in Policies(\Delta, auth+) \\ pol(state) = enabled \\ \langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s\ , \sigma, \Delta\rangle \\ s \in set_s \\ \langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t\ , \sigma, \Delta\rangle \\ t \in set_t \\ a \in pol(action) \\ r_1, r_2 \text{ are new in } \sigma \\ \sigma_1 = \sigma[r_1 \mapsto t] \\ \sigma_2 = \sigma_1[r_2 \mapsto s] \\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle true\ , \sigma_2, \Delta\rangle \\ filter(a, pol) == \text{“”} \end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle true, \sigma, \Delta\rangle} \quad \text{(allows true)}$$

If any of the propositions described above is false, then the execution of `allows` evaluates to false. This is described by the five rules shown below.

$$\frac{pol \notin Policies(\Delta, auth+)}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \quad \text{(allows false 1)}$$

$$\frac{\begin{array}{l} \langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s\ , \sigma, \Delta\rangle \\ s \notin set_s \end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \quad \text{(allows false 2)}$$

$$\frac{\begin{array}{l} \langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t\ , \sigma, \Delta\rangle \\ t \notin set_t \end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \quad \text{(allows false 3)}$$

$$\frac{a \notin pol(action)}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \quad \text{(allows false 4)}$$

$$\frac{\begin{array}{l} r_1, r_2 \text{ are new in } \sigma \\ \sigma_1 = \sigma[r_1 \mapsto t] \\ \sigma_2 = \sigma_1[r_2 \mapsto s] \\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle false, \sigma_2, \Delta\rangle \end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \quad \text{(allows false 5)}$$

In the rules: (allows true) and (allows false 5), before we evaluate the constraint of the policy, we add the target and subject objects in the current state. That's because the constraint may require the target/subject object when being evaluated. If a constraint is either subject or target based, the

policy is not always valid or always invalid. It may be valid/invalid for each of the targets in the target domain, independently. The example in Figure 5.5 shows how the given authorisation policy is valid for one of the target objects but not for the other two.

```
inst auth+ A {
      subject   s = … ;
      target    t = /T ;
      action    … ;
      when      (t.status = "active");
}
```

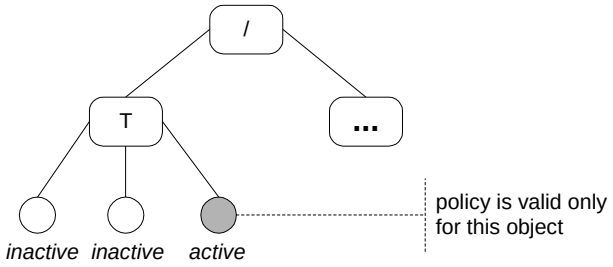


Figure 5.5 Example constraint on target state

The disallows command

The evaluation of the `disallows` command is very similar to the evaluation of the `allows` command. The only difference is that in this case the action parameters are irrelevant, and we are not concerned with filters since negative authorisation policies do not have filters. The interested reader can see Appendix C for the rules describing the execution of the `disallows` command.

5.3.3 Constraints and Subject/Target Evaluations

The evaluation of the constraint, subject and target elements of a policy equates to the evaluation of the corresponding expression or domain scope expression. The (constraint) rule specifies that in order to evaluate the constraint of a policy, we evaluate the constraint-expression and return the value of that execution. Similarly targets and subjects are evaluated to the set of objects to which the corresponding domain scope expression evaluates, as shown in rules (subject) and (target).

$$\frac{pol(constraint) = expr \qquad \langle expr, \sigma, \Delta\rangle \longrightarrow^{*} \langle val, \sigma_1, \Delta\rangle}{\langle pol(constraint), \sigma, \Delta\rangle \longrightarrow \langle val, \sigma_1, \Delta\rangle} \quad (constraint)$$

$$\frac{pol(subject) = dse \qquad \langle dse, \sigma, \Delta\rangle \longrightarrow^{*} \langle set, \sigma_1, \Delta\rangle}{\langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set, \sigma_1, \Delta\rangle} \quad (subject)$$

$$\frac{pol(target) = dse \qquad \langle dse, \sigma, \Delta\rangle \longrightarrow^{*} \langle set, \sigma_1, \Delta\rangle}{\langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set, \sigma_1, \Delta\rangle} \quad (target)$$

Expressions

We describe the rules that provide the operational semantics for the basic expression types in the abstract syntax. The complete set of rules for the evaluation of expressions is given in Appendix C. We call *ground terms*, those terms that cannot be further rewritten. A term `t` is ground `iff t` is a primitive value (i.e `PrimValue`), a set value (i.e. `SetValue`), or $t=r_i$ for some i (i.e. a reference). We first show how constant definitions add new bindings to the state. This is demonstrated by the

(primType) rule where an expression is assigned to a primitive type identifier (e..g Integer, String etc). The rule evaluates the expression, and adds the identifier to the state σ bound to the value to which the expression evaluates. The rest of the constant types are evaluated in the same way and can be found in the appendix.

$$\frac{\langle expr,\ \sigma,\ \Delta\rangle \longrightarrow^{*} \langle val,\ \sigma_1,\ \Delta\rangle}{\langle PrimType\ id = expr,\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma_1[id \mapsto val],\ \Delta\rangle}\ (primType)$$

The abstract syntax for expressions defines the following expression types (see Appendix C): `Value | Var | Expr.methodName((Expr|Dse)*) | Expr.fieldName | OCLexpr.`

The evaluation of values and variables is simple. Values do not need any evaluation, and variables evaluate to the value to which the variable is bound in the state σ. `OCLexpr` expressions are based on the OCL syntax version 1.3 [OMG 1999b]. We assume the function `evalOCL(σ, Δ, OCLexpr)` which evaluates an `OCLexpr` using state σ and store Δ. We do not cover the operational semantics of the execution of the `evalOCL` function. Work towards the operational semantics of OCL can be found in a number of recent papers: [Hami et al. 1998; Richters et al. 1998; Cengarle et al. 2001].

$$\frac{evalOCL(\sigma,\ \Delta,\ OCLexpr) = val}{\langle OCLexpr,\ \sigma,\ \Delta\rangle \longrightarrow \langle val,\ \sigma,\ \Delta\rangle}\ (OCL\ expr)$$

The two remaining expression types are method calls and field accesses. We discuss the rules which describe the execution of a method call; field access rules are very similar. The first thing required for the execution of a method call is the evaluation of the parameters. The parameters (i.e. expressions) are evaluated one at a time from left to right as shown in rule (method call 1).

$$\frac{\begin{array}{l} val_j \text{ is ground for } j \in \{1..k\text{-}1\} \\ \langle expr_k,\ \sigma,\ \Delta\rangle \longrightarrow \langle expr'_k,\ \sigma',\ \Delta\rangle \end{array}}{\begin{array}{l} \langle expr_1.m(val_2,\ldots,val_{k\text{-}1},\ expr_k,\ldots expr_n),\ \sigma,\ \Delta\rangle \longrightarrow \\ \langle expr_1.m(val_2,\ldots,val_{k\text{-}1},\ expr'_k,\ldots expr_n),\ \sigma',\ \Delta\rangle \end{array}}\ (method\ call\ 1)$$

After the parameters have been evaluated, the method call is essentially translated to an action execution, modelled using the `exec` command. This means that the execution of a method call during the evaluation of an expression (e.g. the constraint of a policy) results in actual calls in the system which are also subject to the access control enforcement described previously.

There are three types of rules based on what the prefix expression of a method call evaluates to. If the prefix expression evaluates to a reference then the target of the action call becomes the object stored in state σ at the evaluated reference, and its either a policy object, a subject object or a target object as shown in rule (method call 2). If the prefix expression evaluates to a path, the target of the action call is the object stored at `Δ(path)` (i.e. a runtime object) as shown in rule (method call 3). Finally, if the prefix expression evaluates to an identifier then the target becomes the object to which the identifier is bound in σ as shown in rule (method call 4), i.e. `σ(id)`. Note that the subject

of the method call is always the current object in the context of which the expression is evaluated. For example, this can be the access controller if the expression is evaluated as part of the constraint of an authorisation policy during an access check.

this identifies the object in the context of which the call is made

$$\frac{\langle expr_1, \sigma, \Delta\rangle \longrightarrow \langle r_i, \sigma_1, \Delta\rangle}{\langle expr_1.m(val_2 \ldots val_n), \sigma, \Delta\rangle \longrightarrow \langle exec(this, \sigma_1(r_i), m, val_1..val_n)), \sigma_1, \Delta\rangle} \quad \text{(method call 2)}$$

this identifies the object in the context of which the call is made

$$\frac{\langle expr_1, \sigma, \Delta\rangle \longrightarrow \langle path, \sigma_1, \Delta\rangle \quad obj = Object(\Delta, path)}{\langle expr_1.m(val_2 \ldots val_n), \sigma, \Delta\rangle \longrightarrow \langle exec(this, obj, m, val_1..val_n)), \sigma_1, \Delta\rangle} \quad \text{(method call 3)}$$

this identifies the object in the context of which the call is made

$$\frac{\langle expr, \sigma, \Delta\rangle \longrightarrow \langle id, \sigma_1, \Delta\rangle}{\langle expr_1.m(val_2 \ldots val_n), \sigma, \Delta\rangle \longrightarrow \langle exec(this, \sigma_1(id), m, val_1..val_n)), \sigma_1, \Delta\rangle} \quad \text{(method call 4)}$$

Field access rules evaluate to a `field` command (see Section 5.2.2) similarly to the way method calls map to action executions (i.e. to the `exec` command). However, if the prefix expression evaluates to a reference r_i, then the field access simply evaluates to $\sigma(r_i, f)$. Note that the execution of the field command does not change the state or the store. The complete set of rules for field access expressions can be found in Appendix C.

Domain Scope Expressions

We introduce the following functions to help in evaluating domain scope expressions:

- `eval(path, Δ)` returns the set of objects under the given path in Δ.
- `eval@(path, n, Δ)` returns the set of objects under the given path in Δ, navigating up to `n` levels deep, excluding domain objects.
- `eval*(path, n, Δ)` returns the set of objects under the given path in Δ, navigating up to `n` levels deep, including domain objects.
- `applySetOp(SetOp, set1, set2)` applies the set operator `SetOp` to the two sets of objects: set_1 and set_2.

The evaluation of domain scope expressions of type: `path`, `*n path` and `@n path` proceeds by calling the corresponding `eval` method defined above. An action call on a domain object identified by a path, is executed as an `exec` command, similar to how a method call is executed as shown in rule (dse action call). A feature call expression is evaluated using the `evalOCL` function since it is an `OCLexpr` as described in rule (dse feature call). This includes expressions of the form: `subject /path1->select(s | s.state = “idle”);`

$$\frac{obj = Object(\Delta, path)}{\langle path.m(val_2 \ldots val_n), \sigma, \Delta\rangle \longrightarrow \langle exec(this, obj, m, val_1..val_n)), \sigma, \Delta\rangle} \quad \text{(dse action call)}$$

$$\frac{evalOCL(\sigma, \Delta, path->featureCall) = \{Obj_1 \ldots Obj_n\}}{\langle path->featureCall, \sigma, \Delta\rangle \longrightarrow \langle \{Obj_1 \ldots Obj_n\}, \sigma, \Delta\rangle} \quad \text{(dse feature call)}$$

The evaluation of composite domain scope expressions of the form dse_1 setOp dse_2 proceeds by first evaluating dse_1, then dse_2, and then calling the function applySetOp defined above to the resulting sets. The complete set of rules for domain scope expressions can be found in Appendix C.

5.3.4 Policy Life-Cycle Commands

The following two rules show the execution of the enable and disable commands, which control the policy life cycle. The enable operation accesses the policy from the given path, changes the state of the policy to *enabled*, and stores it back to the store Δ. The disable operation is executed in the same way but changes the state to *disabled*. Note that the two rules shown below apply to all types of policies; the only thing that changes is the first proposition of each rule.

$$\frac{pol \in Policies(\Delta, auth+) \quad pol = \Delta(path) \quad pol_1 = pol[state \mapsto enabled] \quad \Delta_1 = \Delta[path \mapsto pol_1]}{\langle enable(path), \sigma, \Delta \rangle \longrightarrow \langle \sigma, \Delta_1 \rangle} \text{ (enable auth+)}$$

$$\frac{pol \in Policies(\Delta, auth+) \quad pol = \Delta(path) \quad pol_1 = pol[state \mapsto disabled] \quad \Delta_1 = \Delta[path \mapsto pol_1]}{\langle disable(path), \sigma, \Delta \rangle \longrightarrow \langle \sigma, \Delta_1 \rangle} \text{ (disable auth+)}$$

5.4 Delegation Policies

Before we proceed to the specification of the semantics for delegation policies, we identify some restrictions to the syntax of delegation policies. Note that these restrictions do not change the semantics of Ponder:

- The elements of the delegation policy are specified in a predetermined order.
- The specification of the *valid*-clause and the *hops*-clause for the positive delegation policy is required. If not specified, *true* is assumed for the valid constraint, and –1 for the hops. We assume –1 means no restriction on the number of cascading delegation hops.

We do not investigate the case of passing delegation policies as parameters, from which to derive access rights, to other delegation policies.

5.4.1 Mapping Delegation to Authorisation Policies

In this section we describe informally how a delegation policy maps to authorisation policies; we then present the transition rules for the operational semantics. The main problem with mapping a delegation policy to authorisations will be the handling of the delegation constraints. Here is a reminder of the various types of delegation constraints (see Section 3.3.4):

- Time restrictions (duration, validity period) to specify the duration or the period over which the delegation should be valid before it is revoked.
- Any arbitrary constraint based on system attributes or subject/target/grantee or action attributes.
- Maximum number of cascading delegations allowed (i.e. maximum number of delegation hops or levels)

The first two types of constraints are specified using the *valid*-clause of a delegation policy. The third one is specified separately as a number following the *hops*-clause. We use the following simple example to demonstrate the mapping of a delegation policy to authorisation policies. Assume the following delegation policy (D_1) and associated positive authorisation (A_1), shown in pseudo format below, where $S_1 \subseteq S$, $T_1 \subseteq T$, `deleg-actions` $\subseteq$ `auth-actions`.

```
inst auth+ A1 {
        subject S
        target  T
        action  auth-actions
        when    C
}
```

```
inst deleg+ D1 (A1) {
        subject S1
        target  T1
        grantee G1
        action  deleg-actions
        when    C1
        valid   C2
        hops    n
}
```

In our specification, we assume an authorisation service with the following interface:

- `delegate(g, actionList)` executes the delegation of the actions in the `actionList` to the given grantee object `g`.
- `revoke(g, actionList)` executes the revocation of the actions in the `actionList` from the given object.
- `cascading(d, g)` returns the number of delegation hops performed with respect to the given delegation policy `d`, starting at the given grantee object `g`. The same delegation policy may have many different cascaded delegation paths as demonstrated in Figure 5.6. This requires some book-keeping by the authorisation service.

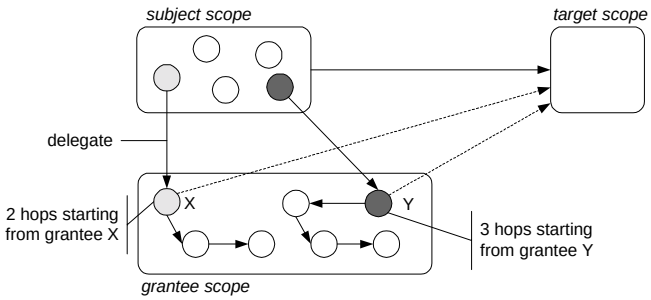


Figure 5.6 Delegation hops

The authorisation policy (AD_1) shown below is created when the delegation instance is compiled. AD_1 authorises the subject S_1 (i.e. grantor) to execute the method '`delegate`' on the authorisation service with grantee `g` as a parameter of the method in order to delegate the set of access rights (i.e.

actionList) specified by the delegation policy (D_1) to that grantee. Note that the number of hops of D_1 is specified in the constraint of the generated policy AD_1, and the *when*-clause constraint of the delegation policy is ANDed to the constraint of AD_1, to restrict its validity based to the validity of the delegation policy. A negative delegation policy maps similarly to a negative authorisation policy. In that case we don't need to map the delegation hops though, since the negative delegation policy does not specify hops.

```
inst auth+ AD1 {
    subject   S1
    target    AS = AuthService
    action    delegate(g, actionList)
    when      C1 and G1->includes(g) and deleg-actions->includes(actionList) and
              (AS.cascading(D1, g) < n)
}
```

The grantor may never exercise the right to delegate actions to a grantee. If it doesn't, then access control is based only on the existing authorisation policies. However, a second authorisation policy (AD_2) is dynamically created by the authorisation service when the action: `delegate(G, action-list)` is executed. This is true only for positive delegation policies. Note that the constraint of the original authorisation policy (A_1) is part of the constraint of AD_2 and is ANDed with the `valid`-clause constraint of D_1. This specifies that the grantee can only execute the access rights whenever the original subject can, and that the access rights are valid only as long as the *valid*-clause constraint of the delegation policy is true.

```
inst auth+ AD2 {
    subject   G
    target    T1
    action    actionList
    when      C2 and C
}
```

The following authorisation policy is also created to allow for cascaded delegation; i.e. to enable a grantee who has been delegated the access rights to delegate them further to other members of the grantee scope.

```
inst auth+ AD3 {
    subject   G
    target    AS = AuthService
    action    delegate(g, actionList)
    when      C1 and G1->includes(g) and deleg-actions->includes(actionList) and
              (AS.cascading(D1, g) < n)
}
```

The execution of a revocation action causes the authorisation service to delete the last two generated authorisation policy instances (AD_2 and AD_3).

5.4.2 Semantic Rules

Type definitions for both positive and negative delegation policies are very similar to those specified for authorisation policies (see Section 5.3.1); the type is stored in the store Δ as is. However, the instantiation of a delegation policy is different, since it must generate an

authorisation policy as informally described previously. The following rule (inst deleg+) shows the execution of the instantiation of a positive delegation policy. A delegation instance is created from which the authorisation policy is constructed as described above using a purely syntactical process. Since in the semantics we require the instantiation of policies from policy types, we first generate an authorisation policy type with no parameters, which can then be instantiated to get the desired authorisation policy instance. Thus the execution of the instantiation of the delegation policy amounts to executing the instantiation of the generated authorisation policy type as indicated by the conclusion of the rule.

The authorisation policy is stored in the path where the delegation instance is stored. Note that the delegation policy instance is also stored in the domain store because we need it later to construct the other authorisation policies during the execution of the delegate method. The rule for a negative delegation policy is very similar. The only thing that changes is the syntactic construction of the corresponding delegation policy, which does not require the constraint for the delegation hops.

z_i are new identifiers in σ
r_1 is new in σ
authPol := $\Delta(path_2/a)$
$\Delta(path/t)$ = type deleg+ path/t (T_0 a) (T_1 x_1, … , T_n x_n) { Bdeleg+ }
$\langle e_i, \sigma, \Delta\rangle \longrightarrow \langle val_i, \sigma_1, \Delta\rangle$
$\sigma_2 = \sigma_1[z_0 \mapsto$ authPol$]$ $[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n]$
$Bdeleg+_1$ = Bdeleg+$[z_0/a, z_1/x_1, \ldots, z_n/x_n]$[state $\mapsto$ disabled]
dPol := $\ll Bdeleg+_1 \gg^{path/t}$
$\sigma_3 = \sigma_2[r_i \mapsto$ dPol$]$
$\Delta_1 = \Delta[path_1/d \mapsto$ dPol$]$
adPolTypeDef := “type auth+ $path_1/t_2$() {subject dPol(subject) ; target AS = AuthService; action delegate(g, actionList); when dPol(constraint) and dPol(action)->includes(actionList) and (AS.cascading($path_1$/d, g) < dPol(hops) }”

$$\frac{\langle adPolTypeDef, \sigma_3, \Delta_1\rangle \longrightarrow \langle \sigma_3, \Delta_2\rangle}{\langle \text{inst deleg+ } path_1/d = path/t\ (path_2/a)\ (e_1, \ldots, e_n), \sigma, \Delta\rangle \longrightarrow \langle \text{inst auth+ } path_1/a_1 = path_1/t_2(), \sigma_3, \Delta_2\rangle}\ (\text{inst deleg+})$$

Delegating Access Rights

The execution of the `delegate` command is formalised in way similar to that described for the instantiation of a delegation policy with the (inst deleg+) rule. The difference is that the rule for the delegation generates two authorisation polices as we described informally in the previous subsection. The (delegate) rule can be found in the appendix; we omit its description here for clarity. Note that we assume there is only one delegation policy to authorise a particular delegation of actions from the subject to the grantee. If more than one delegation policy allows the delegation of the actions from the subject to the target, we assume that analysis at the specification time selects only one based on specificity or prioritisation.

5.5 Obligation Policies

In the semantics of obligation policies described in this section, we do not cover the following:

- We do not specify the semantics for the catch-clause, i.e. the exception mechanism, for obligation policies.
- We do not include the case of using a path as a prefix to an obligation action (to indicate an action on a specific object in the target set instead of the entire target set).

We assume that an action always has a prefix. So if a prefix is omitted we always assign the subject prefix to it. We simplify the semantics by only allowing the keyword *subject* or *target* to be specified as the prefix to the action. Subject means, the current subject, where as target means the target set of the policy.

5.5.1 Events

Obligation policies are event-triggered rules. This requires a different transition system to the one used for the semantics of authorisation and delegation policies to include the event histories, which trigger obligation policies. We adopt an approach similar to that described in [Lobo et al. 1999] to specify the semantics for PDL, an event-based policy language. We interpret policies over event histories where an event history is a sequence of event instances as defined below.

A named event is an event of the form: `ident(parameters)`; it has a set of attribute names associated with it, and denotes a *class* of events. An *event instance* includes values for all of the attribute names of an event class. We adopt the following syntax: for a named event *e* (event class), e^i is an instance of *e*. We denote an event history with H, and we use E to stand for any event, composite or basic. We denote an empty history by ε. For the sake of simplicity we ignore Timer events. A Timer event can be associated with a named event, which has no parameters; e.g. for `Timer.at("17:00:00")` we can assume a named event `TimeIsFivePM` which occurs at 05:00pm everyday. Table 3.2 describes the operators used to compose events in Ponder (see Section 3.4.1).

We check whether an event E has occurred in an event history H, by checking whether there exists a *minimal event history* for E in H. Borrowing the idea from [Lobo et al. 1999], we define a minimal history of an event E as follows:

An event history $H = e^i_1, \ldots, e^i_n$ is a minimal history of E if and only if one of the following conditions is true:

- E is a basic event = $e(f_1, \ldots, f_k)$. In this case n = 1. The history contains only one event and there is an instance $e^i(v_1, \ldots, v_k)$ of event $e(f_1, \ldots, f_k)$ such that $e^i = e^i_1$. We define (e^i , E) to be a trace T of E in H.

- $E = E_1 \rightarrow E_2$, and there exists a minimal history H_i for each E_i such that $H = H_1, H_c, H_2$, where H_c could be ε, and H_c is not a minimal history for E_1 or E_2. If T_1 is a trace of E_1 in H_1 and T_2 is a trace of E_2 in H_2, then $T = T_1, T_2$ is a trace of E in H.
- $E = E_1 \mid E_2$, and there exists a minimal history H_i for either E_1 or E_2, such that $H = H_i$. If T_i is a trace of E_i in H, then T_i is a trace of E in H.
- $E = E_1$ && E_2, and there exists a minimal history H_i for each E_i such that $H = H_1, H_c, H_2$, or $H = H_2, H_c, H_1$, where H_c could be ε, and H_c is not a minimal history for E_1 or E_2. If T_1 is a trace of E_1 in H_1 and T_2 is a trace of E_2 in H_2, then $T = T_1, T_2$ or $T = T_2, T_1$ is a trace of E in H.
- $E = (E_1)$ and H is a minimal history of E_1. Any trace of E_1 in H is a trace of E in H.
- $E = m * E'$ with $m > 0$. This is equivalent to $E'_1 \rightarrow E'_2 \ldots \rightarrow E'_m$. The satisfaction of this is based on the second case.
- $E = E_1$ + time. We define a named event e_2 with no parameters, which occurs *time* seconds after E_1. So $E = E_1 \rightarrow e_2$. The satisfaction of this is the same as in the second case.
- $E = \{ E_1 ; E_2 \} ! E_3$, and there exists a minimal history H_1 for E_1, and a minimal history H_2 of E_2, such that $H = H_1, H_c, H_2$, and H_c is not a minimal history of E_3. If T_1 is a trace of E_1 in H_1 and T_2 is a trace of E_2 in H_2, then $T = T_1, T_2$ is a trace of E in H.

We define a function which determines whether an event E occurs in a given event history $H = e^i_1, \ldots, e^i_n$.:

$occ ::= event\ history, event \rightarrow (true \mid false)$
$occ(H, E) = true$, if there exists a j such that $H_m = e^i_j, \ldots, e^i_n$ is a minimal history of E, and a trace T of E in this minimal history.
$occ(H, E) = false$, otherwise.

We define as $eAttrs(E)$ the list of attribute names of the event E and as $eValues(T)$ the list of values for the attributes of the event instances in T, where T is a trace of E in an event history H. We assume that the event service (ES), maintains an event history H_i for each obligation policy O_i in the system (i.e. $O_i \in Policies(\Delta, oblig)$). We introduce the following transition system for the runtime execution of obligation policies, which includes the event histories. The terminal configuration $fail$ is introduced later to denote a failure in the execution of obligation actions.

$Configuration_{oblig} ::= \langle Ponder\ term, state, store, \sum event\ history \rangle \cup \langle state, store, \sum event\ history \rangle \cup fail$
$\longrightarrow_o : Configuration_{oblig} \rightarrow Configuration_{oblig}$

Figure 5.7 Configurations and transition rules for obligation policies

The following notation is used throughout the semantics to refer to, and manipulate event histories:

$\sum H_i$ denotes the list of all event histories.

$(\sum H_i)[H_k \mapsto H'_k]$ denotes the fact that the history H_k is changed to a new history H'_k.

$\sum H'_i = (\sum H_i)[H_k \mapsto H'_k]$ denotes the updated history list.

$(\sum H_i)[H_i \mapsto H'_i]$ denotes the fact that each history H_i is updated to H'_i.

We can retrieve the event history associated with an obligation policy by calling: $ES(O_k)$.

$ES(O_k) = H_k = (\sum H_i)[k]$

An obligation policy object has the following elements:

$ObligObject ::= \ll subject = Dse, target = Dse, event = EventExpr, action = ObligAction, constraint = Expr, state = (true \cup false) \gg^{Type}$

5.5.2 Obligation Policy Execution

In this section we discuss the execution of an enabled obligation policy when an event occurs that triggers that policy. In the rules presented in the following discussion we use A to range over ObligAction and ba to range over BasicAction.

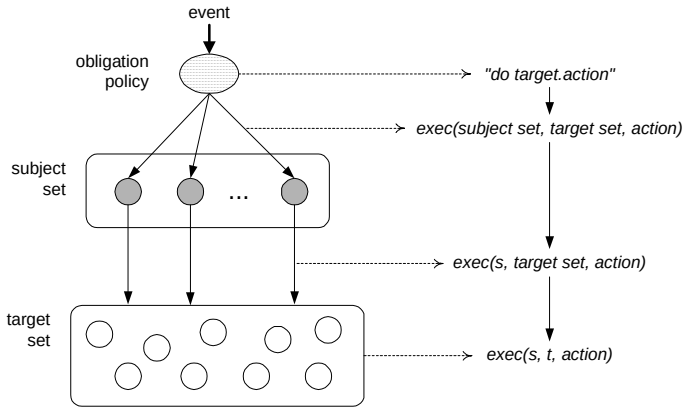


Figure 5.8 Obligation policy execution

Figure 5.8 illustrates the execution of an obligation policy, and more precisely the execution of a single target-based action in an obligation policy, i.e. we assume that the action element of the policy is: do target.ba. When an event occurs that triggers the policy, both the subject and the target sets of the policy are evaluated. The execution of an action of the form target.action then proceeds as follows: *All* subjects are asked to execute the action on *all* targets. For each subject, this translates to a series of action calls on each target. We use the different variations of the exec command to model action executions as described in Section 5.2.2. Thus, when the obligation is triggered and the subject and target sets are evaluated, the term do target.ba is executed as a command of the form $exec(set_s, set_t, ba, (v)^*)$ (i.e. the execution of the action ba on the set of target objects by the set of subjects) . This command is translated to a series of: $exec(s_i, set_t, ba, (v)^*)$ $\forall s_i \in set_s$ commands, each of which is in turn executed as a series of $exec(s, t_i,$

ba, $(v)^*$) $\forall t_i \in set_t$ commands. The rule (event exec target) shows the beginning of this execution, i.e. the evaluation of the term do target.ba. We use the occ function to determine if the current event history, which results from the addition of a newly occurring event, triggers the obligation policy. If it does, then the event history associated with the triggered obligation policy is emptied. Note that the state σ is updated with the values of the names that the event introduces, and so the evaluation of the constraint, subject and target, all happen using the updated state with access to those values. The execution of the action will also take place using the updated state (see the conclusion of the rule).

$$\frac{\begin{array}{l}\text{new event instance } e^i \text{ occurs} \\ \sum H'_i = (\sum H_i)[H_i \mapsto H_i + e^i] \\ pol \in Policies(\Delta, oblig) \\ pol(action) = target.ba \\ pol(state) = enabled \\ H_k = ES(pol) \\ occ(H_k, pol(event)) = true \text{ and } T_k \text{ is the trace of } pol(event) \text{ in the history } H_k \\ \sigma' = \sigma[x_i \mapsto val_i] \ \forall\ x_i \in eAttrs(pol(event)) \text{ and } \forall\ val_i \in eValues(T_k) \\ \langle pol(constraint), \sigma', \Delta\rangle \longrightarrow \langle true, \sigma'_1, \Delta\rangle \\ \langle pol(subject), \sigma'_1, \Delta\rangle \longrightarrow \langle set_s, \sigma'_2, \Delta\rangle \\ \langle pol(target), \sigma'_2, \Delta\rangle \longrightarrow \langle set_t, \sigma'_3, \Delta\rangle \\ \sum H''_i = (\sum H'_I)[H_k \mapsto \varepsilon] \\ <e_i, \sigma'_3, \Delta> \longrightarrow^* <v_i, \sigma'_4, \Delta>\end{array}}{\langle \text{do target.ba}(e_1 \ldots e_n), \sigma, \Delta, \sum H_i \rangle \longrightarrow_0 \langle \text{exec}(set_s, set_t, ba, v_1 \ldots v_n), \sigma'_4, \Delta, \sum H''_i\rangle}\ \text{(event exec target)}$$

The execution happens similarly when the action is subject-based i.e. for the term do subject.ba. In that case the term evaluates to an exec(set_s, set_s, ba, $(v)^*$) command that it is executed as a series of exec(s_i, s_i, ba, $(v)^*$). Each of these commands hides the details of executing the ba action internal to the subject (i.e. on the subject object itself) for every subject in the evaluated subject set of the policy. The rules which describe how execution proceeds for both subject and target based actions, as illustrated in Figure 5.8 can be found in Appendix C.

Failure of an action execution within the obligation context (a management component), means that the action was denied by the access control system as shown in rule (exec fail) or by a refrain policy; an action cannot be executed if there is a refrain that disallows the action execution as shown in rule (exec refrain fail). Success means the action was allowed as shown in rule (exec success). Note that we do not deal with failure of executing an action due to other reasons in this specification (e.g. network failures etc). We introduce an explicit error configuration called fail to model dynamic errors in the execution of actions.

$$\frac{\exists\ pol \in Policies(\Delta, refrain)\ (\ \langle disallows(pol, s, t, ba), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle true, \sigma, \Delta, \sum H\rangle)}{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 fail}\ \text{(exec refrain fail)}$$

$$\frac{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow deny}{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 fail} \quad \text{(exec fail)}$$

$$\frac{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow grant}{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle val,\ \sigma,\ \Delta,\ \sum H\rangle} \quad \text{(exec success)}$$

$$\frac{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle execFilter(s,\ t,\ ba,\ v_1 \ldots v_n,\ filterExpr),\ \sigma,\ \Delta\rangle}{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle execFilter(s,\ t,\ ba,\ v_1 \ldots v_n,\ filterExpr),\sigma,\ \Delta,\ \sum H\rangle} \quad \text{(exec success filter)}$$

The access control decision may return an `execFilter` command instead of a `grant` configuration, if the execution is granted but there is a filter that must be applied to the result of the execution (see Section 5.3.2). In that case the execution of an action evaluates to the execution of the `execFilter` command as shown in rule (exec success filter). The following two rules show the execution of the filter expressions i.e. the evaluation of `execFilter`. The filters are evaluated in order as described by the first of the rules (exec filter next). Execution stops at the first filter whose condition evaluates to true, and the return expression of that filter is applied to the value `val` that the normal execution returns, described in rule (exec filter). The resulting value `val'` is returned as the result of the filter execution.

$$\frac{filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n;\ result = expr_r\}\quad \langle expr_c,\ \sigma,\ \Delta\rangle \longrightarrow \langle false,\ \sigma_1,\ \Delta\rangle}{\langle exec(s,\ t,\ ba,\ v_1 \ldots v_n, filter_1 \ldots filter_k),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle exec(s,\ t,\ ba,\ v_1 \ldots v_n, filter_2 \ldots filter_k),\sigma_5,\ \Delta,\ \sum H\rangle} \quad \text{(exec filter next)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n;\ result = expr_r\} \\ \langle expr_c,\ \sigma_1,\ \Delta\rangle \longrightarrow \langle true,\ \sigma_2,\ \Delta\rangle \\ \langle exec(s,\ t,\ ba,\ v_1 \ldots v_n),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle val,\ \sigma,\ \Delta,\ \sum H\rangle \\ \sigma_3 = \sigma_2[z_1 \mapsto val] \\ \langle expr_r,\ \sigma_3,\ \Delta\rangle \longrightarrow^* \langle val',\ \sigma_4,\ \Delta\rangle \end{array}}{\langle execFilter(s,\ t,\ ba,\ v_1 \ldots v_n, filter_1 \ldots filter_k),\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle val', \sigma_4,\ \Delta,\ \sum H\rangle} \quad \text{(exec filter)}$$

Execution of Composite Actions

The execution of composite policy actions, and the semantics of the concurrency operators were given informally in Section 3.4.1 (see Table 3.3). The transition rules which formalise that description are presented in Appendix C. We omit them from this discussion for reasons of clarity.

Refrain Policies

A `disallows` command executed in the context of a policy management agent determines whether the given action execution is disallowed by a refrain policy, and should thus be filtered from the execution of an obligation policy, as demonstrated in the rule (exec refrain fail) described previously. The rules which show the execution of the `disallows` command for refrain policies are

very similar to those presented in Section 5.3.2 for the `allows` command in authorisation policies, although the transition system is different. For example the following rule is the corresponding of rule (allows false 5) and specifies that a refrain does not apply to the action execution if its constraint evaluates to false, forcing the execution of `disallows` to return false. For the complete set of rules, the interested reader can see Appendix C.

$$\frac{\begin{array}{l} r_1, r_2 \text{ are new in } \sigma \\ \sigma_1 = \sigma[r_1 \mapsto t] \\ \sigma_2 = \sigma_1[r_2 \mapsto s] \\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle false\ ,\ \sigma_2, \Delta\rangle \end{array}}{\langle disallows(pol, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle false, \sigma, \Delta, \sum H\rangle} \quad (refrain\ false\ 5)$$

5.6 Composite Policies

In this section we provide the semantics for the composite policy types by presenting the semantics for groups and roles; relationships and management structures are executed in a similar way. Basic policies (types and instances) inside a composite policy are stored in the directory entry representing that composite policy. Any path specification prefix to the name of the basic policy is thus discarded. We assume single inheritance for composite policy types with overriding of common policy instances and policy types. Policy instances with the same name and policy types with the same name and the same parameters, are overridden.

We define the following functions to enable the specification of the semantics:

- `createDomain(`Δ`, path/d)` creates a new domain `d` under `path`, within the given store Δ.
- `override(CompTypeBody,` $\texttt{CompTypeBody}_{\texttt{baseType}}$`)` overrides the definitions in a composite policy type body with those in the type of its base type and adds the two bodies together taking into account the inheritance rules described.
- `replacePath(CompTypeBody, path)` replaces the path which is prefix to the name of policies (types and instances) within the body of a composite policy , with the given path.
- `replaceSubject(compTypeBody, path)` replaces the subject element of policies inside the body of a composite policy, with the given path. This is used only for roles.

5.6.1 Groups

Composite (i.e. multiple) statements within a group are executed sequentially as described with rules: (program1), (program2). We use the same notation as before to represent composite policy objects; components of a composite policy type are policy type definitions and policy objects:

$$\texttt{CompObject} ::= \ll(\texttt{LabelName} = (\texttt{TypeDef} \cup \texttt{PolicyObject}))^{*} \gg^{\texttt{Type}}$$

A group type is stored in the store as is if it does not extend any other group type as shown in rule (type group). If a type extends another type, the bodies of the two types are combined, with all common elements (policies) in the extending type overriding those in the base type. This is described in rule (group type extends) which uses the `override` function to combine the two group bodies. The two rules apply to all composite policy structures.

$$\frac{\Delta_1 = \Delta[\text{path/t} \mapsto \text{type group path/t } (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ \text{Bgroup}\ \}]}{\langle \text{type group path/t } (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ \text{Bgroup}\ \}, \sigma, \Delta\rangle \longrightarrow \langle \sigma, \Delta_1\rangle} \quad \text{(type group)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ \Delta(\text{path}_1/t_1) = \text{type group path}_1/t_1\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ \text{Bgroup}_1\ \}\ \text{extends} \ldots \\ \langle e_i, \sigma, \Delta\rangle \longrightarrow^* \langle \text{val}_i, \sigma_1, \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto \text{val}_1]\ldots[z_n \mapsto \text{val}_n] \\ \text{Bgroup}_2 = \text{Bgroup}_1[z_1/x_1, \ldots, z_n/x_n] \\ \text{Bgroup}_3 = \text{override(Bgroup, Bgroup}_2\text{)} \\ \Delta_1 = \Delta[\text{path} \mapsto \text{type group path/t } (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ \text{Bgroup}_3\ \}] \end{array}}{\langle \text{type group path/t } (T_1\ x_1, \ldots, T_n\ x_n) \text{ extends path}_1/t_1(e_1, \ldots, e_n)\ \{\ \text{Bgroup}\ \} \longrightarrow \langle \sigma_2, \Delta_1\rangle} \quad \text{(group type extends)}$$

An instantiation of a group type replaces the values for the actual parameters inside the body of the group, creates a new domain entry for the group using the `createDomain` function, changes the domain paths for all policies inside the group body to the domain of the group using the `replacePath` function, and evaluates the body of the resulting group instance as indicated by the transition: $\langle \text{Bgroup}_2, \sigma_2, \Delta_1\rangle \longrightarrow^* \langle \sigma_3, \Delta_2\rangle$. This executes the statements in this body one at a time, and thus creates and stores policy instances and types within the new domain entry. The group instance is created and added to the state, but it is not placed in the store. The (inst group) rule also shows that the body of the group (the policies inside the group) are evaluated with respect to a new state resulting after all the parameters of the group have been added to the state: $\sigma_2 = \sigma_1[z_1 \mapsto \text{val}_1]\ldots[z_n \mapsto \text{val}_n]$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ \Delta(\text{path/t}) = \text{type group path/t } (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ \text{Bgroup}\ \} \\ \langle e_i, \sigma, \Delta\rangle \longrightarrow^* \langle \text{val}_i, \sigma_1, \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto \text{val}_1]\ldots[z_n \mapsto \text{val}_n] \\ \text{Bgroup}_1 = \text{Bgroup}[z_1/x_1, \ldots, z_n/x_n] \\ \text{Bgroup}_2 = \text{replacePath(Bgroup}_1\text{, path/g)} \\ \Delta_1 = \text{createDomain}(\Delta, \text{path}_1/g) \\ \langle \text{Bgroup}_2, \sigma_2, \Delta_1\rangle \longrightarrow^* \langle \sigma_3, \Delta_2\rangle \\ \sigma_4 = \sigma_3[r_1 \mapsto \ll \text{Bgroup}_2 \gg^{\text{path/t}}] \end{array}}{\langle \text{inst group path}_1/g = \text{path/t}(e_1, \ldots, e_n), \sigma, \Delta\rangle \longrightarrow \langle r_1, \sigma_4, \Delta_2\rangle} \quad \text{(inst group)}$$

5.6.2 Roles

A role type is stored in Δ in the same way this was described for groups. The instantiation rule for a role is also very similar to that described for a group and is given below (role inst). The only difference is that the subject of the policies inside the role is set to the subject domain of the role using the `replaceSubject` function.

z_i are new identifiers in σ
r_1 is new in σ
$\Delta(path/t) = type\ role\ path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Brole\ \}$
$\langle e_i, \sigma, \Delta\rangle \longrightarrow^{*} \langle val_i, \sigma_1, \Delta\rangle$
$\sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n]$
$Brole_1 = Brole[z_1/x_1, \ldots, z_n/x_n]$
$Brole_2 = replacePath(Brole_1, path/g)$
$Brole_3 = replaceSubject(Brole_1, path_2)$
$\Delta_1 = createDomain(\Delta, path_1/g)$
$\langle Brole_3, \sigma_2, \Delta_1\rangle \longrightarrow^{*} \langle \sigma_3, \Delta_2\rangle$
$\sigma_4 = \sigma_3[r_1 \mapsto \ll Brole_3 \gg^{path/t}]$

$\langle inst\ role\ path_1/g = path/t(e_1, \ldots, e_n)\ @path_2, \sigma, \Delta\rangle \longrightarrow \langle r_1, \sigma_4, \Delta_2\rangle$ (inst role)

5.7 Domain System Model

We use Alloy [Jackson 2000] to specify a model of the domain system, which can then be analysed using the alloy constraint analyser [Jackson et al. 2000] to demonstrate that it doesn't suffer from under- or over-constraint. We include the graphical representation of the alloy model below (Figure 5.9); the complete textual representation is included in Appendix C, with appropriate comments.

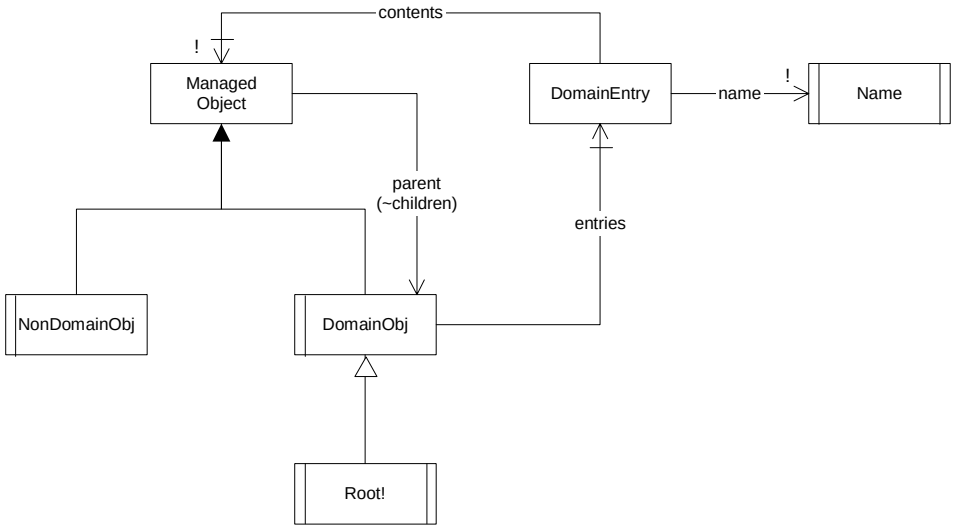


Figure 5.9 Graphical alloy domain-system model

The model consists of managed objects, which are either non-domain objects (`NonDomainObj`) or domain objects (`DomainObj`). The two classifications of the managed objects are *static* indicated with the vertical line on the left of their corresponding boxes. This means that an object cannot be both a domain and a non-domain object, preventing a non-domain object from becoming a domain object during its lifetime, and vice-versa. The filled arrowhead indicates that this is a *partition* of managed objects; they can only be either domain or non-domain objects. The root is a domain object, and there is only one root indicated with the exclamation mark next to the word *Root*. The root is a particular type of domain object which is *fixed* (indicated with the vertical lines on both sides of the root box).

Domain objects contain one or more domain entries (*DomainEntry*), which can be accessed using the *entries* relation on domain objects. Each domain entry has a unique name (assumed to be drawn from a fixed set of predefined names), as well as unique contents. The contents of a domain entry is a managed object and can be accessed using the *contents* relation.

The above diagram cannot express the operations part of the alloy specification, which is included in Appendix C. We specify the operations:

- `op NewDomainEntries (d: DomainObj, es: DomainEntry')`, to add a set of entries `es`, to a domain `d`.
- `op Create (d: DomainObj!, o: Object'!, n: Name)`, to create a new object `o` with name `n`, in a domain `d` and
- `op DeleteEntry (d: DomainObj!, n: Name)`, to delete an entry with name `n` from domain `d`.

We then use assertions to ensure that the execution of these operations does not leave the system in an inconsistent state (see Appendix C). Note that the operation of moving an object from one domain to the other can be performed as a sequence of create and delete operations.

5.8 Conclusions

In this chapter we have given a structural operational semantics to the policy specification language presented in Chapters 3 and 4. The semantics treats policies as objects with a predefined set of attributes corresponding to the elements of each policy type. We have presented the semantics by separating the execution of authorisation and delegation policies (i.e. the access control enforcement), from the execution of obligation and refrain policies (i.e. the subject-based policy enforcement). The operational semantics is a term rewrite system which maps configurations consisting of Ponder terms to new configurations, and shows the step-by-step execution of those terms. Policy specifications are evaluated in the context of a domain store, a state, and the list of event histories if the specification involves subject-based policies. We handle policy specification, instantiation and storage in the domain service, as well as runtime execution of policies. The semantics presented demonstrate the simple execution model dictated by the declarative nature of the policy language. Note that the current specification does not handle meta-policies and does not include a type inference system, both of which are left for future work.

Finally, we present a formal treatment of the domain system using Alloy, an object modelling notation. This includes operations on the domain system and proof of consistency for the domain operations.

Chapter 6

Policy Compiler

The main motivation behind the design of a common high-level language for different applications of policy-based management is the ability to uniformly specify policies, which can be enforced across multiple heterogeneous application domains and mechanisms. In this chapter we describe automated support for compiling policies into implementable specifications. We present a simple scenario of a backup and archiving system for a computer research institution, which we use to explain the design choices in this chapter and the next.

6.1 Scenario

In this scenario, we consider the management of a backup and archiving system for a research institution with many autonomous operating units corresponding to the departments (inspired from a scenario in [Hegering et al. 1999]). Certain services, including the backup and archiving servers, are available centrally for periodic backups (e.g., once a month) but departments may have their own backup servers for more frequent use (e.g., every evening). File servers are available for each department and it should be possible to specify backup strategies for individual users, based on their requirements (frequency, what data to backup, life of backed-up data).

It is important that the administration of managed data can be delegated on a hierarchical basis. Domains can be used to model a hierarchical structure of the system, which will also enable policy specification, and logically centralised system administration. Figure 6.1 shows a partial view of a domain structure for the system. The */staff* subtree contains the system administrators as well as research and development staff subdivided by division and department. The */staff/admin* subtrees are also subdivided by department (not shown in the figure due to space limitations), to reflect the fact that departmental administrators are responsible for the users of the corresponding department. The */system* subtree contains the resources (files and data) partitioned to reflect departmental structure, as well as the system servers (e.g. backup, account, mail etc). Finally, the */managementInfo* subtree is used to group policies and other management specific information. Only policy administrators (*/staff/admin/policy*) and security administrators (*/staff/admin/sec*) are permitted to access management information.

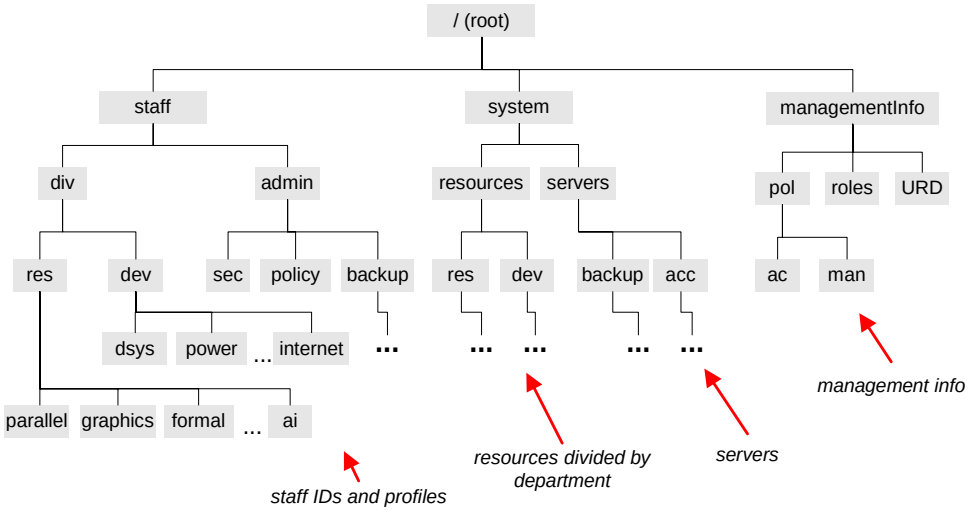


Figure 6.1 Research institution partial domain structure

Security is an important aspect in the system. Authorised users are able to access their own files, those shared in their department, as well as those to which they have been explicitly granted access. The account servers hold account information for each user in the system. Users are not normally allowed to execute actions that are carried out for them by the administrators (e.g., regular backups, setting of backup parameters). However, administrators can delegate those actions to the users they administer. The management system includes role-based management features. A security manager role is specified which defines the authorisation and obligation policies associated with an account security manager. Similarly a backup administrator role is defined to which backup administrators are assigned.

6.2 Design Choices

Ponder policies can be mapped to low-level representations suitable for enforcement by the security mechanisms of the underlying system or by the deployed management agents, or into formats appropriate for shipping and manipulating policies such as XML.

Obligation and refrain policies are enforced by automated management components or agents which act as subjects in the system, and thus the enforcement code for these policies must be suitable for interpretation or execution by distributed enforcement components. A very attractive solution is the use of Java code, which can be distributed across networks and executed on any system with a Java virtual machine. One alternative is to generate a Java program for each of the obligation and refrain policies specified, which can then be loaded into any agent for execution; an agent which is not Ponder policy-aware, can also be used in this case. The Java code will accept requests from administrators (possibly through the agent into which the code is loaded), and must provide interaction with the underlying event and domain services in order to enforce the policy.

The generated Java program will be used in the same way Java mobile code is used to extend the functionality of the agents in the system in a plug-and-play fashion. Although this approach is attractive, it has some disadvantages:

- Modifications to the interfaces of the underlying services will render the generated Java policy objects unusable because of the dependencies of the Java code on those services. The deployed policies will need to be retracted, recompiled and redistributed, which is a very expensive operation.
- Distributing policies as mobile code increases the security risks [Chess 1998].
- The size of the generated policy objects that need to be stored in policy repositories becomes large and might be a problem in large-scale management systems.

Code generation for authorisation policies is different because their implementation requires close interaction with the underlying access control mechanism. We expect to be able to map authorisation policies onto a variety of heterogeneous security platforms and mechanisms, such as firewalls, operating systems security, database security and Java authorisations. However, direct mapping to some of the available access control mechanisms is not possible because in most cases they do not support all of the features of Ponder authorisation policies. This means that authorisation policies will often be enforced by access control entities, which we will term access controllers in our architecture (see Chapter 7), and it thus becomes important to also distribute authorisation policies in a format suitable for interpretation by access controllers.

We choose to follow an approach that decouples the implementation of the policies in the runtime architecture from the code generated by the policy compiler, while still enabling easy interpretation of policy objects by distributed enforcement components. For each policy, the Ponder compiler generates a Java object which maintains a data-structure representation of the policy. In the following subsection, we define the interfaces for this standard representation of policies as runtime objects. The runtime representation reflects the analysis performed on the policy specification by the compiler, and makes it easier to evaluate the different elements of the policy at runtime.

6.2.1 Policies as Runtime Objects

A policy is implemented in the runtime system as an object; the policy compiler generates an object for each policy and stores it in the domain service. A policy object can then be accessed from the domain service and distributed to the enforcement components responsible for enforcing that policy, and it enables other objects and components in the system to query it to determine the subject, target and other elements of the policy. The representation of policies as runtime objects follows the class hierarchy presented in Figure 3.1. An annotated class diagram for the basic policy classes is shown below. The policy compiler creates an appropriate subclass from the class

hierarchy of Figure 6.2 for each of the user-defined policy types. A user defined positive authorisation policy for example, is compiled into a subclass of the *Auth+* class. This class can then be instantiated into a policy object using standard object-oriented techniques.

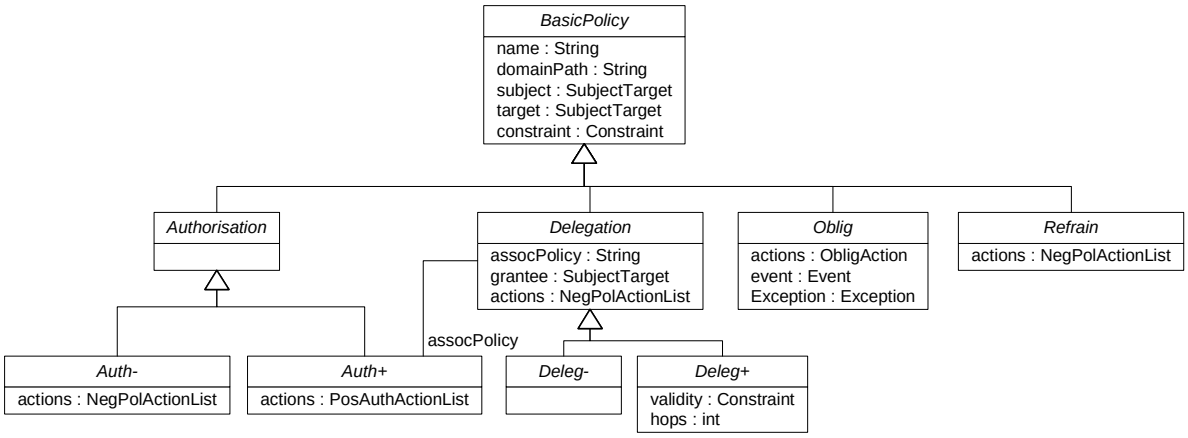


Figure 6.2 Runtime basic policy object classes

We outline the representation of the various elements for basic policies in the following subsections.

Domain Scope Expressions (DSE)

The *SubjectTarget* type (see Figure 6.2) is used to model subjects, targets as well as grantees in basic policies. A *SubjectTarget* type contains a string indicating the IDL-type of the specification if any, and also a DSE. A DSE (Figure 6.3) is represented as a tree where each non-leaf node is a *CompositeDse* and each leaf node is either a *SingleDse*, a *SingleAtDse* or a *SingleStarDse*. A leaf DSE node contains a reference to a *DomainObject*: *DomainObjectPath*, *DomainObjectCall* or *DomainObjectAttribute*.

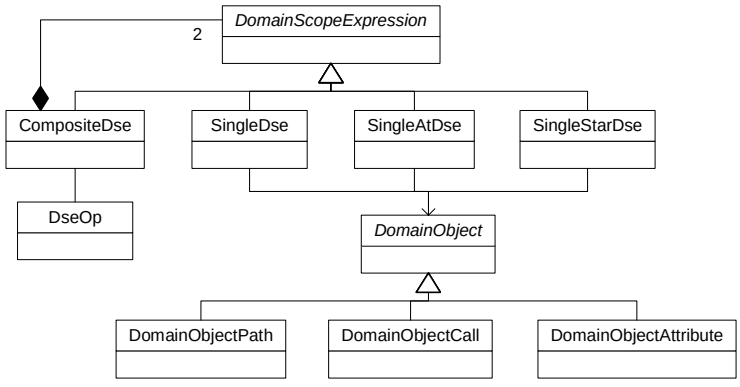


Figure 6.3 Domain scope expression class hierarchy

Here is what each of these classes models in a domain scope expression:

- A *SingleDse* models a DSE of the form: *DomainObject,* which represents a single domain
- A *SingleAtDse* models a DSE of the form: *@ [intValue] DomainObject,* which returns all non-domain objects *intValue*-levels down the domain hierarchy starting from *DomainObject*

- A *SingleStarDse* models a DSE of the form: * *[intValue] DomainObject,* which returns all objects *intValue*-levels down the domain hierarchy starting from *DomainObject*
- A *CompositeDse* models a DSE of the form: *DomainScopeExpr DseOp DomainScopeExpr*

Three subclasses of *DomainObject* are used to model simple domain objects:

- A *DomainObjectPath* models a path.
- A *DomainObjectCall* models a method call on a domain object, e.g. `d.get("e")`.
- A *DomainObjectAttribute* models an attribute (subject or target) of a policy object, e.g. `myAuthPolicy.subject`

The following is an example of a DSE and its runtime representation.

```
(@5 /a/b/c - /a/e) ^ (d.get(“e”))
```

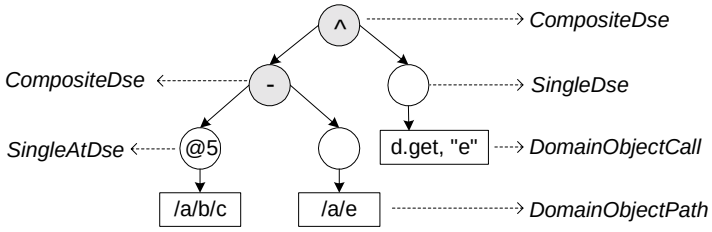


Figure 6.4 Domain scope expression runtime representation

Constraints

Constraints are divided into different classes (Figure 6.5), based on the part of the basic policy specification they apply to:

- A *SubjectConstraint* models a constraint on the subject of the policy.
- A *TargetConstraint* models a constraint on the target of the policy.
- An *EventConstraint* models a constraint on a parameter of an event of the policy (for obligation policies only).
- An *ActionConstraint* models a constraint on a parameter of an action of the policy (for non-obligation policies).
- A *TimeConstraint* models a constraint of the form: *Time.methodName(parameters)*.
- A *SystemConstraint* is used to model any other type of constraint. i.e. A call on the system to get state or other information.

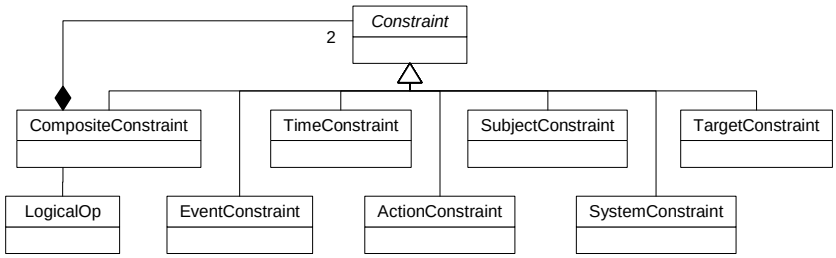


Figure 6.5 Constraint class hierarchy

The following is an example of a composite constraint in a positive authorisation policy. Note that the subject and target constraints actually contain relational expressions. In general, constraints are composed of expressions, which we describe at the end of this subsection.

```
inst auth+ myAuth {
        subject s = …;
        target  … ;
        action  a1(p1, p2), a2();
        when    (s.getStatus() == "idle") and (p1 < 5);
}
```

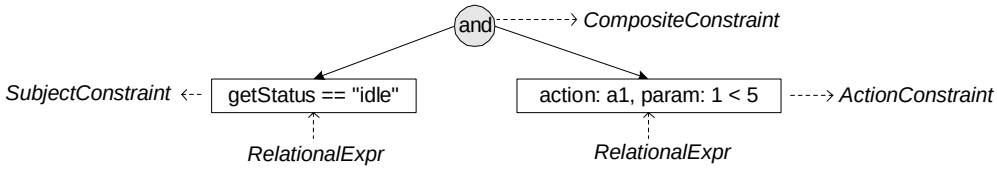


Figure 6.6 Constraint runtime representation

Events

The classes that are used to model events (Figure 6.7), include:

- A *TimeEvent* models an event of the form: *Event + time*
- A *MultipleEvent* models an event of the form: *n * Event*
- A *NoInterleaveEvent* models an event of the form: *Event1 ; Event2 ! Event3*
- A *SingleTimerEvent* models an event of the form: *Timer.methodCall(parameters)*.
- A *SingleNamedEvent* models a named event, e.g. `loginFailure(userId)`

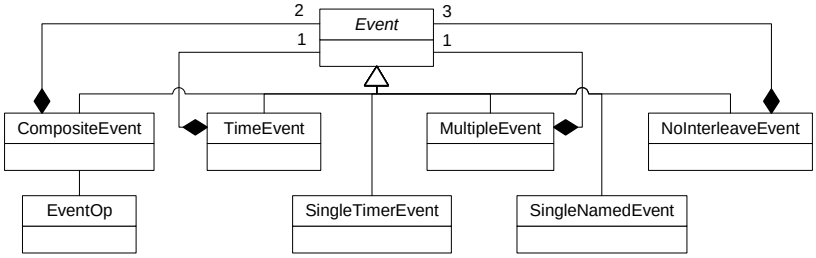


Figure 6.7 Event class hierarchy

A simple example of a composite sequential event is shown below:

```
((5*e1) -> (Timer.at("18:00:00") + 8))
```

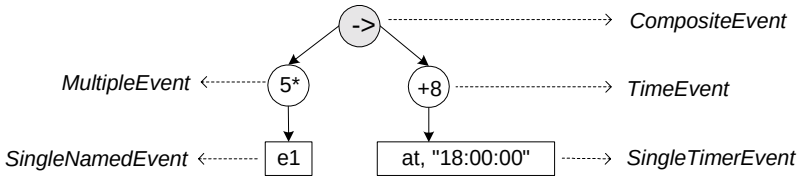


Figure 6.8 Events runtime representation

The above event specifies a sequential event composed of two events: The first is a *MultipleEvent* specifying 5 occurrences of event e1. The second is a timer event which occurs 8 secs after 6:00pm. The composite event is thus detected 8 secs after 6:00pm if e1 had occurred 5 times.

Obligation Policy Actions

Obligation actions are either composite or single actions (Figure 6.9). Single actions contain an indication of whether the action is to be executed on the subject or the target of the policy as shown in the example in Figure 6.10.

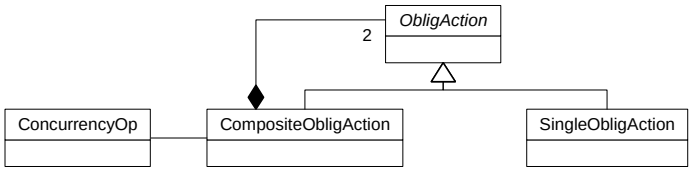


Figure 6.9 Obligation actions class hierarchy

Note that in the following example, the parameters to the action `t.a3()` are maintained within the *SingleObligAction* object, which represents that action at runtime. Each of the parameters is an instance of an expression.

```
inst oblig myOblig {
        on …
        subject s = …;
        target  t = … ;
        do      a1()| t.a2() -> t.a3(5, true);
}
```

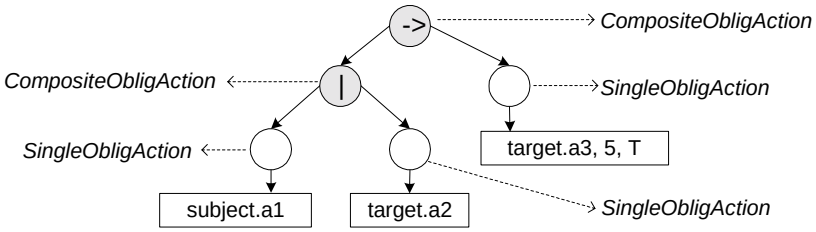


Figure 6.10 Obligation action runtime representation

Positive Authorisation Policy Actions

A positive authorisation policy contains a list (*PosAuthActionList*) of positive authorisation actions (*PosAuthAction*). Each of these actions contains a filter list (*FilterList*). Each *Filter* contains a condition (*true* if not specified), an expression for the result and a list of in-parameter filter expressions (*InFilterParameter*). Each of these objects contains the expression to be applied to the parameter, and the position of the parameter within the action call. The following is a simple example:

```
type auth+ filterLocationT (subject s, target t) {
        action lookup(x,y) if belongs(s, extUsers) {
                in x = 0;
                in y = Maths.abs(y);
                result = selectBuilding(result);      // external script call
        };
}
```

The filter attached to the lookup action is executed if the subject of the policy belongs in the group of external users (extUsers), and it restricts the values of the input parameters to the method call, as well as the result of the method call. The lookup method is called to return the location of a target object in an electronic-badge system. The first parameter specifies whether the system should update the location before sending the result, with 0 meaning no update when requested by an external-user. The second parameter is the id of the user requesting the information and the filter

expression just makes sure this is an integer value. The result is filtered using an external script to specify that the reply should be at the granularity of a building when called by external users.

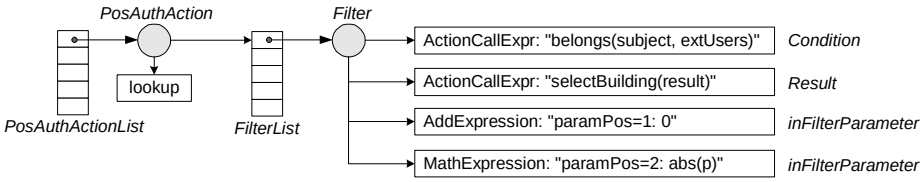


Figure 6.11 Positive authorisation actions runtime representation

For negative authorisation, refrain and delegation policies, the action specification is just a list of names modelled by a simple list data-structure. Exceptions in obligation policies are single action calls, which contain a reference to an action-call expression. Finally, we describe the runtime representation of expressions.

Expressions

The following classes are used to model expressions at runtime:

- An *ActionCallExpr* models an action-call on an object.
- An *ObjAttributeExpr* models an object attribute expression i.e. a reference to an attribute of an object in the system (including policy objects).
- A *FeatureCallExpr* models a feature-call expression, which is a method call on a collection (i.e. set) literal.
- A *MathExpr* models an expression of the form: *Math.methodCall(parameters)*, which is a call on the predefined math library object.
- A *LiteralExpr* is a (value, type) pair.

These expressions can also be composed using composite expressions (i.e. logical expression, a multiplication expression, a relational expression and an additive expression). The complete class hierarchy for expressions is shown in the following figure.

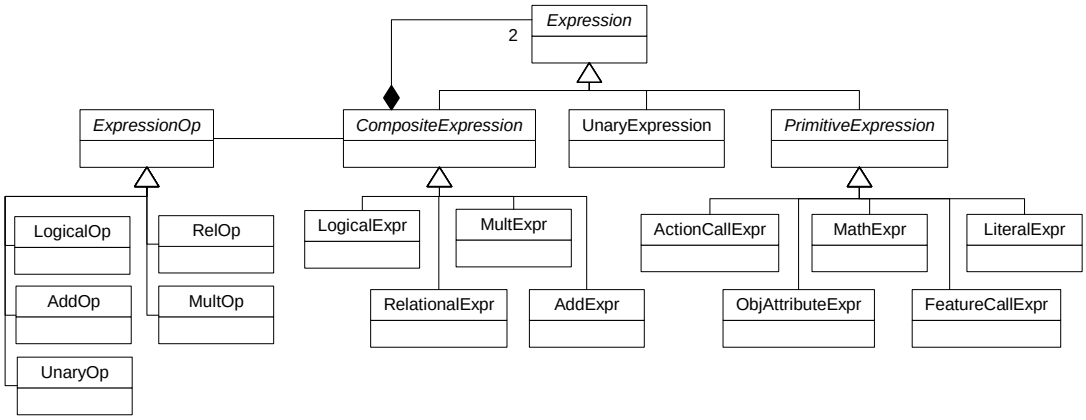


Figure 6.12 Expression class hierarchy

6.3 Compiler Design and Implementation

Dedicated code generators (compiler back-ends) must be implemented to translate the Ponder specification into the variety of runtime representations required. Figure 6.13 shows the main modules of the policy compiler framework. The main phases of the compiler generate intermediate code, which is then passed on to all the code-generators added to the compiler. The code assembler module is responsible for storing the generated code for a given policy in the domain service under the appropriate domain entry. Its implementation is thus specific to the domain service implementation. An entry for a policy can contain one or more representations for that policy. This is particularly true of authorisation policies. The default representation is the Java code, which is an implementation of the interfaces described in Section 6.2.1; the rest of the representations are termed auxiliary and can be accessed from the domain entry for a policy.

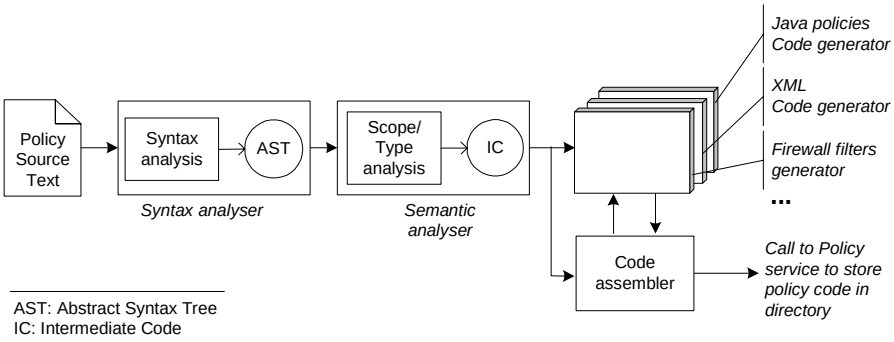


Figure 6.13 Compiler framework

The compiler is based on a LALR(1) parser generated with SableCC [Gagnon 1998], an object-oriented Java parser generator. The components of the compiler implementation are illustrated in Figure 6.14. The *SyntaxAnalyser*, generated with SableCC using the grammar of the language presented in Appendix B, parses a policy specification into an abstract syntax tree (AST) of Java objects. The AST is then passed on to the semantic analysers. Semantic analysis is performed in two passes. *SemanticAnalyserI* handles the definition of names (scope analysis) and checks that basic policies contain all the required elements (completeness checks). *SemanticAnalyserII* completes the scope analysis, performs type analysis, and generates an intermediate code (IC). The IC is a tree structure of Java objects corresponding to the composite and basic policies of the specification. It allows navigation of the specification to access the individual policies and policy elements, and implements the interfaces presented in Section 6.2.1.

The IC is passed on to the *CodeAssembler*, which is the component responsible for coordinating the code generation phase. The main module of the compiler maintains a list of code generators which implement a standard interface and can be dynamically added to the compiler in a plug-and-play fashion without recompiling the system The *CodeAssembler* passes the IC to all code generators

which are enabled by the user to generate code. The code generators call the *CodeAssembler* to store the generated code in the directory. The implementation of the *CodeAssembler* is specific to the underlying domain service implementation, and interfaces have been defined to allow customisation of the code assembler without the need to modify the rest of the components.

A *PolicyAnalyser* can also be added to the compiler. The *PolicyAnalyser* implements a predefined interface and receives a copy of the IC if policy analysis is selected. A *PolicyAnalyser* to perform modality conflict detection was implemented by a colleague at Imperial College and has been tested successfully with the current compiler implementation.

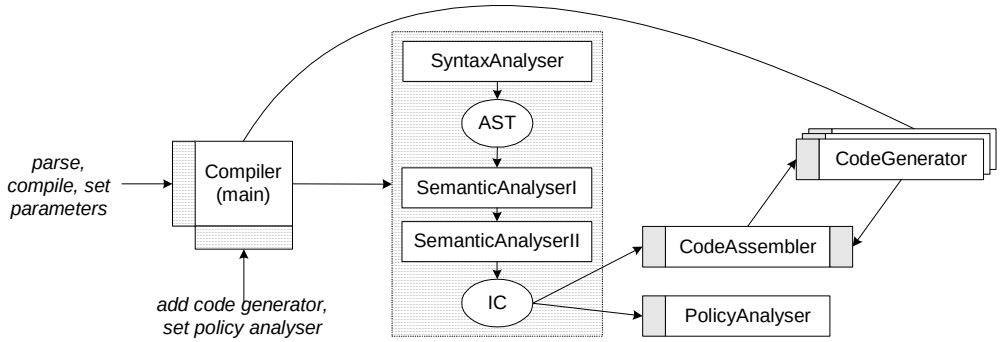


Figure 6.14 Compiler implementation

The main compiler module exposes two interfaces: A standard interface to set any compilation parameters and begin the parsing or compilation of a given policy specification, and an interface to add code generators and set the policy analyser. A Java code generator is included by default to generate a Java object representation of policies as defined in Section 6.2.1. Policy types are compiled into Java policy classes and stored in the domain hierarchy by the *CodeAssembler*. Instantiation of a basic policy type creates and initialises a Java policy object. Predefined classes exist which implement the basic functionality for each of the classes corresponding to basic policy types (see Figure 6.2). User-defined basic policy types are generated as Java classes which extend the appropriate basic policy class. For basic policies that are specified directly as instances, an unnamed Java class is generated in the background, and then instantiated. The following is an example of an obligation policy type from the example scenario, and its generated Java code.

```
type oblig spaceExceedT(subject backupAdmin) {
    on       userExceedQuota(uid);
    target   ms = /system/servers/mailServer;
    do       ms.notifyQuota(uid);
}
```

The above obligation policy requires that backup administrators (the subject of the policy which is a parameter to the policy type) send a notification to a user when that user's disk space quota is exceeded.

The Java code generator receives the intermediate code constructed by the main phases of the compiler for the above obligation, and generates the Java class shown below in Figure 6.15 as a subclass of the predefined *Obligation* class. The obligation class maintains the various policy elements and allows access to them. Since the objects for the various elements are already

constructed, the Java code generator simply serialises them and then reconstructs them when the *spaceExceedT* class is instantiated, so that they can be set in the super class.

```
import java.io.*;
import java.util.LinkedList;
import ponderToolkit.compiler.codeGen.policyObjects.*;
import ponderToolkit.compiler.codeGen.policyObjects.events.*;
import ponderToolkit.compiler.codeGen.policyObjects.constraints.*;

public class spaceExceedT extends Obligation implements Serializable {

    public spaceExceedT(String name, SubjectTarget backupAdmin)
    {
              // set the name and domain path of the policy
              super(name, "/managementInfo/pol/man");

              // any actual parameters to the instantiation are stored in a linked list
              LinkedList actualParams = new LinkedList();
              actualParams.add(backupAdmin);
              super.setActualParams(actualParams);

              // set the subject object
              super.setSubject(backupAdmin);

              // The target policy element
              // the bytes for the serialised target object
              byte[] targetBytes = { 0xffffffac, 0xffffffed, 0x0, 0x5, 0x73, ... };
              // reconstruct the target object
              SubjectTarget target = null;
              try {
                ByteArrayInputStream in = new ByteArrayInputStream(targetBytes);
                ObjectInputStream ois = new ObjectInputStream(in);
                target = (SubjectTarget)ois.readObject();
              } catch (IOException e) { System.out.println(e); }
                catch (ClassNotFoundException e) { System.out.println(e); }
              // set the target object
              super.setTarget(target);

              // The action policy element
              // the bytes for the serialised action object
              byte[] actionBytes = { 0xffffffac, 0xffffffed, 0x0, 0x5, 0x73, ... };
              // reconstruct the action object
              ObligAction action = null;
              try {
                ByteArrayInputStream in = new ByteArrayInputStream(actionBytes);
                ObjectInputStream ois = new ObjectInputStream(in);
                action = (ObligAction)ois.readObject();
              } catch (IOException e) { System.out.println(e); }
                catch (ClassNotFoundException e) { System.out.println(e); }
              // set the action object
              super.setActions(action);

              // The event policy element
              // the bytes for the serialised event object
              byte[] eventBytes = { 0xffffffac, 0xffffffed, 0x0, 0x5, 0x73 ... };
              // reconstruct the event object
              Event event = null;
              try {
                ByteArrayInputStream in = new ByteArrayInputStream(eventBytes);
                ObjectInputStream ois = new ObjectInputStream(in);
                event = (Event)ois.readObject();
              } catch (IOException e) { System.out.println(e); }
                catch (ClassNotFoundException e) { System.out.println(e); }
              // set the event object
              super.setEvent(event);
    }
}
```

Figure 6.15 Generated Java code snapshot

The enforcement of authorisation policies will probably require the mapping onto additional auxiliary representations. In our scenario, if servers used to store data in the AI research group are

Linux based while servers in other departments are Windows 2000 based, then appropriate access control code will be generated based on the type of server. The compiler has been evaluated in various projects, which resulted in preliminary implementations for translating Ponder policies onto various access control platforms. These include:

- A Java back-end to transform Ponder authorisation policies into access control policies for the Java platform. This has required several extensions to the Java security model in order to enable run-time Ponder policy evaluation, constraint checking and filtering [Corradi et al. 2001].
- Code generators for translating Ponder authorisation policies to Windows 2000 security templates and firewall rules.
- Experimentation with mapping Ponder authorisation policies to Linux access controls. System level scripts have been specified to program the Linux security kernel. A code generator translates Ponder policies into calls on those scripts.

The compiler handles the predefined libraries for *Timer*, *Time* and *domain* objects defined in Appendix B.

6.4 Policy Specification Support

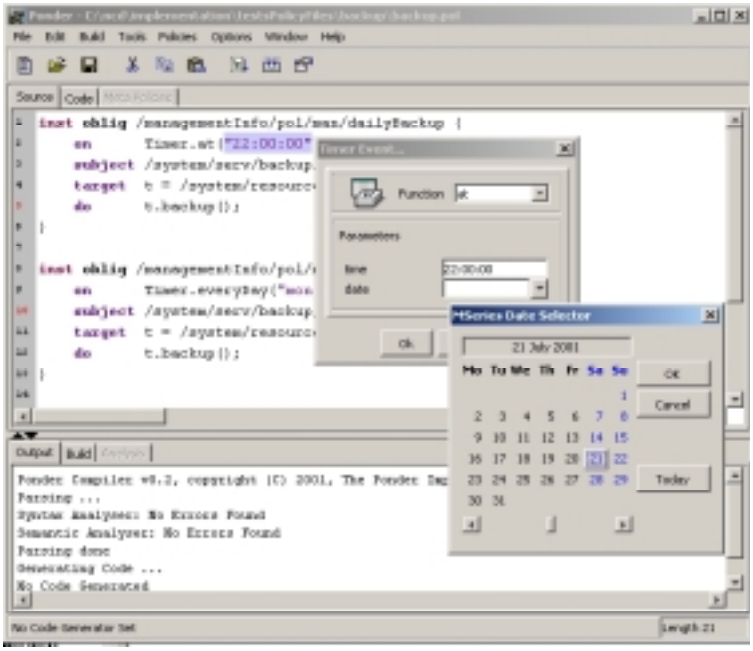


Figure 6.16 Policy editor

We have implemented an integrated development environment (IDE) for the specification of Ponder policies. The policy editor tool (Figure 6.16) is integrated with the policy compiler and provides an easy to use development environment for specifying, reviewing and modifying

policies. Templates can be used to create policies easily, and the domain browser can be invoked to select the subject and target domains for policies. Existing policies and policy types can be selected from the directory with the aid of the domain browser, loaded into the editor, modified, recompiled and stored back to the directory. Code generators added to the compiler framework, are accessible and can be enabled from within the editor to select the type of code to be generated.

Examples of policies that can be specified for the backup and archiving scenario include:

```
inst auth+ /managementInfo/pol/ac/userMgmtAC {
    subject   /staff/admin/sec;
    target    ds = /system/servers/domainServer;
    action    ds.addUser(), ds.deleteUser();
}

inst auth+ /managementInfo/pol/ac/userBackupMgmtAC {
    subject   /staff/admin/sec;
    target    bs = /system/servers/backup;
    action    bs.createUserAcc(), bs.deleteUserAcc(), bs.setPreferences();
}
```

The userMgmtAC policy authorises security administrators to add and delete users in the domain service. The userBackupMgmtAC authorises the security administrators to create and delete user accounts on the enterprise backup system used in the institution. It also allows the setting of backup and restore preferences for individual users.

```
domain /managementInfo/pol/ac/;

type auth+ backupLogsAcT(subject backupAdmin, set logData){
    target    t = /system/servers/backupLog;
    action    t.delete(data), t.read(data);
    when      data = logData;
}

inst auth+ backupLogsDev = backupLogsAcT(/staff/admin/backup/dev/,
                                         /system/resources/dev/);
```

The backupLogsAcT policy type authorises backup administrators to read and delete certain backup data from the backupLog servers. The backupLogsDev policy instantiated from this type, authorises the backup administrators of the development division to read and delete backup logs stored under the /system/resources/dev/ directory.

```
domain /managementInfo/pol/man/;

type oblig dailyBackupT(subject backupServer, set resources) {
    on  Timer.at("22:00:00");
    do  backupDaily(resources);
}

inst oblig dailyBackupInternet = dailyBackupT(/system/servers/backup/dev/internet,
                                              /system/resources/dev/internet);

type oblig weeklyBackupT(subject backupServer, set resources) {
    on  Timer.everyDayAt("fri", "*:*:*", "23:00:00");
    do  backupWeekly(resources);
}

inst oblig weeklyBackupParallel = weeklyBackupT(/system/servers/backup/res/parallel,
                                                /system/resources/res/parallel);
```

In the above examples, two obligation policy types are specified which refer to automated agents in the system, not human users. The dailyBackupT specifies that the backup servers must perform a daily backup of the given set of resources at 10:00pm every day. The weeklyBackupT specifies that the backup servers must perform a weekly backup of resources given as a parameter to the policy type. The backup takes place on Fridays at 11:00pm. The first instantiation statement creates an instance (dailyBackupInternet) of the dailyBackupT policy type. The internet department in the development division of the institution may require daily backups which are carried out by the corresponding backup server. On the other hand (weeklyBackupParallel) the research departments (the parallel department in this specific example) may require a backup of their data only on a weekly basis. Again the backup is performed by the appropriate backup servers.

The policy editor can make the specification of the above policies easy through template dialog boxes to create skeleton bodies for all of the policies, event dialogs to select the triggers for the obligation policies, and interaction with the domain browser to select the domain paths used in the specification.

6.5 Conclusions

In this chapter we described the design and implementation of a policy compiler to generate enforcement code for Ponder policies. The fact that the language is high-level and platform-independent imposes two requirements on the design of the compiler: (i) the ability to generate a variety of low-level representations of the policy based on the mechanisms that will be used to enforce the policy. This is more important for authorisation policies, which will be enforced by mapping the policies to the target's security mechanisms (ii) the design of a runtime representation for policies to enable distribution to the enforcement components, and make the interpretation of policies easy.

The compiler is implemented in Java and satisfies the first requirement by allowing addition of customised back-ends to generate code based on the application. A default Java code generator is included to generate a Java object for each policy in order to satisfy the second requirement. We have described the interfaces which are implemented by the generated Java code to enable easy interpretation of the policy objects by the enforcement components. The compiler has been integrated with a policy editor that simplifies policy specification through policy templates and interaction with the domain browser. In the following chapter we will show how the code generated by the compiler is used to deploy policies. We present performance results for the compiler implementation in Chapter 8 where we critically analyse the policy framework.

Chapter 7

Policy Management Platform

In the previous chapters we have described both in-formally and formally the proposed policy specification language. In this chapter we complete the description of our policy-based management framework by introducing an architecture for deploying policies. The deployment architecture supports the instantiation, distribution and life-cycle management of policies, as well as their enforcement by automated enforcement components. We assume an object-oriented view of the underlying distributed system where interaction occurs through remote object invocations and asynchronous event notifications. An overview of the architecture can be found in [Dulay et al. 2001a]. We present an implementation of the architecture and an integrated policy-administration toolkit. Examples are used from the scenario described in Section 6.1 to make the discussion clearer.

7.1 Deployment Model Overview

Figure 7.1 illustrates the architecture of the management system. It includes three supporting services: a domain service, a policy service and an event service. The *policy service* acts as the interface to policy management; it stores compiled policy classes, creates and distributes new policy objects. The d*omain service* manages a distributed hierarchy of domain objects and supports the efficient evaluation of subject and target sets at run-time. Each domain object holds references to its managed objects but also references to the policy objects that currently apply to the domain. In concept, the domain service is similar to a directory service such as LDAP, with extensions to allow changes to the membership of a directory to be distributed to interested parties, e.g. via events. The domain service can also be implemented by database systems. The e*vent service* collects and composes system events as well as those from the managed objects in the system, and forwards them to registered policy management components to trigger obligation policies. We call the objects that generate events, event publishers in the figure. The event service exposes a publish/subscribe interface whereby clients can subscribe to receive certain types of events.

The user (i.e. policy administrator) interacts with the policy and domain services to design the domain structure, specify new policies, compile and store them in the domain service. Management

tools interface with the two services and provide a graphical environment for specifying and managing policies. The distribution of policies is initiated through the policy service. Policy control objects (PCOs) are generated and maintained by the policy service to manage policies at runtime. This includes the distribution of the policies to their enforcement components, and the handling of dynamic domain-membership changes to the objects to which the policies apply. The fact that Ponder policies explicitly define their subjects and targets makes the automated distribution of policies possible. Obligation and refrain policies are distributed to their subjects and authorisation policies to their targets by the corresponding PCOs.

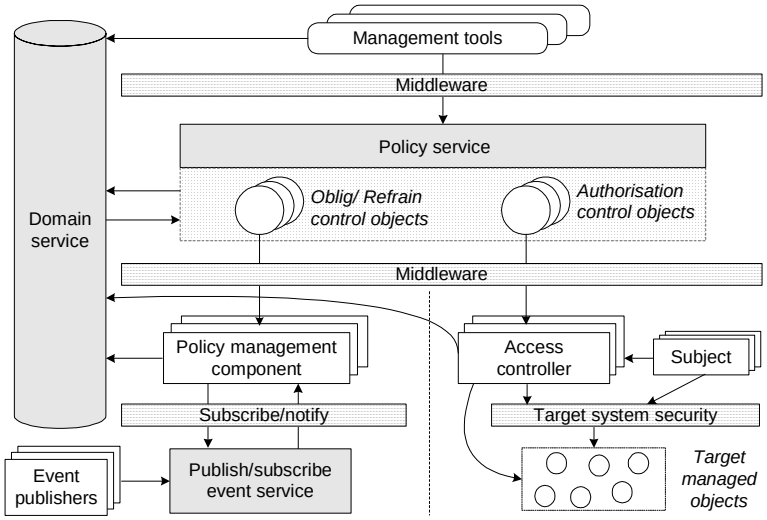


Figure 7.1 Management system architecture

The entities that enforce policies are called enforcement components and are potentially distributed. Enforcement components implement a policy enforcement interface to support loading, unloading, enabling and disabling of policy objects disseminated from PCOs.

Authorisation policy enforcement is delegated to one or more enforcement components that intercept actions on target objects and perform checks on whether the access is permitted. In our model, the enforcement components for authorisation policies are termed *access controllers* (ACs) and typically interface to lower-level access control mechanisms that really carry out the access control; for example a firewall protecting the services on its network, an operating system protecting its resources, or a database manager protecting its databases. An access controller will normally protect all the targets at its location and enforce all authorisation policies relating to them. The subjects of an authorisation policy can be any objects that initiate invocations.

The subjects of refrain and obligation policies are instances of special enforcement components called *policy management components* (PMCs) whose behaviour is defined by the refrain and obligation policies that apply to them (or the real-world entity that they represent). Policy management components thus directly enforce all the refrain and obligation policies for a subject,

and their architecture can be generic, although they can be targeted to specific applications e.g. QoS, security or storage management. The enforcement of obligation policies requires a PMC to subscribe the event specifications of its enabled obligation policies with the event service. The event service notifies the PMC of events which trigger its obligation policies, and the PMC is then responsible for executing the actions defined by those policies if the constraints are valid and if no refrain policies apply to those executions. Refrains are similar to negative authorisations but are enforced at the subject by the PMC and act as filters to the actions that the subject invokes.

7.2 Policy Administration Toolkit

The overview of the Ponder architecture implementation is shown in Figure 7.2. The implementation uses LDAP [Wahl et al. 1997a] to realise the domain service, and Java RMI [Sun 1999c] as the middleware for communication between the various system components. All tools are implemented in Java, and Swing is used for the graphical user interfaces. Policy management components and access controllers are implemented as Java RMI remote objects. The policy and domain services are also implemented using Java RMI. Policy control objects are remote objects as well, and are also bound to the LDAP directory for persistence. The execution of policy life-cycle operations occurs through direct interaction with the RMI policy control objects. The Java Naming and Directory Interface (JNDI) [Sun 1999d] is an API that provides directory and naming functionality to Java applications, and is used as the interface to the LDAP directory. For the current implementation of the event service we use Elvin [Segall et al. 1997; Arnold et al. 2001], a publish-subscribe messaging system that implements content-based event notification.

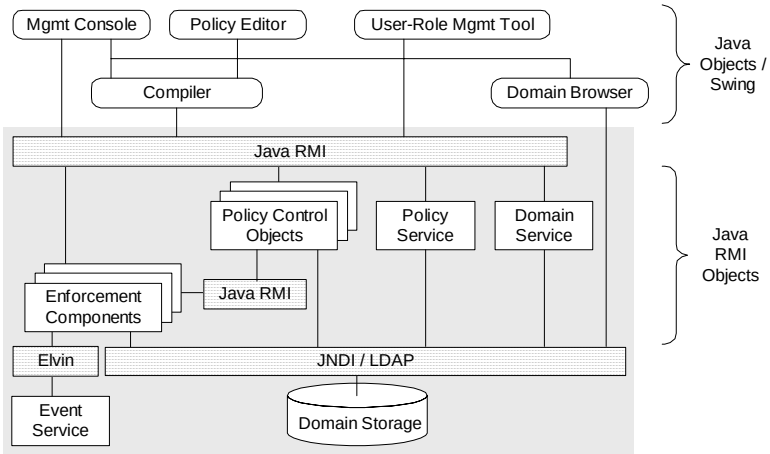


Figure 7.2 Management system architecture implementation

The domain browser, which is implemented by a colleague [Tonouchi 2001] in a related project at Imperial College (see Section 2.5.1), provides a common user interface for all management interaction with objects stored in the domain service. Other tools interact with the domain browser

to select objects from the domain service. Figure 7.3 shows the steps involved in managing a policy-based system. Policies and roles are created using the *policy editor* (see Section 6.4), compiled and stored in the domain service. The *management console* and *user-role management* tools can then be used to distribute policies and to manage roles (activation /deactivation and user-role assignment). The tools can be used in a distributed manner by a number of administrators.

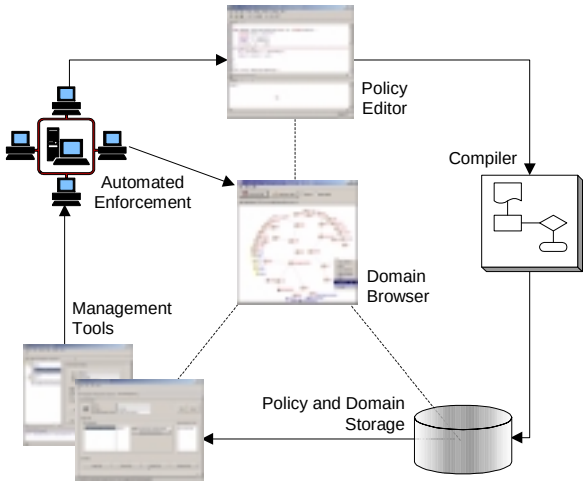


Figure 7.3 Policy management cycle

The domain service is implemented using an LDAP version 3 directory [Wahl et al. 1997b] with extensions to allow objects to be members of multiple domains. We define an LDAP schema based on the object class-hierarchy of the language (Appendix A), which extends the schema for representing Java objects as defined in [Ryan et al. 1999]; we add an *object reference* LDAP class to represent multiple parents (i.e. directed acyclic graphs). Note that we use LDAP for both storing policies and for grouping subject/target objects whereas the IETF framework [Moore et al. 2001] uses directories only as policy repositories.

7.2.1 Main Console

The GUI component of the toolkit consists of a *main console* for accessing the individual tools, with an implementation which allows easy customisation and addition of new tools. The operation of all the tools requires connection with the policy and domain services. The configuration parameters can be specified using a *configuration manager* tool, and shared by all the tools in the system. Snapshots of the main console and the configuration management tool are shown in Figure 7.4.

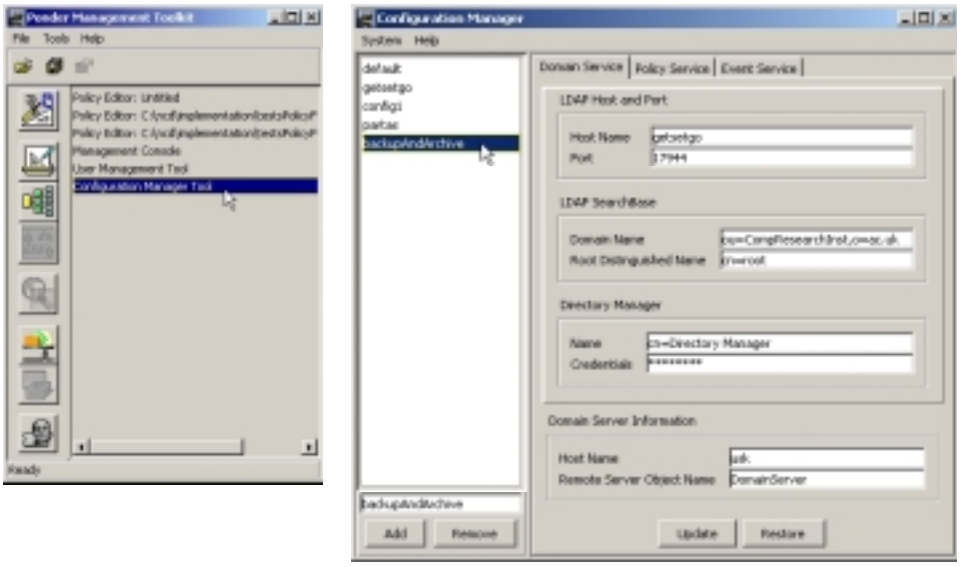


Figure 7.4 Toolkit main console and configuration tool

Customising the Toolkit

We have defined interfaces to enable the addition of new tools to the main console. We illustrate the interaction of tools with the main console and the role of these interfaces in Figure 7.5. A tool needs to implement the *PonderTool* interface in order to enable the main console (the *ToolOwner*) to invoke these tools and display information about them.

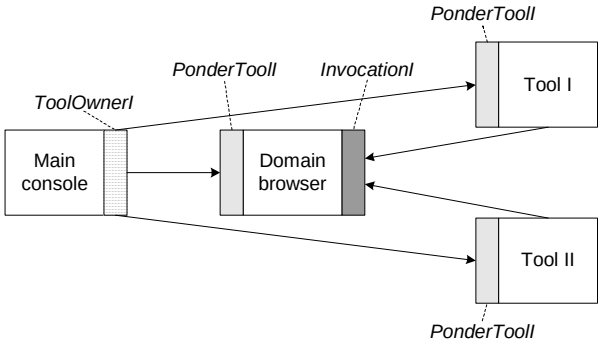


Figure 7.5 Tool implementation interfaces

A domain browser is a key component for all aspects of an integrated management environment. External tools can be invoked from within the domain browser for a specific managed resource or policy, depending on the current selected context, or interface with the domain browser to allow for navigation or selection of objects from the domain service. For example, the policy editor (see Section 6.4) allows the specification of a policy's subject and target domains by selecting them from the domain browser. In order to enable this kind of interaction, an *invocation* interface is specified for the domain browser, so that other tools can interface with it. The *PonderTool* and the *Invocation* interfaces are both implemented by the domain browser [Tonouchi 2001] described in Section 2.5.1. Existing LDAP browsers can also be used as long as they are extended with an implementation of the interfaces as shown in the above figure.

7.3 Policy Distribution

A key aspect of the architecture is the automation of the distribution of policies to their enforcement components without the need to manually manage the associations between the policy objects and the enforcement components. These associations are explicitly defined within policies as subject and target DSEs, and thus the enforcement components responsible for implementing the policies can be identified at runtime. For obligation and refrain policies, this is the set of policy management components in the subject scope of the policy. For authorisation policies, this is the set of access controllers responsible for enforcing access control for the set of managed objects in the target scope of the policy.

Figure 7.6 illustrates the steps involved in deploying a policy instance, and shows the interactions between the components of the architecture. The policy administrator interacts with the policy service to specify a new policy, which is compiled to a policy object and stored in the domain repository by the policy service. The administrator can select stored policy objects again by interacting with the policy service. The policy service accesses the domain repository and creates a corresponding policy control object (PCO) for the selected policy to coordinate run-time access and dissemination of the policy; an authorisation control object (ACO) for authorisation policies, a refrain control object (RCO) for refrains and an obligation control object (OCO) for obligations.

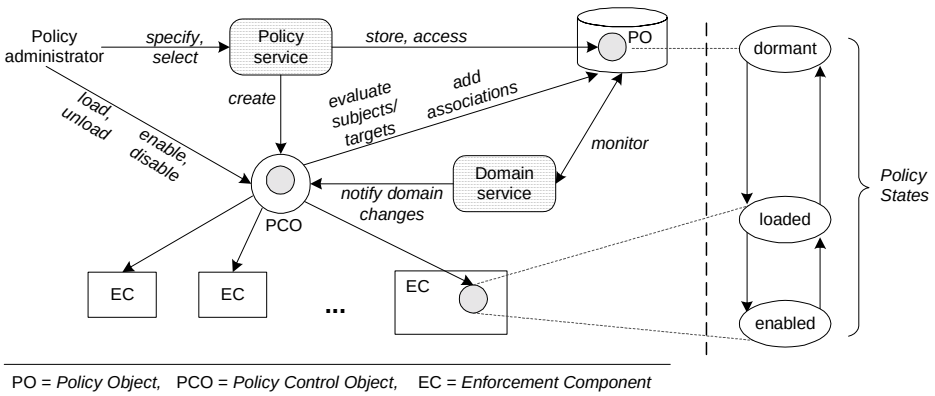


Figure 7.6 Policy deployment steps

A policy object can be loaded into its enforcement components, and once loaded, it can be enabled, disabled or unloaded from its enforcement components (see Figure 7.6 for the possible policy states). Unloaded (i.e. dormant) policies can be either re-loaded or deleted. The policy control object co-ordinates these life-cycle policy operations, and acts as a centralised control point for managing concurrent and possibly conflicting requests from multiple policy administrators. Note that replication of policy control objects is possible, to enhance scalability.

The administrator interacts with the corresponding PCO to manage a policy. When the administrator executes a load request, the PCO evaluates the set of objects that the policy applies to

– the subject-set for refrain and obligation policies, the target-set for authorisation policies, and distributes the corresponding policy object to all enforcement components which are responsible for its enforcement. Enable, disable and unload operations are similarly forwarded to all enforcement components by the PCO. In addition to distributing the policy object, the PCO associates the policy with the domain objects to which the policy applies. This enables the domain service to notify the PCO when changes occur to these domains in order to automate the process of loading the policies on newly added enforcement components, or remove the policy from those enforcement components to which the policy no longer applies. We will describe this process in more detail in Section 7.3.2.

An enabled policy object becomes active and runs within each of the enforcement components to which the policy is distributed. Details of the execution of obligation and refrain policies are described in Section 7.4. Authorisation policies are distributed in the same way to access controllers, and we discuss the enforcement of access control in more detail in Section 7.5.

7.3.1 Management Console Tool

We have implemented a management console tool (Figure 7.7) for dynamically managing policies, which interfaces with a domain browser to select policies and enforcement components from the directory, as well as with the policy compiler to interactively instantiate policies.

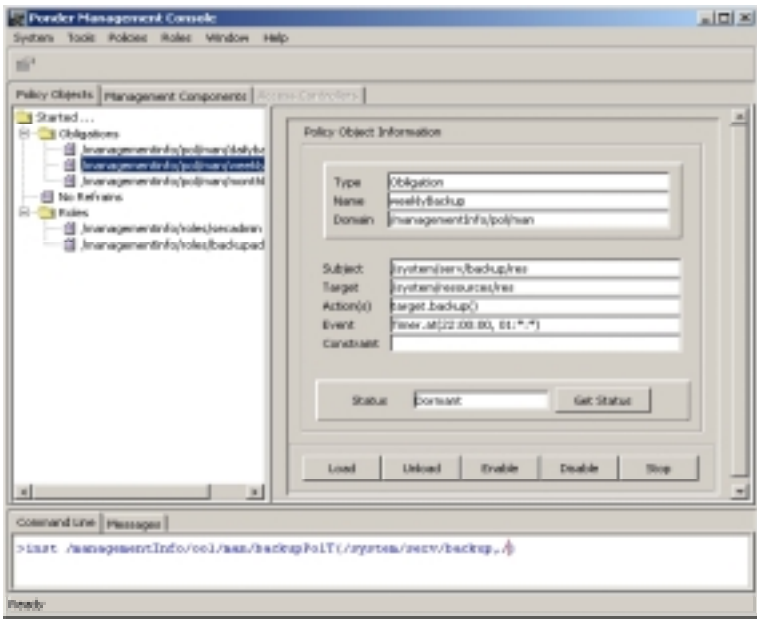


Figure 7.7 Management console tool

The steps involved in using the tool are demonstrated in Figure 7.8. The tool has two main views:

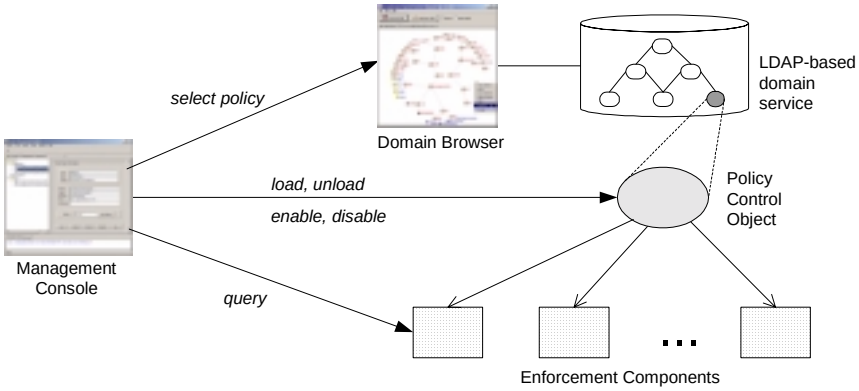


Figure 7.8 Managing the policy life-cycle

In the *Policy Objects View*, a policy instance can be selected from the domain service (using the domain browser) and loaded into the management console. Similarly, if a domain is selected all policy instances under that domain will be loaded into the management console in an expandable tree-navigator. Policies can then be loaded, unloaded, enabled or disabled as needed, by transparently interacting with the PCOs. Details about the selected policy and its status are displayed. Note that groups are managed in the same way as domains. A group can be selected from the directory to get access to all policies inside that group. Policies are then managed on an individual basis, as groups are merely syntactic structures.

In the *Management Components View*, policy management components can be selected from the domain service using the domain browser and information about the policies loaded into them is displayed in a tabular format. This includes policy information and the status of each of these policies. A *Command-line Window* is included and can be used to type single-line commands to the policy compiler. This allows interactive instantiation of policy types.

Note that multiple management consoles could be used to manage the same domain of policy objects in a distributed fashion. Although LDAP implementations do not support concurrency control, this is not a problem in the case of management consoles, as they are not modifying the contents of the LDAP server. More than one management console can be accessing the same PCOs, and concurrent requests are handled at the level of the policy control RMI object.

7.3.2 Domain Membership Changes

Domains are not static and when an object is added to a domain, deployed policies applying to that domain must automatically apply to the new object. Similarly policies must cease to apply to objects removed from domains to which the policies refer. Consider the following example from the archiving scenario where the *selfDiskCheck* obligation policy applies to backup servers in the research division. The servers must perform a self-check of available disk space on the 1st of each month. When a new server that is installed in the system is added under the */system/serv/backup/res*

domain, this policy must be automatically loaded and enabled on the server. The domain service detects the addition of the new server, and notifies the PCO of the *selfDiskCheck* policy. The PCO then loads and enables the *selfDiskCheck* policy on the new server. The same is true for all policies applying to that domain. Note that the backup servers in our scenario act as subjects for obligation policies and must thus implement the PMC interface.

```
inst oblig selfDiskCheck {
    on       Timer.at("23:00:00", "01:*:*");
    subject  backServ = /system/servers/backup/res;
    do       backServ.checkDisk();
}
```

Similarly, the two authorisations below must be automatically loaded in the access controller responsible for protecting the newly added backup server.

```
inst auth+ backupServAdmin {
    subject   /staff/admin/backup;
    target    t = /system/servers/backup;
    action    t.startBackup, t.startRestore, t.config, t.jobDelete, t.jobDetails;
}

inst auth+ backupShutdown {
    subject   /staff/admin/sec;
    target    t = /system/servers/backup;
    action    t.shutdown;
}
```

The above two positive authorisation policies specify access rights to backup servers for security and backup administrators respectively. The security administrators are authorised only to shutdown the servers, whereas the backup administrators have access rights relating to the configuration, backup, restore and job management operations.

In order to enable the above process, any policies that are in a loaded or enabled state need to be informed of changes to the memberships of domains to which the policy refers. The domain service detects changes in domain membership and notifies the PCOs affected by those changes. The domain service includes three monitoring components, one for each of the different types of objects that can be added or removed from the domain service, to distribute the load of monitoring and notifying these changes as illustrated in Figure 7.9.

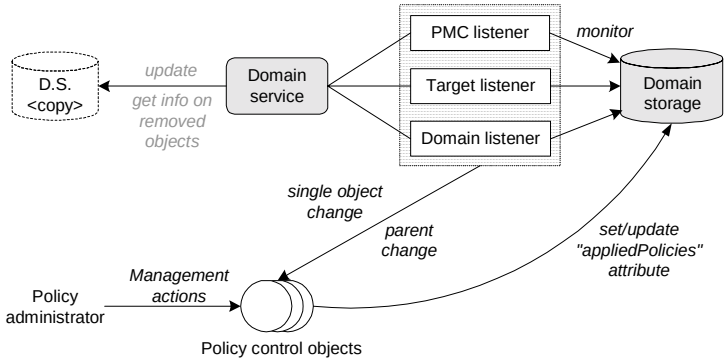


Figure 7.9 Domain membership monitoring

PMC listeners monitor changes that involve policy management components, *target listeners* monitor changes that involve any type of object apart from PMC's and domain objects, and *domain listeners* monitor those changes that involve domains, including additions or removals of multiple

parents to an object. Obligation and refrain policies are interested only in changes involving the subject domains of the policy thus they receive notifications from PMC-listeners and domain-listeners. Authorisation policies are interested in changes that involve the target domains and thus receive notifications from target-listeners and domain-listeners. Note that the implementation of the domain service and its listeners can be distributed.

The whole process is based on an attribute added to the domain object entries, which we call *appliedPolicie*s. This attribute contains the list of references to policies which are affected by changes on that domain and are thus interested to be informed of any such changes affecting the domain. The PCO is responsible for updating the *appliedPolicies* attribute when it receives load and unload requests from policy administrators. When an obligation policy is loaded, the corresponding OCO evaluates the list of domains in the scope of the subject DSE of the obligation policy, and changes the *appliedPolicies* attribute of all those domains to include the obligation policy. Inversely, the unload method of a OCO removes the policy from the *appliedPolicies* attribute of the subject domains. The same happens for authorisations only the domains involved are those in the scope of the target DSE of the policy. Change notifications received by a PCO from the domain listeners may also require updating the *appliedPolicies* for domains, if these notifications involve the addition or removal of parent domains.

We identify two different types of changes that affect policy control objects. The first one involves the addition or removal of a single non-domain object from a domain. The second (termed *parentChange* in Figure 7.9) involves the addition or removal of a new parent to an object. A single object change is notified by a PMC-listener for obligation policies, or a target-listener for authorisation policies. The parent change is notified by a domain-listener. We describe the two cases for obligation policies; a similar algorithm is used for authorisations.

The single object change is easy: PMC-listeners identify the domain in which the PMC was added, or from which the PMC was removed, and send a notification to all policy control objects included in the list of *appliedPolicies* of that domain. The notification includes the object (a reference to the PMC) which was added or removed and whether this was an addition or a removal. If it was an addition, then the OCO will simply load (and enable if necessary) the obligation policy on the new PMC. If it was a removal, then the OCO will remove the policy (disable and unload) from the PMC.

Parent changes are more complicated and may result from the addition or removal of a new parent to an object. Note that domains can overlap and both objects and sub-domains can be members of multiple domains. Thus, changes in the membership of a domain do not always result in a change to the set of objects to which a policy applies, e.g. a domain may be used in both a set union operation and a set difference operation. The following example demonstrates the issues involved.

Consider the domain structure of Figure 7.10 and an obligation policy whose subject is: `/A - /D`. Before the addition of the link shown in domain structure (b), the policy applies to the PMC object x and is thus loaded in x. When the new link is added as shown in (b), the PMC is no longer in the scope of the obligation policy and the policy must be unloaded from x. The exact opposite would happen if the link was to be removed to go from domain structure (b) to (a). In addition, the situation would be different if the subject domain was `/A ^ /D`; in that case, the PMC x is not in the scope of the subject DSE before the addition of the link (domain structure a), and it becomes part of the subject set after the addition of the link from domain E to domain C (domain structure b).

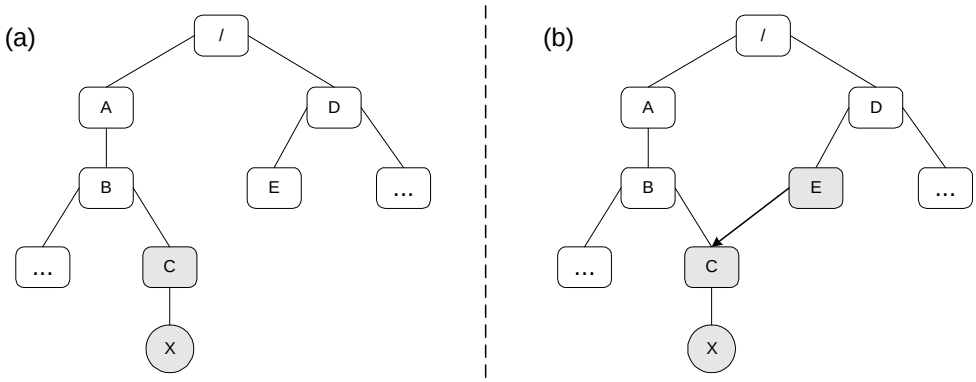


Figure 7.10 Domain parent changes

All of the above cases are handled by the OCO when it receives a *parentChange* notification: the OCO re-evaluates the subject DSE, identifies any objects added (or removed) by comparing the new set with the old one (before the change was received), and accordingly proceeds to load/enable or disable/unload the obligation policy from the set of PMCs resulting from the comparison. The question is: when does a policy receive a *parentChange* notification? The domain-listener sends a *parentChange* notification only if the policy appears in the *appliedPolicies* list of the domain which receives the new parent (in our example, domain C). It sends a *singleObjectChange* notification if the policy appears only in the *appliedPolicies* list of the domain which becomes the new parent (domain E). The same happens when links are removed.

We use JNDI naming listeners to implement the domain service listeners. A JNDI listener registers interest in certain events using filters to restrict the change notifications it receives, to only those for which it is responsible. The listeners then process the events and access LDAP to get information on the policies affected by these events. Finally, they request references to the policy control objects corresponding to those policies and notify them as appropriate.

Other Issues

When a new domain is added, it inherits the *appliedPolicies* attribute of its parent domain. However, some of the policies in the list of *appliedPolicies* of the parent, may not be candidates for inheritance if the DSE, which caused their addition, restricts the propagation. To handle this case,

the entries of the *appliedPolicies* attribute must include the policy object reference together with a number indicating the number of levels it can propagate down the domain hierarchy.

Figure 7.9 includes a *copy* domain storage maintained by the domain service. If the domain repository implementation does not provide the ability to get information on removed objects, the domain service will need to maintain a minimal copy of the domain storage including only information that may be required after the removal of certain types of objects. For example, if a managed object is removed from the domain service, the ACOs of the authorisations that applied to that object will need to notify the access controller responsible for the object about the change. In order to do so they will need to get information about the object, which will only be accessible from the copy domain storage.

In the following section we describe the architecture of the policy management component and explain the issues involved in enforcing subject-based policies.

7.4 Policy Management Component

Policy management components enforce all the enabled refrain and obligation policies for a subject. PMCs can act as proxies for the user to allow access to resources permitted by the authorisation policies or they can act as automated agents, interpreting obligation policies on the user's behalf. PMCs are created and stored in the directory in domains which are used to specify the subjects for obligation and refrain policies. An overview of the operation of a PMC is shown in Figure 7.11.

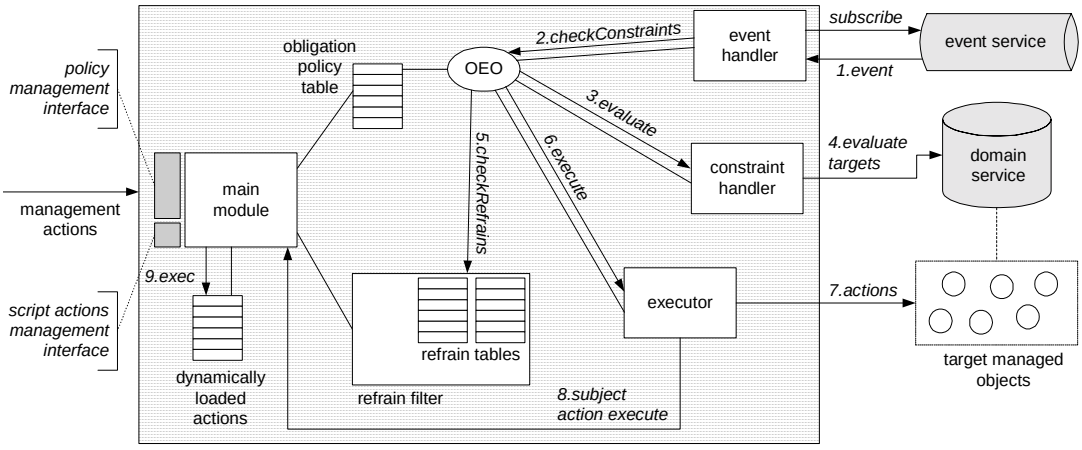


Figure 7.11 Policy management component

The obligation and refrain policy objects are loaded from corresponding policy control objects (OCO, RCO) using the policy management interface exposed by the PMC. The policy objects are encapsulated by appropriate enforcement objects and stored locally. An obligation policy is

encapsulated by an obligation enforcement object (OEO) and stored in the obligation policy table, whereas a refrain policy is encapsulated by a refrain enforcement object (REO) and stored in the refrain filter component. The enforcement objects are removed and deleted from the tables when the corresponding policy is retracted. Policy life-cycle operations are executed on the PMC through the *policy management interface*. These operations are then forwarded to the appropriate enforcement object. When an obligation policy is enabled, the corresponding OEO registers the obligation event specification with the event service. Enabling a refrain policy simply changes the status of the corresponding REO to "*enabled*". An OEO is composed of an *event handler*, a *constraint handler* and an *executor component*. The event handler is the component which subscribes the obligation event with the event service. The event service processes events received from event publishers and disseminates them to obligation policy event handlers based on the subscriptions.

We numbered the interactions in the figure to illustrate the steps involved when obligations are triggered. On receiving an event (1) the event handler initiates the execution of the obligation policy by requesting an evaluation of the constraint specification (2). The constraint handler component of the OEO is called to evaluate the constraint (3). If the constraint is target-based, evaluation of the constraint has to be performed for each of the target objects in the scope of the policy as it may be true for some targets but not for others. In this case, the constraint handler evaluates the set of target objects in the target DSE (4), and then performs the evaluation of the constraint for each of the targets and returns the set of those targets for which the policy is valid. The next step is the evaluation of the refrain policies which may apply to the set of targets returned by the constraint handler (5). The refrain filter component of the PMC identifies the refrain policies which are valid for the set of targets and actions of the obligation and returns a table of filtered actions and associated set of target objects on which each action is to be executed. The OEO initiates the execution of the actions by delegating the task to its executor component (6). The executor component executes the target actions on their associated target sets (7), and the subject actions by calling the main module of the PMC (8) with the name and parameters of the action. The PMC provides a *script actions management* interface that allows action objects to be dynamically loaded into the PMC to extend its functionality, and to be stored locally in a table. The PMC executes a subject-based internal action by identifying it in the table of stored actions and calling its *exec* method to start the execution of the action (9). The executor component of an OEO is a multithreaded component also responsible for the execution of composite obligation actions. Using the structured specification of obligation actions, the executor creates threads to handle the execution of each of the nodes of the obligation action tree. The following subsections describe in more detail the evaluation of constraints, the execution of refrain policies and the event handling part of the obligation policy execution.

7.4.1 Evaluating Constraints

The constraints for both obligation and refrain policies are evaluated in the context of a given set of target objects, which identify the targets of the policy. The runtime policy objects loaded in the policy management components contain a structured constraint specification; the compiler (Section 6.3) resolves the different types of constraints and generates a corresponding constraint object hierarchy. Constraint evaluation can thus be performed based on the type of the constraint. A separate subcomponent of the constraint handler is delegated the task of evaluating each type of constraint. Apart from target-based constraints, all other types of constraints evaluate to either true or false. False indicates that the policy is not valid for any target. Target-based constraints are evaluated for each of the target objects and result in a set of targets for which the policy is valid. If that set is empty, then the policy is not valid for any of the targets. Composite constraints are evaluated by combining the results of the individual constraints based on the logical operator which is used in the composite constraint. In the case of a conjunction the result of the composite constraint is the set intersection of the targets evaluated from the individual constraints. In the case of a disjunction the result of the composite constraint is the set union of the two target sets evaluated by the individual constraints.

7.4.2 Enforcing Refrains

The refrain filter component of a PMC is responsible for applying the enabled refrain policies to the execution of the actions of an obligation policy, in order to filter those actions which the PMC must refrain from performing. The left side of Figure 7.12 shows the tables maintained by the refrain filter, and the right side of the figure shows the table returned by the refrain filter when called to evaluate the refrains for an obligation policy.

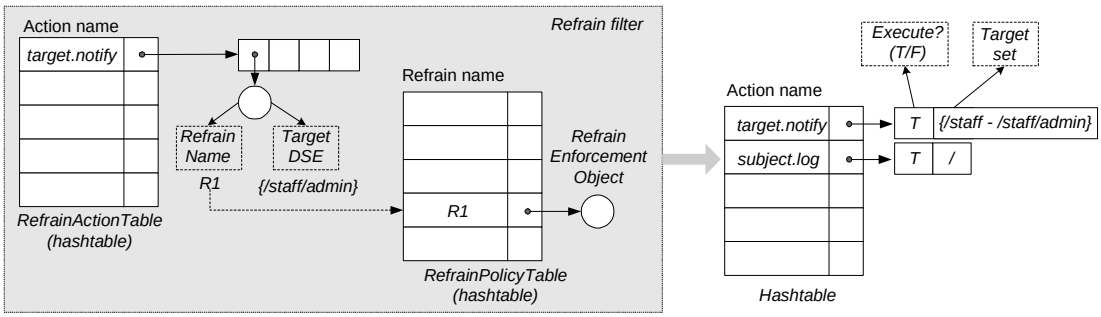


Figure 7.12 Refrain filter tables

When a new refrain policy is loaded in the PMC, the corresponding REO is passed to the refrain filter and added in the refrain policy table (RPT). In addition, the refrain filter extracts the names of the actions from the refrain policy and the target DSE of the policy, and fills in the refrain action table (RAT). For each of the actions in a refrain policy, the refrain filter adds the target DSE of the

policy in the list of those targets to which the action must not be executed. More than one refrain policy may refer to the same action so each action is associated with a list of prohibited target DSEs. Consider the following two policies loaded and enabled in a security administrator PMC:

```
inst oblig availabilityCheck {
    on       userExceedQuota(uid);
    subject  /staff/admin/sec;
    target   t =/staff;
    do       t.notify()|| log(uid);
}
```

```
inst refrain R1 {
    subject  /staff/admin/sec;
    target   t =/staff/admin;
    action   t.notify();
}
```

The obligation policy specifies that security administrators must send a notification to all staff whenever a user's quota is exceeded, to remind them about space availability as a measure against accidental denial-of-service attacks, and also log the event. The refrain policy specifies that the notification should not be send to administrators.

Figure 7.12 illustrates the contents of the RAT and RPT when the refrain R1 is loaded in the PMC. We assume that this is the only refrain policy which applies to the action *notify* and so the list of refrain targets for this action contains only one entry. This entry identifies R1 and the target on which the action must not be specified: */staff/admin*, which is the target of R1.

When the refrain filter is called to apply the refrains for the *availabilityCheck* obligation policy, it returns a table identifying the actions in the obligation policy, together with the set of targets to which each action can be executed after the refrains have been applied. For each of the actions of the obligation policy, the refrain filter subtracts those targets which may appear in the RAT for that action, and whose refrain policy is valid (the constraint of the policy evaluates to true). Note that the refrain policy may be valid for a subset of its targets, and thus only those targets are subtracted from the obligation policy targets. If an action does not appear in the RAT, then the action is allowed for all targets given.

In the above example, the first action of the obligation policy (*notify*) is identified in the RAT. The refrain policy R1 associated with the entry for that action is valid (there is no constraint defined for R1) and so the action *notify* is allowed to execute but only on targets which are in the scope of: */staff* - */staff/admin*. The second action of the obligation policy does not appear in the RAT and is thus allowed to execute with no restrictions. Since this is a subject action there are no targets associated with it. The refrain filter identifies between target-based and subject-based actions. A subject-based action contains an empty refrain target list in the RAT, and if any refrain policy applies to it and is valid, the action is not allowed.

7.4.3 Handling Events

In order to make the architecture adaptable to a variety of event service implementations with minimum change, we isolate the implementation of the component that handles the registration of the event specification, from the main event handler component of an obligation enforcement object. Figure 7.13 shows the architecture for event handlers.

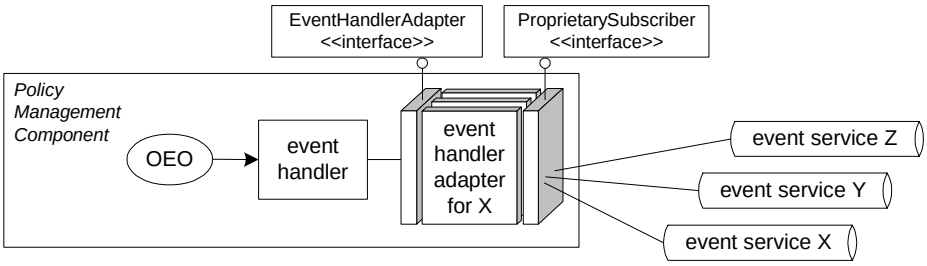


Figure 7.13 Event handler adapters

Each OEO contains an event handler, which interacts with an implementation specific *event handler adapter*. The event handler adapter implements a standard interface to interact with the event handler, and is also specific to the underlying event service. It interacts with the event service to subscribe the event specification passed to it by the event handler, and is also the object that receives the notifications for event triggers from the event service. Event handler adapters can perform event filtering and correlation if that is not supported by the underlying event service.

Event Correlation

Event correlation is performed by utilising the runtime representation of event specifications (see Section 6.2.1). An event handler adapter creates a tree of event consumer objects which correspond to the runtime event specification created by the compiler. Each of the event consumers is responsible for a node of the event tree. Event consumers appearing as leaf nodes in the tree correspond to single events and are responsible for subscribing with the event service to receive notification for those events. Here is a simple example to demonstrate the idea.

Event specification: `(e1 -> e2) && e3`

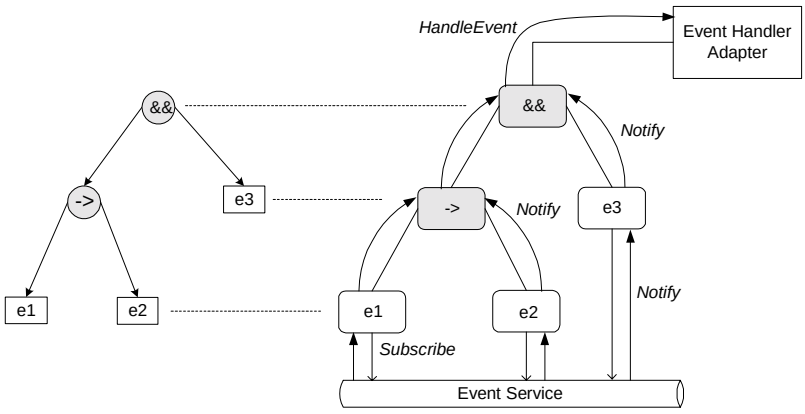


Figure 7.14 Event consumers

The tree on the right side of Figure 7.14 illustrates the tree of event consumers which correspond to the runtime event representation of the left side. Composite event consumers are responsible for enforcing the semantics of the event operator they are representing. The composite consumer for the -> operator will only send a notification to the next level if it receives a notification for event *e1*

before *e2*. Similarly, the composite consumer for the && operator will notify the event handler adapter after it receives a notification for both the *(e1 -> e2)* and the *e3* events. The event consumers which handle the subscription for single events, add the values for the event parameters received by the event service, to the corresponding events. As a consequence, the composite event detected at the root of the consumer tree is enriched with the actual parameter values of the event instances, which might be required in the rest of the process for deploying the policy described previously.

Elvin, which is used to realise the event service in our prototype implementation, supports event filtering through a subscription language, but its current release does not support combination of events. We have implemented event handlers which realise the Elvin notification-listener Java interface, and provide for composition of events at the management component (the consumer side) as explained above.

7.5 Access Control Enforcement

Access controllers enforce all the authorisation policies for one or more target objects and are normally co-located with the targets that they protect. They require close interaction with the underlying access control mechanism, for example, with the host operating system, or a firewall, or the method dispatch mechanism of a programming language, and the means used to interact with each mechanism will vary. Access controllers provide methods to load, enable, disable and unload authorisation policy objects similarly to PMCs.

In previous sections we have concentrated on the enforcement of subject-based policies (i.e. obligations and refrains). For those policies, the default runtime representation of policies as objects was sufficient to enforce the policies. The PCOs use the policy objects to distribute and manage the life-cycle of those objects, including domain membership changes. Policy management components also use the runtime representation of the loaded policy objects to enforce them. With authorisation policies a variety of representations may be required for a policy depending on the underlying security platform on which the target objects run.

The distribution of authorisation policies however, is the same as that described for obligation policies. The policy service creates a corresponding authorisation control object (ACO) for the authorisation policy that the administrator selects to distribute. The ACO uses the authorisation policy object to evaluate the set of target objects on which the policy is to be distributed. Each target object will be associated with an access controller responsible for enforcing access control for that object. The reference to the associated access controller will be accessible from the entry of the target in the domain service. The ACO accesses the reference to the access controller for each

of the targets in the target set of the policy, and distributes the policy object to all identified access controllers.

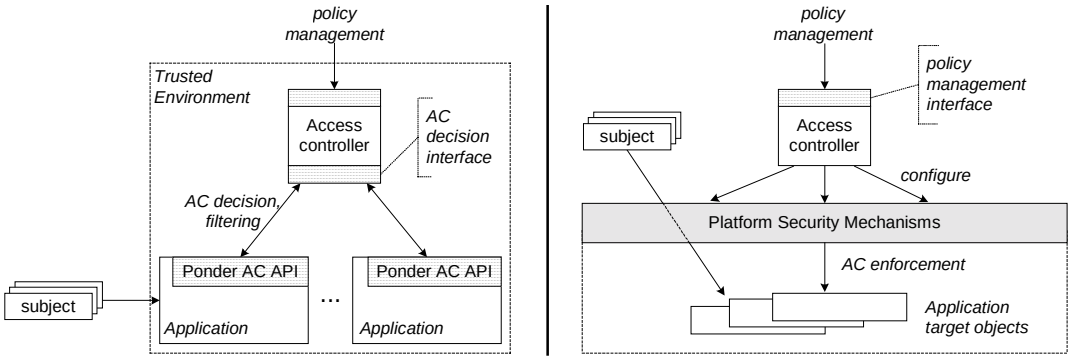


Figure 7.15 Access Control Enforcement

There are two ways of implementing access control in Ponder as illustrated in Figure 7.15. The first approach requires Ponder-aware applications which interface with a local trusted access controller to enforce authorisation policies on their behalf. We call Ponder-aware those applications which implement a *Ponder-AC* interface to allow interaction with access controllers. When an action is requested of a Ponder-aware application, the application forwards the call to the local access controller. The access controller uses its enabled authorisation policies to decide whether the action should be permitted or not, by implementing the access control decision process which was formalised in Section 5.3. In addition, the access controller applies any authorisation filters to the action call parameters and returned values.

The second approach, illustrated on the right side of Figure 7.15, is to configure the security mechanisms of the underlying platform using the enabled authorisation policies. Based on the target security platform for an access controller, the ACO may need to request an additional representation of the policy, stored with the policy entry in the domain service, to pass to the access controller. Specialised code generators can be implemented to map the authorisation policy specification into an appropriate format, which can then be used by access controllers to configure the underlying security platform.

Some of the features of Ponder authorisation policies are not supported by most security mechanisms available, and include constraints and filters. In those cases where it is not possible to map the specified authorisation policies to the available security mechanisms, the first approach to access control enforcement is preferred. Note that combination of the two approaches is also possible.

7.6 Composite Policy Enforcement

7.6.1 Groups

A group is a syntactic scope whose significance is restricted to specification time. Groups can be specified, compiled and stored in the domain service as group entries c.f. domains. Policies inside the group are stored as subentries of the group entry. This allows navigation of the group contents in the same way that domains are navigated. Instantiating a group type creates a domain entry for the group instance, instantiates all the policies contained in the group type, and places the policy instances inside the newly created domain entry. Enforcement of the individual policies stored inside the group entry happens independently for each policy, as has been described in Section 7.3.1. Relationships and management structures are compiled and stored in the domain service similarly to groups. We do not deal with the enforcement of relationships as this would first require the specification of interaction protocols which we haven't included in the language.

7.6.2 Role-based Management

Roles can be specified, compiled and stored in the domain service in the same way as groups. Roles however, provide a semantic grouping and abstraction mechanism, which simplifies management and is thus maintained at runtime by using a role control object, created by the policy service, to manage operations on a role. The management console can be used to select a role from the directory, load it into the tool, and perform control operations on it, without the need to access the individual policies of the role. The corresponding role control object takes care of loading, unloading, enabling, and disabling all the policy instances inside the role to the enforcement components which are members of the subject domain of the role. In addition, the role control object handles the updating of the domain associations and the domain membership changes on behalf of all policy objects contained inside the role.

In the following we specify three roles for our scenario. The first two apply to humans (to the automated agents which run on a user's behalf), whereas the third applies to backup servers.

```
// Role type for backup administrators
type role /managementInfo/roles/backupAdminT(set backupServers, set logData, set users) {

    inst oblig spaceExceed {
        on      userExceedQuota(uid);
        target  ms = /system/servers/mailServer;
        do      ms.notifyQuota(uid);
    }

    type auth+ backupLogsAcT(set lData){
        target  t = /system/servers/backupLog;
        action  t.delete(data), t.read(data);
        when    data = lData;
    }

    inst auth+ backupLogsDev = backupLogsAcT(logData);
```

```
    inst auth+ backupServAdmin {
        target  t = backupServers;
        action  t.startBackup, t.startRestore, t.config, t.jobDelete, t.jobDetails;
    }

    inst deleg+ backupConfigsT(backupServAdmin) {
        grantee g = users;
        target  t = backupServers;
        action  t.config, t.jobDetails;
    }
}

// a backup administrator for the development division
inst role backupAdminDev = backupAdminT(/system/servers/backup/dev/,
                        /system/resources/dev/, /staff/div/dev/) @/system/admin/backup/dev;
// a backup administrator for the development division
inst role backupAdminRes = backupAdminT(/system/servers/backup/res/,
                        /system/resources/res/, /staff/div/res/) @/system/admin/backup/res;
```

Backup administrator roles can be created for each of the divisions or even for each of the departments of the institution. The role is thus specified as a type so that the specification can be reused. Note that the delegation policy specified for this role allows backup administrators assigned to an instance of the role to delegate certain access rights to the users they administer. The access rights are: the configuration of the backup parameters and viewing of job details on the backup servers which are used for backing the files of the corresponding users.

```
// Role for security administrators
inst role /managementInfo/roles/securityAdminT {

    inst auth+ userMgmtAC {
        target  ds = /system/servers/domainServer;
        action  ds.addUser(), ds.deleteUser();
    }

    inst auth+ userBackupMgmtAC {
        target  bs = /system/servers/backup;
        action  bs.createUserAcc(), bs.deleteUserAcc(), bs.setPreferences();
    }

    inst auth+ backupShutdown {
        target  t = /system/serv/backup;
        action  t.shutdown;
    }
} @ /managementInfo/roles/securityAdmin
```

There is only a single security administrator role for the institution so the role is not specified as a type.

```
// Role type for backup server components
type role /managementInfo/roles/backupServerT(set weeklyResources) {

    type auth+ resourceAccessT(set resources) {
        target  t = resources;
        action  t.read();
    }

    inst oblig selfDiskCheck {
        on      Timer.at("23:00:00", "01:*:*");
        do      backServ.checkDisk();
    }

    type oblig weeklyBackupT(set resources) {
        on    Timer.everyDayAt("fri", "*:*:*", "23:00:00");
        do    backupWeekly(resources);
    }

    inst oblig weeklyBackupParallel = weeklyBackupT(weeklyResources);
}

// a backup server for the Parallel department of the research division
inst role /managementInfo/roles/backupServerParallel =
    backupServerT(/system/resources/res/parallel)@/system/servers/backup/res/parallel;

// Role type for backup server components
type role /managementInfo/roles/backupServer2T(set dailyResources)
                    extends backupServerT(dailyResources) {
```

```
    type oblig dailyBackupT(set resources) {
        on    Timer.at("22:00:00");
        do    backupDaily(resources);
    }

    inst oblig dailyBackupInternet = dailyBackupT(dailyResources);
}

// a backup server for the Internet department of the development division
inst role /managementInfo/roles/backupServerInternet =
    backupServerT(/system/servers/backup/dev/internet)@/system/servers/backup/dev/internet;
```

The two role types specified above are used for backup servers. The first one (*backupServerT*) is used for backup servers of the research division, which are obliged to backup data only on a weekly basis. The second role (*backupServer2T*) extends the *backupServerT* role, thus inheriting all of its policies, and adds another obligation to perform a daily backup of the parameterised data. This role is used for backup servers responsible for the data in the development departments of the institution which require daily backups in addition to weekly backups. We show two examples of instantiating these two roles for two of the departments (one in the research division and the other in the development division).

Role Assignment

An important dimension of role-based management is that of assigning users to roles, and activating/deactivating roles. A user representation domain (URD) is a persistent representation of a human user in the system [Lupu 1998]. In the model defined in this thesis, a URD contains two different kinds of objects: one or more policy management components and a user profile object (UPO). PMCs can act as proxies for the user to allow access to resources permitted by the authorisation policies or they can act as automated agents, interpreting obligation policies on the user's behalf. A user is assigned to a role by including a PMC from the user's URD into the subject domain of that role. PMCs are application specific and different PMCs can be used to support different management functions (e.g. security management, backup administration etc) if the user is assigned to different roles with different automated management needs. The UPO contains information about the roles and domains to which a user's PMCs have been assigned. Note that automated agents implementing policies are also PMCs which can be assigned to roles but do not represent a human user.

Figure 7.16 shows the steps involved in assigning users to roles. These steps are summarised below in relation to the user-role management tool (Figure 7.17), which is used to manage users for which policies are specified in the system. This tool requires interaction with the domain browser to select users (i.e. user representation domains), and roles from the domain service. The tool consists of three sub-views:

In the *User-Management View* new users can be created, assigned and removed from domains. Creating a user means, creating a URD for the user, and a UPO, which is stored inside the URD. Assigning a user to a domain implicitly creates a reference in the selected domain, which points to a PMC within the URD. This corresponds to steps 1 and 2 in Figure 7.16.

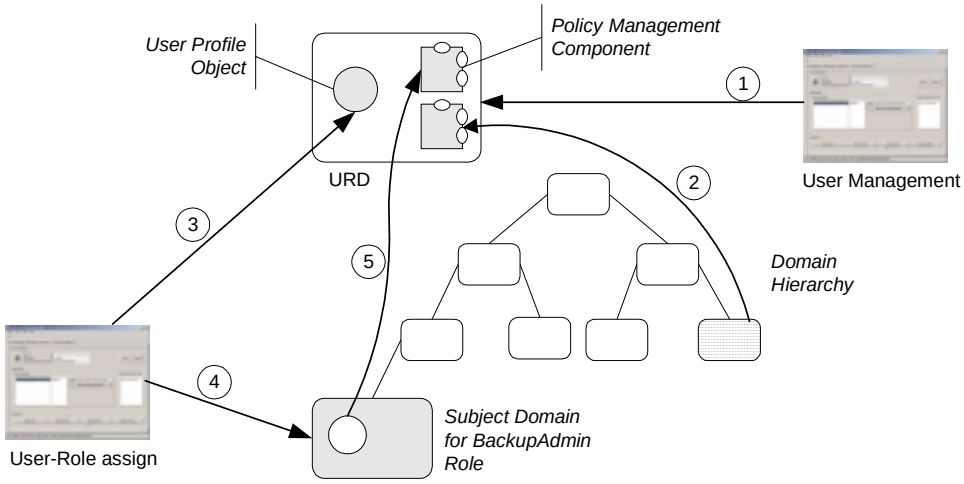


Figure 7.16 User-role management steps

In the *Management Components View* new policy management components can be created (or deleted) for users and stored in the users' URD. This is also related to step 1 of Figure 7.16. PMCs stored in domains (URDs or other domains), can be started and stopped remotely onto any host, using the user management tool. This requires a *PMC-Server* to be running on the remote host. For example, when a new backup administrator for the research division of our scenario is added to the system, a corresponding URD will be created under */managementInfo/URD*, and a backup-enabled PMC will be instantiated in the URD. The administrator is then assigned to */staff/admin/backup/res*, and his PMC is started on the research network.

In the *User-Role Management View* users can be assigned and removed from roles. This simply updates the list of roles to which the user is assigned within the UPO (steps 3 and 4). Assigning a user to a role requires the selection of a PMC from the user's URD, that will be used to represent the user in the role. Roles assigned to a user can be selectively activated and deactivated. Activating a role creates a reference in the role's subject domain which points to the associated PMC in the user's URD (step 5). This implements the RBAC concept of sessions [Sandhu et al. 1996]. The new backup administrator in our scenario can be assigned to the backupAdmin role by selecting the administrator's URD, the backup-enabled PMC from this URD, and the backupAdmin role using the domain browser. The role can then be activated and deactivated with a click of a button by selecting it from the list of roles assigned to the user.

In the current implementation users do not interact with their PMCs. However, future implementations of the PMC can prompt the user about the actions to be taken and may require input from the user before taking certain actions. Although the prototype implementation includes a graphical interface for viewing the status of a PMC and the policies which run on it, this will need to be extended to allow users to remotely interact with their PMCs.

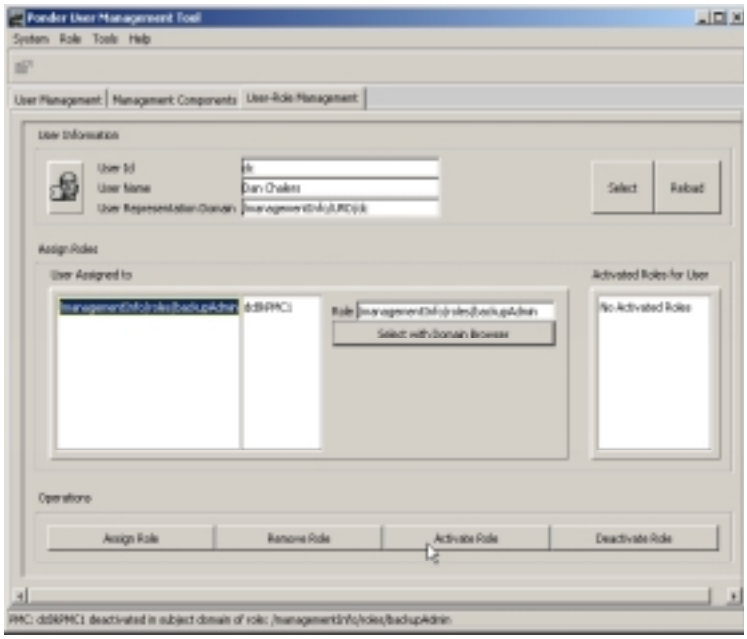


Figure 7.17 User-role management tool

7.7 Conclusions

In this chapter we have proposed an architecture for deploying Ponder policies and discussed the issues involved in the deployment process. A key aspect of the deployment model is the automated distribution of Ponder policies to their enforcement components. Ponder policies explicitly define their subjects and targets enabling the distribution of policies without the need to manually manage the associations between policies and enforcement components. In addition, we avoid the need for a pull architecture where enforcement components must query the policy repository for the policies that apply to them. This makes it easy to provide the functionality of dynamically reacting to domain membership changes, and to enable automated distribution of already loaded policies to new enforcement components, or retraction from enforcement components which are removed from the domains to which the policies apply.

Obligation and refrain policies are enforced by generic policy management components whose operation does not depend on the platform on which the system is implemented. We have described the architecture for policy management components, which can serve as a reference for specific implementations of agents enforcing obligation and refrain policies.

Access control policy enforcement requires close interaction with the underlying access control mechanism, for example, with the host operating system, or a firewall, or the method dispatch mechanism of a programming language. The distribution of authorisation policies to their enforcement components, termed access controllers, may involve alternative representations of the policy, which are generated by the policy compiler and stored in the domain service. Access

controllers are responsible for configuring the underlying security mechanisms based on the policies they receive. For Ponder-aware applications, interaction with a local trusted access controller will be possible to provide for a generic access control decision mechanism based on the enabled authorisation policies that apply to the applications. This includes enforcement of features which may not be supported by current access control mechanisms such as: constrained authorisations and filters.

The architecture includes support for role-based management. A role control object is created to handle the distribution of all policies specified inside the role in the same way this is done for basic policy objects. Another dimension of role-based management is the assignment of subjects to roles and the implementation of sessions whereby only a subset of the roles assigned to a subject are activated. We handle role-activation by placing policy management components in the subject domain of the role. Policy management components can be used as proxies for human users and are stored inside a user's URD.

Finally, we have provided evidence of the feasibility of our policy framework by presenting an implementation of the deployment architecture described. The implementation includes a policy administration toolkit providing a graphical user interface for managing policies from specification to deployment. The toolkit includes a policy specification IDE, a management console tool for managing the distribution of policies, and a user-role management tool for managing users, roles and automated management components. Extensibility is build into the toolkit by allowing easy customisation with new tools, and integration through a domain management tool used for designing and accessing the domain service. The implementation of the toolkit is summarised in [Damianou et al. 2002].

Chapter 8

Critical Analysis

The work presented in this thesis has been motivated by the need to support policy specification and deployment for distributed systems using a generic policy specification language in an integrated environment. In this chapter, we examine the limitations and deficiencies of the proposed framework and indicate aspects of the work that can be improved. We place the framework in the context of relevant work, and provide a critical evaluation of the language, the proposed deployment architecture and the implementation.

8.1 Relationship to Relevant Work

In this section we compare our work to important relevant work presented in Chapter 2, and outline the relationship between the Ponder framework and the work in related areas. The Ponder framework is directly applicable to security management with emphasis on non-discretionary access control [Abrams 1993], where administrators have the authority to specify security policies with delegation and propagation of authority permitted only within the scope defined by those policies. However, we have demonstrated using examples, how Ponder can be used to specify both DAC and MAC policies (see sections 3.6 and 4.5.1). In addition, RBAC as well as role-based management (RBM) policy specifications are intrinsic in Ponder and were identified as part of the design requirements of the language. RBAC is a subset of RBM as described in Section 2.5.3, although the two models take different approaches. The RBM model supported in Ponder extends RBAC models with the duties of the subjects assigned to organisational positions in the form of obligation and refrain policies. In addition, the Ponder role model provides class level inheritance based on well-defined object-oriented concepts as opposed to instance level inheritance supported by the RBAC models as a means of reusing permissions. Class level inheritance allows specialisation of roles and parameterisation with policy targets, providing more flexibility in reusing specifications. See [Lupu et al. 1997a] for a comparison between RBAC and RBM.

A lot of the work on policy specification relates to security, but none includes the range of policies available in Ponder, and most lack the extensibility features and necessary abstractions to support large-scale policy specifications. Formal logic-based approaches (see [Abadi et al. 1993; Jajodia et

al. 1997; Ortalo 1998; Barker 2000]) are generally not intuitive and do not easily map onto implementation mechanisms. They assume a strong mathematical background, which can make them difficult to use and understand.

Graphical security specification approaches (see [Hoagland et al. 1998] and [Heydon et al. 1990]), although attractive, have a very limited scope to satisfy the requirements of security management; they apply to specific situations (e.g. file system security) and do not scale. Similarly, other approaches are limited to specific types of policy. Examples include RSL99 [Ahn et al. 1999] specific to separation of duty constraints in RBAC systems, and a language for RBAC constraints described in [Chen et al. 1995]. It can be shown that both of these languages are subsets of OCL and we can thus specify all of their constraints as Ponder meta-policies. We believe that the design of specialised languages complicates policy specification because it leads to a multiplicity of incompatible policy specifications that need to be applied to the same application domain. This increases the possibility for policy conflicts and inconsistencies, and requires complex enforcement architectures which need to cater for the enforcement of various forms of policy. Ponder combines security management and policy-driven network management specification in a single specification language.

Recently, there has been considerable interest in approaches used to specify higher-level policies such as SLA's and trust statements. We see Ponder as complementary to some of these approaches; Ponder can act as a lower-level of both specification and implementation for SLA or trust specifications. This is related to the issue of policy refinement, which we haven't addressed in this thesis, whereby high-level specifications can be refined to Ponder policies. In particular, we consider trust specifications as abstract authorisation and security management policies, which can be implemented using access control and authentication mechanisms. It should therefore be possible to refine trust specifications to Ponder security policies. In addition, Ponder authorisation policies can be used to specify the conditions under which trust relationships can be established [Mont et al. 1999a]. Approaches to the problem of refinement can be found in [Mont et al. 1999b; Michael et al. 2001].

Trust management frameworks combine authentication with authorisation [Grandison et al. 2000] and are seeing an increasing application in the design of security architectures. We compare the Ponder framework with the Strongman security policy architecture [Keromytis et al. 2001] presented in Section 2.3.1, which is an example of this approach, and draw some conclusions. In [Keromytis et al. 2001] they acknowledge the need for a single method of policy specification based on the results of a survey on Internet security presented in [Howard 1997]. However, they criticise the use of a "*universal*" high-level language as being too complex, clumsy and susceptible to mistakes. In addition, they see such an approach as "*inappropriate because it often presumes*

homogeneity and cannot handle mixtures of multiple mechanisms for different parts of the same network". Ponder was designed to specifically avoid these problems; in Ponder we can specify security policies which can map to heterogeneous security mechanisms and platforms. Their approach instead allows the use of multiple application-specific policy languages to specify the security policies for particular applications. These languages map (using compilers) to a common policy interoperability layer implemented with Keynote [Blaze et al. 1998]. The disadvantage of this approach is that certain features they want to uniformly provide in their architecture, such as delegation, must be specified at the lower-level (i.e. in Keynote) making the specification more complicated. We believe that interoperability is best provided at a higher-level of specification using a single language. Note that Ponder can map to Keynote to take advantage of the Strongman security mechanisms if the application domain requires it.

In general, trust management frameworks couple access control policies and authentication at specification time. Ideally access control policy specification and enforcement should be decoupled from the mechanisms that enforce authentication of users in the system. The binding of users to public-keys and the specification of credentials can be done separately. Credentials can then be used in a Ponder system to restrict assignment of users to domains or roles, using an approach similar to the Trust Policy Language (TPL) approach described in [Herzberg et al. 2000]. Non-authenticated users, or users with inadequate credentials, will not be allowed to become members of certain domains, thus restricting their access rights by limiting the policies that apply to them. Note that Ponder can currently support similar functionality to that of TPL by using authorisation policies to restrict the assignment of users to domains and roles.

From a different perspective, Ponder can also complement approaches that concentrate on low-level policy specification. Examples include: directly specifying Java policies to control mobile code security, using compiled or interpreted programs to specify policies (e.g. using C programs as described in [Bos 1999]), or using an information modelling approach for QoS policy specification as prescribed by the IETF [Moore et al. 2001]. Policies can be more conveniently specified in Ponder and then mapped to any lower-level specification using automated software tools (i.e. backends to the Ponder compiler presented in this thesis). The information modelling approach followed by IETF can be used for storing and distributing Ponder policies across network boundaries. Implementation experiments have shown that it is easy to map Ponder policies to both the PCIM [Moore et al. 2001] and QPIM [Snir et al. 2001] information models defined within the IETF.

In policy-based networking most of the tool support comes from the industry and is based on the IETF policy framework. Unlike the Ponder framework, most of these tools are specific to quality of service and bandwidth management with few providing access control configuration, while most of

this work focuses on managing individual network elements. Scalability to enterprise wide management is not obvious as the dissemination of policies to specific elements is often performed manually whereas in Ponder this is automated based on domain membership of subjects and targets, which are explicitly specified in policies. In addition only few of the commercial products support role-based management features. Most of these products are optimised to work on single-vendor networks and are implemented to manage vendor-specific hardware with only a few working towards hardware-independence. Ponder is motivated by the need to manage heterogeneous networks and the framework we have described in this thesis enables the implementation of multi-vendor policy-based management products.

A fundamental problem which remains unsolved in all work on policy-based management is the issue often referred to as policy validation [Verma 2001], i.e. given a policy, can it be instrumented in an existing hardware/software configuration, and is it possible to check and validate that this is the case before deploying the policy? The answer to these questions could form an interesting research direction within the Ponder framework.

8.2 Critical Evaluation of the Framework

In this section we discuss the aspects of the policy-based framework that need to be improved and describe shortcomings of the work. We start with the policy language and use feedback from users who have tried to apply the language to specific management scenarios, as well as our own experience, to identify those features of the language that are hard to use or understand.

8.2.1 Policy Language Design

The specification of constraints is one of the most powerful features of the language. Although the use of OCL enables the specification of complex constraints, some features of OCL are not well defined [Vaziri et al. 1999], and complicate the formalisation of the language. This makes the process of analysing policies more complicated. The current syntax does not allow for the specification of prioritisation rules, and could be simplified if sub-types of meta-policies are defined for well-documented constraints such as separation of duties, user-role assignments, concurrency constraints etc, as identified in the examples in Section 4.5.1.

Experimentation with specifying policies for network-level firewall rules or IP traffic control often results in the need to use IP addresses for the subjects and targets of policies Similarly, in key-based authorisation systems where access rights are specified in terms of the public-keys owned by principals, subjects and targets are more conveniently defined using keys or certificates. Future investigation could identify the need to extend the grammar with alternative ways of specifying

subjects and targets or for specification-time mapping of domains to IP addresses or public-keys. Users often find it hard to use domains in these situations. In general, situations such as in mobile systems, where users and/or resources cannot be explicitly identified make it difficult to map those entities to domains. This creates an additional burden within the Ponder framework, which assumes the specification of policies based on domains of objects.

The definition of relationship structures in the current version of the grammar raises questions that have to do with the necessity of using relationships in certain situations; the policies specified in a relationship can sometimes be specified in the roles involved in the relationship. This results from the fact that we have not addressed the specification of interaction protocols in the current grammar. The addition of interaction protocols in relationships, to define the permitted message exchanges between occupants of the roles, is important to complete the definition of relationships. A related question has to do with the use of both roles and relationships to model and enforce policies that apply to humans. Ponder can be used to specify such policies in order to support reasoning about the behaviour of the system. However, its current use is restricted to policies that apply to and can be implemented by automated components. Policies applying to human users can be partially supported in future implementations by extending policy management components with the functionality to prompt the user of actions to perform as specified by the obligation policies that apply to that user.

Delegation is a fairly ill-defined term [Abadi et al. 1993], and an area in which policy specification is still lacking. The use of Ponder delegation policies sometimes results in confusion. Users often misuse delegation as a way of specifying access rights or as a way of controlling delegation and revocation in the system. Delegation policies in Ponder simply specify the authorisation to delegate access rights. The actual delegation and revocation of those access rights is a separate issue. A runtime API will need to be designed to allow users to interact with the underlying authorisation service to control delegation and revocation of access rights. In the literature delegation is often associated with the ability to specify the principals that can act on behalf of another principal without binding the delegation specification to specific access rights [Abadi et al. 1993]. In addition, arguments often exist in favour of specifying the sequence of cascading delegations allowed instead of simply the number of cascading levels. These issues are not easy to specify in Ponder and may require additional research and application of Ponder delegation policies in specific security scenarios.

The idea of refrain policies as a form of subject-based negative authorisation is not addressed in other access control specification approaches, and is often queried by users. The question to ask is the following: how useful is it to specify what subjects must refrain from performing if you have to assume that subjects are well-behaved in order to expect enforcement of such policies? Negative

authorisation policies can also be distributed to the subjects and used as refrains within subjects. Is there a need to specify both refrain and negative authorisation policies or is this an implementation issue that is better to be addressed in the deployment architecture? The answer to these questions is that at some stage there is a need to distinguish between subject-enforced and target-enforced policies. In addition, refrain policies can also filter executions of internal subject-based actions, whereas negative authorisations restrict the execution of only target-based actions.

The issue of scalability for the language has been addressed throughout the thesis and was an important design goal identified in Section 1.2. Scalability was successfully achieved in three ways: (i) The object model (see Appendix A) allows the language specification to be easily extended in the future with new types of policy added as sub-classes of existing policy types (ii) The use of domains to group the objects to which policies apply, and the composition of policies into composite policy structures to cater for large-scale organisational policy specifications allows scalability to large systems (iii) The use of parameterised instantiations of policy types and inheritance for composite policy structures permits reuse of specifications and enhances scalability as well as flexibility. We provide a comprehensive summary of achievements regarding the design of the language in Section 9.1

8.2.2 Management Architecture

The policy deployment model is based on the use of domains as a means of grouping objects to which policies apply. This provides for scalability and enables the handling of changes to managed objects and deployed policies. The algorithms involved in domain membership evaluation offer scope for optimisations. An algorithm for deciding membership of an object in a given DSE without the need to re-calculate the DSE is detailed in [Yialelis 1996]. In addition, we haven't investigated the possibility of runtime changes to policy structures already stored in the domain service. In the current model we assume that policies that need to be changed are retracted if already distributed, removed from the domain service, modified, recompiled and then stored back and re-distributed to their enforcement components. This process becomes complicated when modifications are made to stored policy types from which a number of policy instances are created and deployed. Note that changes to already deployed policies may also result from modifications of the domain structure. For example, the addition of a newly created domain A as the parent of domain /B/C requires that policies whose subject or target is /B/C be changed to /A/B/C.

The enforcement of Ponder authorisation policies on various security platforms needs to be investigated further; different security platforms have different enforcement semantics and some may include restrictions which prevent direct mapping of certain features of Ponder policies onto these platforms. Most platforms do not support the enforcement of constrains or authorisation

filters. A Java implementation of an access control service that maps Ponder policies to Java code [Corradi et al. 2000] demonstrates the difficulty of mapping authorisation policy filters to existing mechanisms (in this case Java security). However, filters are directly applicable to database systems, in which case they can be enforced by externally implemented access controllers that are trusted by the application as described in Section 7.5.

Although we have formalised the implementation of delegation policies using authorisations (Section 5.4), their enforcement is not implemented in the current version. Enforcing delegation policies assumes an authorisation service with the ability to keep track of delegation steps for each grantor. Authorisation and delegation rely on authentication, which is often achieved through authentication protocols. Although our goal was to decouple the specification of authorisation policies from the authentication in the system, further work will need to extend the enforcement architecture with mechanisms for providing authentication of subjects, as well as integrity and secrecy of policy information communicated in the system.

The Ponder framework is distributed in that all objects including policies, target managed objects, policy control objects and policy management components are maintained on distributed domain servers. Enforcement components are distributed and normally co-located with the objects they manage, and making a policy decision does not require those components to query policy servers in order to identify the policies that apply to the current request. Policies are distributed to their enforcement components and the policy decision process is performed locally at each enforcement component. Scalability is also provided by the fact that the addition of new enforcement components in the Ponder system does not require any change in the code of the management system, or any effort to activate the new enforcement components. Inclusion of an enforcement component in a domain automatically applies existing policies to that component.

At the heart of every management architecture are the definitions of concrete managed objects in the respective information model. We define a simple object-oriented model on which the management architecture is based. The model includes the definition of policy objects, domains, and policy enforcement entities as subclasses of a generic managed object class. The information model does not include detailed definitions of target objects such as system elements, logical and physical network elements, services etc. as work on this exists and is underway within the DMTF [DMTF 1999a]. Instead, it can easily be extended to include other models; Ponder could be applied to CIM defined managed objects.

We acknowledge the existence of available technologies that can be used to implement policy management solutions and refrain from specifying new technologies or protocols in our management framework. The architecture is based on simple ideas and can be implemented using existing protocols. We do not define any new protocols for the communication of management

information (i.e. policies). Instead, depending on the implementation of the architecture and the requirements of the application, we assume that management information is communicated using existing mechanisms. These can be object-middleware such as CORBA and distributed Java (Java RMI), or protocols such as HTTP, the Common Open Policy Service Protocol (COPS) or the Internet management architecture's Simple Network Management Protocol (SNMP) for network level enforcement of policies.

8.3 Critical Evaluation of the Implementation

The main purpose of the prototype we have presented in this thesis was to evaluate the feasibility of implementing the proposed framework. We use the experience gained from this implementation in order to identify the issues that remain unsolved and those that must be improved on in future implementations of the framework.

An important aspect of the implementation was the policy compiler, which is used to generate the enforceable policy representations. The most important feature of the compiler is its customisation with multiple backends used to map policies into a variety of representations. This feature was appreciated through the various uses of the compiler in projects involving the mapping of Ponder authorisation policies to various access control mechanisms, as well as the translation of Ponder policy specifications to XML for transmission across the network.

The implementation of the compiler does not contain any optimisations to enable faster evaluation of the policy elements at runtime. Although SableCC, the parser generator used to build the compiler, proved a very flexible way of extending the syntax analysis with semantic checks, it generates a very large number of classes making the execution of the compiler slow. The current implementation of the Java code generator is inefficient and the generation of Java code for the policies is slow. We have compared the use of the Java code generator with that of an access control policy generator which maps authorisations to XML code, to compile and store 1000 authorisation policies in an LDAP server. The file was 6000 lines long and the tests where executed on a 450Mhz Pentium III machine with 128Mb of RAM running Windows 2000. The CPU time used was measured using the Windows Task Manager which indicates CPU time of processes with a precision of seconds. When no code generation was selected, the compilation – parsing and semantically analysing the policies – required 11 seconds. Code generation with the experimental access control code generator required 25secs CPU time to compile the policies, but more than 4 minutes to store the policies as entries on the LDAP server. Accessing the LDAP server thus proves slow. Generating Java code using the Java code generator took considerably longer: the CPU time required was over 17 minutes, but the elapsed time was 42 minutes. The Java code

generator creates a Java file for each policy, stores it on the local hard-disk and executes the Java compiler (i.e. javac) to generate the bytecodes for the file. It then reads the bytecodes and stores them in LDAP. The reason for the long elapsed time is mainly due to accessing the disk and executing the Java compiler. The performance can be significantly improved if the code generator directly creates the bytecodes from the Ponder specification using bytecode manipulation tools without the need to create Java files and running the Java compiler. One such tool is the Byte Code Engineering Library (BCEL), an open source project that provides an API to create classes from scratch at runtime [Apache 2001].

Finally, the lack of a formal specification for the OCL part of the Ponder grammar made it difficult to implement some of the features of OCL in the current version of the compiler. In addition, the compiler does not check with the domain service to validate the domain paths and target method calls referenced in a policy specification. It is assumed that the policy administrator specifies valid policies in that respect. This feature must be added to the compiler as a customisable component, which can perform requested checks on the domain service based on the underlying implementation of that service.

The Ponder deployment model and toolkit are implemented using Java and LDAP, and use JNDI to interface with LDAP directories. We have used the Netscape directory server version 4.2 [Netscape 2000], but restricted the implementation to a single server. It is important to experiment with the use of multiple servers in order to provide for replication and fault-tolerance of both target objects and policies. The use of multiple LDAP servers will also enhance the scalability of the implementation, and is important in enterprise wide networks where access to servers on different sub-networks is either too slow or not possible.

Other technologies widely used to implement distributed systems could have been used instead for the implementation of the deployment model. These include:

- Middleware solutions such as CORBA [OMG 1999a] or other ORBs and communication middleware such as TSpaces [Lehman et al. 1999] called by its creators as "*an intelligent Connectionware component*".
- Java web-based management technologies such as the Java Dynamic Management Kit [Sun 1999a; Sun 1999b], Jini network technology [Arnold et al. 1999], Jiro technology - an implementation of the Federated Management Architecture [Monday et al. 2001] - and Java Message Service [Sun 2000] solutions.

CORBA and other middleware solutions such as Jini and Jiro are increasingly being used as enabling infrastructures in distributed systems. It would thus be important to assess the practical contribution of the Ponder framework by implementing it on top of these technologies, to support

management of CORBA-based or Jini-based applications. The CORBA security services [OMG 2001] provide support for authorisation and role-based access control, with the philosophy that security unaware applications should be able to run securely on a secure ORB without any active involvement on the site of the application objects. The ORB Services replaceability package requires implementation of an *access control interceptor,* which determines whether an invocation can be permitted. This request-level interceptor can be substituted to interface with our access controllers in order to enforce Ponder authorisation policies for CORBA-based application objects. In addition, Java web-based management technologies (such as JMX/JDMK) are used to provide for management of Java applications. The agent functionality provided by these technologies allows the implementation of agents which can be accessed using a variety of protocol APIs such as SNMP, HTTP, CIM/WBEM and TMN. We can use this functionality to implement PMCs which can be managed using web-browsers or any of the other communication protocols based on the application domain.

We have evaluated a variety of existing solutions to provide an event service for our management system (e.g. the Cambridge Event Service [Bacon et al. 2000]), but haven't been able to find a solution that satisfies all of the requirements of such a service. These include:

- Publish/subscribe functionality
- Event filtering
- Event correlation
- Java client interfacing

We have chosen to use Elvin [Segall et al. 1997; Arnold et al. 2001] mainly due to its Java API implementation. Elvin can accept and deliver 'packets of information' called notifications between programs which are part of an Elvin network. The Elvin developers claim that a single Elvin server can handle event volumes of over 10000 events per second, and up to 1000 client connections. Multiple Elvin servers can be setup to share the load of notification distribution in an "Elvin Federation" and increase availability, reliability, management and performance. However, we have used a single Elvin server in our current implementation and we haven't implemented any fault-tolerance in the Elvin consumers (i.e. the PMCs). Elvin provides an easily programmable API for many common languages and provides a subscription language giving the programmer freedom to create complex expressions for receiving only the notifications that are required. The problem is that the current Elvin implementation does not support composition of events (see Section 7.4.3). Our implementation of event composition is thus inefficient as it composes events at the consumer site, i.e. within the PMC, and results in a lot of unnecessary notification delivery by the Elvin server.

The execution time for obligation and refrain policies implemented in policy management components depends on the evaluation of domain scope expressions in the target element of the policies. Although the efficiency of the algorithms used was not considered in detail it can be shown, using the formalisation presented in Section 5.3.3, that the computation is not hard and is always guaranteed to terminate. Note that we have not considered network failures during action execution or during the evaluation of domain scope expressions by PMCs.

8.4 Conclusions

In this chapter we have evaluated the Ponder framework in two ways:

- We placed the proposed framework in the context of relevant work and tried to answer the following question: What is the relation of the work presented in this thesis with work which exists or is underway in the area of policy-based distributed systems management? This includes, other policy specification approaches, management architectures and trust management solutions. The Ponder framework is significant in solving some of the problems of other approaches related to the specification and enforcement of policies, as well as complementing some of these approaches.
- We described the problems and shortcomings of the Ponder language, the proposed management architecture and the current implementation, and identified the issues that have been hard to provide, and those which need to be resolved in the future.

The Ponder framework uses a number of services, which it considers available, such as directory service, event service, monitoring service etc. However, many of the implementations of such services have serious limitations in functionality, performance or both as has been discussed for the LDAP and Elvin service implementations used within the Ponder framework. This causes the performance of some components (e.g. compiler) to be relatively disappointing. Note however, that no major efforts were undertaken to optimise the implementation, and that these components are mostly off-line components. The compilation of policies for example is not a frequent operation, and it takes place off-line thus it does not influence the overall performance of the system. Implementation still needs to be provided for some of the more complex elements of the language, and the language can be extended to integrate with other complementary approaches.

Chapter 9

Conclusions

In this chapter we provide a summary of the work presented in this thesis, and identify what has been achieved. We conclude with an account of future work resulting from the evaluation of the framework presented in the previous chapter.

9.1 Review and Discussion of Achievements

"*Policy-based network management (PBNM) turned out to be difficult to put into practice. Early adopters have found that developing and deploying policies is not simple, cheap or quick. Instead, PBNM has been a time-intensive, complex, expensive process. Additionally, it has demanded that the enterprise organisation mutate to match the technology—rather than the technology meeting the enterprise's management needs. Finally, many PBNM solutions tend to be single-vendor approaches and, as such, are not readily employed in existing multivendor networks*" Michael Jude, March 2001 [Jude 2001].

Quotes like the above are common in describing policy-based management solutions and are drawn from the experience of users with using tools that promise to allow policy control through simple graphical interfaces, and to enable enforcement of these rules into the network, as traffic management or access control instructions. Although useful tools are being developed and vendors are improving their policy-based management products, there is a lack of complete vendor-independent solutions to the problem of policy management. The work presented in this thesis helps in that direction because it defines a complete framework for policy-based management from which future work in the area can benefit.

We have presented a comprehensive survey of security models and policies, policy specification approaches, policy-based management architectures and platforms, and acknowledged the need for a generic policy specification language. We proposed a definition of policy and identified the requirements for the design of the policy specification language, in an effort to help in understanding the concept of policy-based management often used in the literature in an ad hoc manner. We believe that a common high-level declarative language for different applications of policy-based management is the right approach to policy specification, and an important technique for simplifying the management of complex distributed systems. By keeping the language simple

and by focussing on implementable policies, we believe that the language can be integrated and effectively used in management frameworks that govern the behaviour of large-scale distributed systems. Furthermore we have shown how Ponder policies can be used to specify most of the situations currently described in more specialised languages, as well as most of the access control models available.

The Ponder framework is derived from experience learnt from work on policy-based management at Imperial College over the past 15 years. We use the experience gained through earlier attempts at defining a policy notation and implementing various different standalone tools and components, and define a complete policy-based management framework to support the whole policy life-cycle relating to specifying and managing deployed policies. The design of the language, the architecture, and the implementation of the integrated toolkit presented are based on solid, well-defined management ideas and concepts developed at Imperial College and used in a number of projects throughout the years. We summarise them here in relation to the Ponder framework:

Domains: The unit of object grouping in Ponder is the domain [Sloman et al. 1994a]. Domains are a unit of management akin to file directories in operating systems, and provide hierarchical structuring of objects. Policies apply to domains, thus each domain holds references both to the objects within the domain and to the policies that currently apply to the domain. Domains are objects themselves, and so can be included in other domains. This allows hierarchical structuring and self-management policies that apply to domains themselves. In order to increase flexibility further, objects and domains can be members of more than one domain, which supports scenarios where objects are members of different groups.

Policies and Meta-policies: We define policy as: *A persistent declarative specification, derived from management goals, of a rule defining choices in behaviour of a system*, based on the definition of policy in [Moffett et al. 1993; Sloman 1994b]. The policies are interpreted rather than compiled into the code of agents, so can be changed dynamically thus changing the behaviour and strategy of the system, without modifying the implementation or interrupting the system's operation. Application-specific conflicts, which arise from the semantics of the policy, cannot be detected automatically without a specification of what a conflict is i.e., the conflicts are specified in terms of constraints on attribute values of permitted policies. We call these constraints meta-policies, first introduced in [Moffett et al. 1993], as they are policies about which policies can coexist in the system or what are permitted attribute values for a valid policy.

Roles and Relationships [Lupu 1998]: Roles provide the means of grouping policies related to a position in an organisation such as a staff member, customer support manager or Chief Executive Officer (CEO). A role can also group policies relating to a specific agent such as one that registers new users or adaptively manages Quality of Service in a network. It thus conveniently groups the

policies, which specify the rights and duties for an agent or position. A person or an agent can then be assigned to (or removed from) a role and so acquires all the policies applying to the role. Relationships between roles are used to define additional policies which are not part of the individual roles but specify how the roles interact e.g. that a manager agent is permitted to perform specific actions on a worker agent and the worker needs to periodically report its status to the manager agent.

Based on the above concepts, we have designed Ponder, a language for specifying policies for management and security of distributed systems, which can be directly mapped into an implementation. An important contribution of this thesis is that we have refined and elaborated the above concepts into an integrated policy-based management framework. Ponder was designed using the requirements identified in Chapter 1, and includes authorisation, information filtering and delegation policies for specifying access control, and obligation and refrain policies for specifying subject-based management actions. Ponder thus provides a uniform means of specifying policy relating to a wide range of management applications – network, storage, systems, application and service management. Access control policies allow us to capture the wide range of access control rules that a typical distributed system will implement, from rules for firewalls, to access controls for the users of services (databases, web), to standard operating system controls. We have addressed the issue of delegation of access rights and formalised the implementation of delegation policies using authorisations. The ability to automatically take appropriate actions when particular events occur in a distributed system is addressed by Ponder's obligation policies, which allow us to define the management actions that need to be taken when specific events occur. The advantage of our approach is that a single language is used to specify all these policies and the policy writer can use an abstract model for specification, leaving the translation of Ponder policies to underlying mechanisms, to software tools.

With Ponder all the managed objects in a distributed system are organised into hierarchical domains. Policies are then written in terms of domains, and domain membership is evaluated whenever policy enforcement is performed. This means that there is normally no need to write policies specific to an object. Instead, whenever an object is added to a domain, the policies that apply to that domain will automatically be applied to the object, until either the object is removed from the domain or the policy is disabled.

The use of a unified model for access control and automated management allows us to easily group different types of policies into roles, and to group roles into management structures. Policy grouping is an important and powerful technique for large-scale policy specification and often lacking in the other languages. The Ponder composite policies (groups, roles, relationships and management structures) allow structured, reusable specifications, which cater for complex, large-

scale organisations. Ponder's object-oriented features allow user-defined types of policies to be specified and then instantiated multiple times with different parameters. This provides for flexibility and extensibility while maintaining a structured specification that can be, in large part, checked at compile time.

Meta-policies in Ponder provide a very powerful tool in specifying application-dependent constraints on sets of policies such as separation of duties and self-management using a declarative notation based on OCL. They allow the global constraints of a system to be preserved by restricting the applicability of policies. Ponder itself is declarative which aids in the analysis of policies. We have demonstrated the use of the language through examples which show the range of policies that can be specified in Ponder including a variety of application specific constraints; we have shown how Ponder can be used to model most of the examples described by other specialised languages.

At a formal level, a *structural operational computation-semantics* [Hennessy 1990], also called *small-step* operational semantics in the literature [Slonneger et al. 1995], was presented to precisely define the language, and provide for an unambiguous definition of the various policy constructs. The semantics define the execution of policy specifications including instantiation and storing in the domain service, as well as runtime execution of policies.

Ponder is not simply a specification approach, but rather a complete management framework. We have presented a runtime object-based model for deploying and managing Ponder policies in a distributed system which supports: (i) the enforcement of authorisation policies using multiple heterogeneous access control mechanisms, (ii) the enforcement of obligation and refrain policies using generic policy management components whose implementation is largely independent of the underlying system, and (iii) enforcement of composite policy structures (groups and roles). Role-based management includes assignment of users to roles, activation and deactivation of assigned roles, as well as dynamic enforcement of policies specified in roles.

The architecture presented handles dynamic changes to domain structures and dynamic adaptation of the system to such changes. Domain membership is dynamic and objects can be added to, or removed from, a domain as needed. If an object is added to a domain, the policies that currently apply to that domain will subsequently apply to the newly added object. Conversely, if an object is removed from a domain, the policies for the domain will no longer be applied to the object. Objects can thus be managed either by (i) modifying the policies applying to the domains of which the objects are members or (ii) adding or removing the objects from domains to which specific policies apply.

The deployment model cleanly separates the dissemination and management of policies from their enforcement and can act as a reference model for other implementations. Although it is strongly

influenced by the Ponder policy specification language, we believe that other policy-based systems can benefit from adopting a deployment model that is similar to ours.

As a proof of concept, we have presented an integrated implementation of a management toolkit based on the Ponder language. We have described what we consider to be the minimum requirements for a toolkit for policy life-cycle management – a high-level language and graphical editor for specifying policies, a compiler for translating policies into formats used by enforcement components targeted to specific platforms, and tools to support an automated approach to dynamically deploying, enabling, disabling and replacing policies in the distributed components that will interpret them. The management tools are integrated around a domain browser to access the domain service and support specification, dissemination and control of policy. It has proven straightforward to provide a Java-based implementation of the deployment model and toolkit.

A Ponder compiler is written using SableCC, with experimental backends for generating XML syntax; Windows NT/2000 access control, and firewall rules; Linux ipchains and PAM modules (pluggable authentication modules); Java security policies. It has been quite easy to produce back-ends for the Ponder compiler, targeted to different platforms, and the current experience demonstrates that the design and implementation of the compiler have been successful. Work is in progress to produce a backend for DiffServ quality of service management as well as a backend that compiles parts of Ponder to field programmable gate arrays (FPGAs). A different implementation that maps Ponder into an agent-based system is being developed at the University of Bologna [Corradi et al. 2001]. The Ponder compiler and policy editor are available in the public domain (from *http://www-dse.doc.ic.ac.uk/research/policies/software/*), and have been used by a number of organisations such as Sun Microsystems Inc., Cisco, Nortel Networks, Siemens AG, Alcatel and Hitachi Europe Ltd., as well as many research institutions.

Throughout this thesis we have used examples, mostly related to security management, to describe the concepts of the language. The framework is however applicable to a wider range of management applications. Current and past projects are investigating the use of the Ponder framework to manage QoS in a DiffServ-enabled network, and we have also experimented with specifying policies for a storage management system. In [Dulay et al. 2001b] the applicability of Ponder in managing distributed agent systems is examined, and in [Lupu et al. 2000b] Ponder is used to realise many enterprise viewpoint concepts.

9.2 Future Work

Several issues for further work have been identified through the discussion in Chapter 8. We list the most important of them in this section and extend them with a few additional issues.

9.2.1 Language Specification

The language specification leaves room for future additions, one of which is the syntax of relationship structures. Relationships must be extended with interaction protocols to specify the permitted and required message exchanges between occupants of the roles. In addition, the current inheritance mechanism for composite policy types does not allow multiple inheritance for relationships and management structures. Future work could be directed towards examining the inheritance model for possible extensions.

Further investigation of possible sub-types of meta-policies to cover specific classes of constraints such as concurrency constraints and user-role assignment constraints, or to specify policy priorities, could result in a more restricted syntax for specifying constraints. In addition, the formal specification of the language must be extended with a type-inference system for compile-time type checking of policy specifications, as well as with the operational semantics of meta-policies.

Application of the language in more complex management scenarios could identify the need to extend the library objects (i.e. Time, Timer, Domain) with additional functions or with new library objects. Similarly, application of delegation policies in future scenarios is also needed to determine whether the syntax for delegation policies needs to be extended.

An interesting aspect to examine would be a graphical notation for specifying Ponder policies. Graphical approaches make it easier for non-technical users to use the language, and enable faster creation of large-scale policy specifications through simple diagrams created with the aid of editing tools. Special symbols can be used to represent the various types of policies in Ponder and connectors can be applied to group policies in composite structures or create associations between the policy objects. Tool support can then make the translation or the extension of the graphical model into a text version easy, simplifying the specification of policy for large systems.

9.2.2 Deployment Model and Implementation

The model needs to be developed further, and an interesting requirement is to cater for consistent updates in the presence of concurrent operations on the domains and policies in the system. Although many operations can be performed in parallel, the detection of possible conflicts is difficult and potentially very slow. An interesting aspect that may help here, is that the policy

system is itself subject to policy control and policies may be written and enforced to prevent undesirable actions on the domains that hold enabled policy objects and on the policy objects themselves.

It would be interesting to examine the possibility of propagating changes to already deployed policies using a runtime API to dynamically modify policies and in particular composite policies such as roles. The API will allow administrators to extend roles with new policies or remove policies from specific roles, which are already stored and deployed in the system with users assigned to them.

Future work needs to evaluate more closely the degree to which the same authorisation policy can be enforced on a variety of security architectures and platforms. Additional work is also needed on the enforcement of delegation policies, which assume an authorisation service with a specific interface as described in Section 5.4. In addition, the current framework does not cover the enforcement of meta-policies. Meta-policies must be supported by back-ends and the architecture must be extended with deployment and management of meta-policies.

This thesis does not cover issues relating to refinement of high-level enterprise goals, service level agreements or trust specifications to Ponder. This process can be considered a requirements engineering aspect to the policy life-cycle and is being addressed in a related project at Imperial College. Furthermore, future work needs to concentrate on the interoperability of policies across administrative domains, an area not adequately covered in the literature, and examine the use of Ponder policies and the proposed framework in solving this problem. Research is needed on defining interfaces for the exchange of policies across organisational networks.

The implementation can be improved as discussed in Section 8.3. Future implementations will need to evaluate and integrate existing monitoring solutions (e.g. [Micromuse 2002]) to the current system. The investigation must also extend to event-service implementations which better satisfy the requirements identified in Section 8.3. Finally, the current implementation using a single LDAP server is centralised and needs to be extended with multiple servers, and better fault-tolerance and replication of data.

9.2.3 Management Toolkit

The management toolkit needs to be developed further. Tools for the refinement of high-level policy specifications (goals, SLA's, etc) are a primary objective. As the refinement process is not expected to be fully automated this will require interactive tool support. Tools for policy analysis and reasoning also need to be developed. The possibility for providing further analysis by simulating the execution of policies is an interesting aspect. An integrated environment for

animating the simulation and viewing the results will be part of such a task. Finally, integrating all management tools around a generic domain browser to give a common ‘look and feel’ to the toolkit is an interesting approach which needs to be developed further in the Ponder framework.

9.3 Closing Remarks

The problem of policy specification and deployment is complex and has been addressed at Imperial College for the past 10 years resulting in significant theoretical advances in the understanding of the problem and its solutions. In this thesis we have shown how the concepts resulting from the past experience are utilised; we have refined and elaborated those concepts into an integrated policy-based management framework and an implementation that is currently being used outside Imperial College. This process has presented a significant engineering challenge, which was tackled successfully in this thesis, and from which other work in the area of policy-based management can benefit. The policy framework includes the design of a policy specification language and an architecture for the deployment and enforcement of policies, both of which are novel and constitute the main contributions of this thesis.

Bibliography

Abadi, M., M. Burrows, B. Lampson and G. Plotkin (1993). *A Calculus for Access Control in Distributed Systems*. ACM Transactions on Programming Languages and Systems, vol. 4(15), pp. 706-734, September 1993.

Abrams, M. D. (1993). *Renewed Understanding of Access Control Policies*. In Proceedings of the 16th National Computer Security Conference, Baltimore, Maryland, U.S.A., pp. 87-96, 20-23 September 1993.

Acharya, A. and G. Edjlali (1998). *History-based Access Control for Mobile Code*. In Proceedings of the 5th ACM conference on Computer and Communications Security, San Francisco, California, USA, ACM Press New York, 3-5 November 1998.

Ahn, G.-J. and R. Sandhu (1999). *The RSL99 Language for Role-Based Separation of Duty Constraints*. In Proceedings of the Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, ACM Press, pp. 43-54, 28-29 October 1999.

Anderson, R., J.-H. Lee and F. Stajano (2001). *Security Policies*. In Book Chapter in Advances in Computers, Academic Press. vol. 55.

Anderson, R. J. (1996). *A Security Policy Model for Clinical Information Systems*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, U.S.A., pp. 30-43, May 6-8, 1996.

Apache (2001) Software Foundation, *The Byte Code Engineering Library Project*, *available from http://jakarta.apache.org/bcel/index.html*, December 2001.

Arnold, D., J. Boot, M. Henderson, T. Phelps and B. Segall (2001), *Elvin: Content-Addressed Messaging Client Protocol*, *Internet Draft - work in process*, June 2001.

Arnold, K., B. O'Sullivan, R. W. Scheifler, J. Waldo and A. Wollrath (1999). *The Jini™ Specification*, Addison -Wesley Longoman, Inc., June 1999.

Bacon, J., K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel and MarkSpiteri (2000). *Generic Support for Distributed Applications*. IEEE Computer, vol. 33(3), pp. 68-76, March 2000.

Baldwin, R. W. (1990). *Naming and Grouping Privileges to Simplify Security Management in Large Databases*. In Proceedings of the IEEE Symposium on Computer Security and Privacy, Oakland, California, U.S.A., IEEE CS Press, pp. 61-70, May 1990.

Barker, S. (2000). *Security Policy Specification in Logic*. In Proceedings of the International Conference on Artificial Intelligence (ICAI00), Las Vegas, Nevada, USA, pp. 143-148, 26-29 June 2000.

Barker, S. and A. Rosenthal (2001). *Flexible Security Policies in SQL*. In Proceedings of the Fifteenth Annual IFIP WG 11.3 Working Conference on Database and Application Security, Niagara on the Lake, Ontario, Canada, 15-18 July 2001.

Barkley, J., R. Kuhn, L. Rosenthal, M. Skall and A. Cincotta (1998). *Role-Based Access Control for the Web*. In Proceedings of the CALS Expo International & 21st Century Commerce 1998: Global Business Solutions for the New Millennium, Long Beach, CA, USA, 26-29 October 1998.

Barkley, J. F., K. Beznosov and J. Uppal (1999). *Supporting Relationships in Access Control Using Role Based Access Control*. In Proceedings of the Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, pp. 55-65, October 28-29, 1999.

Bertino, E., C. Bettini, E. Ferrari and P. Samarati (1998). *An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning*. ACM Transactions on Database Systems, vol. 23(3), pp. 231-285, September 1998.

Bertino, E., P. Bonatti and E. Ferrari (2000). *TRBAC: A Temporal Role-Based Access Control Model*. In Proceedings of the 5th ACM Workshop of Role-Based Access Control, Berlin, Germany, pp. 21-30, 26-28 July 2000.

Beznosov, K. and Y. Deng (1999). *A Framework for Implementing Role-Based Access Control Using CORBA Security Service*. In Proceedings of the Fourth ACM Workshop on Role-Based Access Control, ACM Press, pp. 19-30, 28-29 October 1999.

Blaze, M., J. Feigenbaum, J. Ioannidis and A. D. Keromytis (1999). *The Role of Trust Management in Distributed Systems Security*. In Secure Internet Programming: Security Issues for Mobile and Distributed Objects. New York, NY, USA, Springer-Verlag, pp. 185 - 210.

Blaze, M., J. Feigenbaum and A. Keromytis (1998). *Keynote: Trust Management for Publc-Key Infrastructures*. In Proceedings of the Security Protocls International Workshop, Cambridge, England, Springer-Verlag LNCS, pp. 59 - 63, April 1998.

Bos, H. (1999). *Application-Specific Policies: Beyond the Domain Boundaries*. In Proceedings of the Sixth IFIP/IEEE International Symposium on Intergrated Network Management (IM'99), Boston, MA, USA, 24-28 May 1999.

Burgess, M. (1995). *A Site Configuration Engine*. USENIX Computing systems, vol. 8(3), Summer 1995.

Burgess, M. (2001). *Recent Developments in CfEngine*. In Proceedings of the Unix NL Conference, The Hague, 2001.

Burgess, M. and F. E. Sandnes (2001). *Predictable Configuration Management in a Randomized scheduling Framework*. In Proceedings of the IEEE/IFIP Workshop on Distributed Systems Operations and Management (DSOM '2001), Nancy, France, 15-17 October 2001.

Cengarle, M. V. and A. Knapp (2001). *On the Expressive Power of Pure OCL*. Technical Report 0101, Ludwig-Maximilians-Universität München, Munich, Germany.

Chen, F. and R. S. Sandhu (1995). *Constraints for Role-Based Access Control*. In Proceedings of the First ACM/NIST Role Based Access Control Workshop, Gaithersburg, Maryland, USA, ACM Press, November 1995.

Chess, D. M. (1998). *Security Issues in Mobile Code Systems*. In Mobile Agents and Security. G. Vigna eds, Springer-Verlag, pp. 1-14, 18 June 1998.

Chomicki, J., J. Lobo and S. Naqvi (2000). *A Logic Programming Approach to Conflict Resolution in Policy Management.* In Proceedings of the 7th International Conference in Principles of Knowledge Representation and Reasoning, Breckenridge, Colorado, USA, Morgan Kaufmann Publishers, pp. 121-132, April 2000.

Clark, D. D. and D. R. Wilson (1987). *A Comparison of Commercial and Military Computer Security Policies*. In Proceedings of the IEEE Symposium on Security and Privacy, 1987.

Corradi, A., N. Dulay, R. Montanari and C. Stefanelli (2001). *Policy-Driven Management of Agent Systems*. In Proceedings of the Policy Workshop 2001, HP Labs, Bristol, UK, Springer-Verlag, 29-31 January 2001.

Corradi, A., R. Montanari, E. Lupu, M. Sloman and C. Stefanelli (2000). *A Flexible Access Control Service for Java Mobile Code*. In Proceedings of the Annual Computer Security Applications Conference (ACSAC 2000), New Orleans, Louisiana, USA, IEEE Press, 11-15 December 2000.

Damianou, N., N. Dulay, E. Lupu and M. Sloman (2000a). *Managing Security in Object-based Distributed Systems using Ponder*. In Proceedings of the 6th Open European Summer School (Eunice 2000), Enchede, The Netherlands, 13-15 September 2000.

Damianou, N., N. Dulay, E. Lupu and M. Sloman (2000b). *Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. The Language Specification - Version 2.3*. Research Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, London, 20 October 2000.

Damianou, N., N. Dulay, E. Lupu and M. Sloman (2001). *The Ponder Policy Specification Language*. In Proceedings of the Policy Workshop 2001, HP Labs, Bristol, UK, Springer-Verlag, 29-31 January 2001.

Damianou, N., T. Tonouchi, N. Dulay, E. Lupu and M. Sloman (2002). *Tools for Domain-based Policy Management of Distributed Systems*. In Proceedings of the Network Operations and Management Symposium (NOMS 2002) (To Appear), Frorence, Italy, 15-19 April 2002.

DMTF (1999a) Distributed Management Task Force, Inc., *Common Information Model (CIM) Specification, version 2.2, available from http://www.dmtf.org/spec/cims.html*, 14 June 1999.

DMTF (1999b) Distributed Management Task Force, Inc., *Specification for the Representation of CIM in XML, version 2.0, available from http://www.dmtf.org/download/spec/xmls/CIM_XML_Mapping20.php*, 20 July 1999.

Drossopoulou, S. and S. Eisenbach (1998), *Towards an Operational Semantics and Proof of Type Soundness for Java, available from http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html*, March 1998.

Dulay, N., N. Damianou, E. Lupu and M. Sloman (2001b). *A Policy Language for the Management of Distributed Agents*. In Proceedings of the Invited paper. Agent Oriented Software Engineering Workshop (AOSE) 2001, Montreal, Canada, 29 May 2001.

Dulay, N., E. Lupu, M. Sloman and N. Damianou (2001a). *A Policy Deployment Model for the Ponder Language*. In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM'2001): Integrated Management Strategies for the New Millennium, Seattle, Washington, USA, 14-18 May 2001.

Gagnon, E. (1998). *SableCC, An Object-Oriented Compiler Framework.* MSc Thesis, School of Computer Science, McGill University. Montreal, Canada, March 1998.

Gligor, V. (1995). *Characteristics of Role Based Access Control*. In Proceedings of the First ACM/NIST Role Based Access Control Workshop, Gaithersburg, Maryland, USA, ACM Press, November 1995.

Grandison, T. and M. Sloman (2000). *A Survey of Trust in Internet Applications*. IEEE Communications Surveys and Tutorials, vol. 3(4), Oct-Dec 2000.

Hami, A., J. Howse and S. Kent (1998). *Interpreting the Object Constraint Language*. In Proceedings of the Asia Pacific Conference in Software Engineering, IEEE Press, July 1998.

Hayton, R. J., J. M. Bacon and K. Moody (1998). *Access Control in an Open Distributed Environment*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, U.S.A., pp. 3-14, May 1998.

Hegering, H.-G., S. Abeck and B. Neumair (1999). *Integrated Management of Network Systems: Concepts, Architectures and Their Operational Application*, Morgan Kaufmann Publishers, August 1999.

Hennessy, M. (1990). *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*, John Wiley and Sons, 1990.

Herzberg, A., Y. Mass, J. Michaeli, D. Naor and Y. Ravid (2000). *Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, USA, 14-17 May 2000.

Heydon, A., M. W. Maimone, J. D. Tygar, J. M. Wing and A. M. Zarenski (1990). *Miro: Visual Specification of Security*. IEEE Transactions on Software Engineering, vol. 16(10), pp. 1185-1197, October 1990.

Hine, J., W. Yao, J. Bacon and K. Moody (2000). *An Architecture for Distributed OASIS Services*. In Proceedings of the Middleware 2000, New York, USA, Lecture Notes in Computer Science, Springer-Verlag, pp. 107-123, 4-8 April 2000.

Hitchens, M. and V. Varadharajan (2000). *Elements of a Language for Role Based Access Control*. In Proceedings of the Information Security for Global Information Infrastructures, IFIP TC11 Sixteenth Annual Working Conference on Information Security, Beijing, China, Kluwer, pp. 371-380, 22-24 August 2000.

Hitchens, M. and V. Varadharajan (2001). *Tower: A Language for Role Based Access Control*. In Proceedings of the Policy Workshop 2001, HP Labs, Bristol, UK, Springer-Verlag, 29-31 January 2001.

Hoagland, J. A., R. Pandey and K. N. Levitt (1998). *Security Policy Specification Using a Graphical Approach*. Technical report CSE-98-3, UC Davis Computer Science Department, 22 July 1998.

Howard, J. D. (1997). *An Analysis Of Security Incidents On The Internet 1989 - 1995.* PhD Thesis, Department of Engineering and Public Policy, Carnegie Mellon University. Pittsburgh, Pennsylvania 15213 USA, 7 April 1997.

Hyland, P. C. and R. Sandhu (1998). *Management of Network Security Applications*. In Proceedings of the 21st National Information Systems Security Conference (NISSC), Hyatt Regency Crystal City, Arlington, Virginia, USA, 5 October 1998.

Inxight Software Inc. (2001), *Inxight Star Tree Software Development Kit*, *available from http://www.inxight.com*.

ISO/IEC (1989) 7498-4, *Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management Framework*.

ISO/IEC (1999) JTC1/SC7/WG3-3N34, *Information Technology - Open Distributed Processing Reference Model - Enterprise Viewpoint*, *ISO ISO/IEC 15414 | ITU-T Recommendation X.911*, 9 January 1999.

Jackson, D. (2000). *Alloy: A Lightweight Object Modelling Notation*, Laboratory for Computer Science, Massachusetts Institute of Technology, Boston, 28 July 2000.

Jackson, D., I. Schechter and I. Shlyakhter (2000). *Alcoa: the Alloy Constraint Analyzer*. In Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000.

Jajodia, S., P. Samarati, M. L. Sapino and V. S. Subrahmanian (2000). *Flexible Support for Multiple Access Control Policies*. ACM Transactions on Database Systems, vol. 26(2), pp. 214-260, June 2001.

Jajodia, S., P. Samarati and V. S. Subrahmanian (1997). *A Logical Language for Expressing Authorisations*. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 31-42, May 4-7, 1997.

Janson, P. A. (1994). *Security for Management and Management of Security*. Chapter 15 in Network and Distributed Systems Management (Sloman, 1994ed), pp. 403-430.

Jude, M. (2001), *Policy-based Management: Beyond The Hype. In Business Communications Review*, *available from http://www.bcr.com/bcrmag/2001/03/p52.asp*, pp. 52-56, March 2001.

Keromytis, A. D., S. Ioannidis, M. Greenwald and J. M. Smith (2001). *Scalable Security Policy Mechanisms*. Technical Report, MS-CIS-01-05, University of Pennsylvania CIS Dept., January 2001.

Kohli, M. and J. Lobo (1999). *Policy Based Management of Telecommunication Networks*. In Proceedings of the Policy Workshop 1999, HP Labs, Bristol, UK, 15-17 November 1999.

Lamping, J., R. Rao and P. Pirolli (1995). *A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI 95), Available from http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/jl_bdy.htm.

Langsford, A. (1994). *OSI Management Model and Standards*. Chapter 4 in Network and Distributed Systems Management (Sloman, 1994ed), pp. 69-93.

Lehman, T. J., S. W. McLaughry and P. Wyckoff (1999). *TSpaces: The Next Wave*. In Proceedings of the Hawaii International Conference on System Sciences (HICSS-32), Hawaii, January 1999.

Lobo, J., R. Bhatia and S. Naqvi (1999). *A Policy Description Language*. In Proceedings of the Sixteenth National Conference on Artificial Intelligence Eleventh Innovative Applications of AI Conference, Orlando, Florida, USA, 18-22 July 1999.

Lupu, E., N. Dulay, N. Damianou and M. Sloman (2000a). *Structuring Devolved Responsibilities in Network and Systems Management*. In Networking and Information Systems Journal, Issue on Multimedia Management. (Invited Paper), vol. 3(2/ 2000), pp. 261-277.

Lupu, E., M. Sloman, N. Dulay and N. Damianou (2000b). *Ponder: Realising Enterprise Viewpoint Concepts*. In Proceedings of the 4th International Enterprise Distributed Object Computing Conference (EDOC 2000), Makuhari, Japan, 25-28 September 2000.

Lupu, E. C. (1998). *A Role-Based Framework for Distributed Systems Management.* PhD Thesis, Department of Computing, Imperial College. London, U. K., July 1998.

Lupu, E. C. and M. S. Sloman (1997a). *Reconciling Role Based Management and Role Based Access Control*. In Proceedings of the Second ACM/NIST Role Based Access Control Workshop, Fairfax, Virginia, USA, ACM Press, November 1997.

Lupu, E. C. and M. S. Sloman (1997b). *Towards a Role Based Framework for Distributed Systems Management*. Journal of Network and Systems Management, vol. 5(1), pp. 5-30, 1997.

Lupu, E. C. and M. S. Sloman (1999). *Conflicts in Policy-Based Distributed Systems Management*. In IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management, vol. 25(6), pp. 852-869, November 1999.

Mansouri-Samani, M. and M. S. Sloman (1997). *GEM: A Generalised Event Monitoring Language for Distributed Systems*. Distributed Systems Engineering Journal, vol. 4(2), pp. 96-108, June 1997.

Marriott, D. A. (1997). *Policy Service for Distributed Systems.* PhD Thesis, Department of Computing, Imperial College. London, U. K., October 1997.

Marriott, D. A. and M. S. Sloman (1996). *Implementation of a Management Agent for Interpreting Obligation Policy*. In Proceedings of the 7th IFIP/IEEE International Workshop on Distributed Systems Operations Management (DSOM'96), L' Aquila, Italy, October 1996.

Martin-Flatin, J.-P., S. Znaty and J.-P. Hubaux (1999). *A Survey of Distributed Enterprise Network and Systems Management Paradigms*. Journal of Network and Systems Management, vol. 7(1), pp. 9-26, September 1999.

Michael, J. B., V. L. Ong and N. C. Rowe (2001). *Natural-Language Processing Support for Developing Policy-Governed Software Systems*. In Proceedings of the 39th International Conference on Technology for Object-Oriented Languages and Systems, Santa Barbara, California, IEEE Computer Society Press, pp. 263-274, July 2001.

Micromuse (2002), *Netcool Solutions*, Available from http://www.micromuse.com/products/, 2002.

Minsky, N. H. and A. D. Lockman (1985). *Ensuring Integrity by Adding Obligations to Privileges*. In Proceedings of the 8th International Conference on Software Engineering, London, U.K., pp. 92-102, August 1985.

Moffett, J. D. (1998). *Control Principles and Role Hierarchies*. In Proceedings of the Third ACM/NIST Role Based Access Control Workshop, Fairfax, Virginia, USA, ACM Press, 22-23 October 1998.

Moffett, J. D. and M. S. Sloman (1993). *Policy Hierarchies for Distributed Systems Management*. IEEE JSAC Special Issue on Network Management, vol. 11(9), pp. 1404-1414, December 1993.

Monday, P. and W. Connor (2001). *The Jiro Technology Programmer's Guide and Federated Management Architecture*, Addison-Wesley, 2001.

Mont, M. C., A. Baldwin and C. Goh (1999a). *Role of Policies in a Distributed Trust Framework*. Technical Report HPL-1999-104, Extended Enterprise Laboratory, HP Laboratories, Bristol, UK, 16th Sepember 1999.

Mont, M. C., A. Baldwin and C. Goh (1999b). *POWER Prototype: Towards Integrated Policy-Based Management*. Technical Report HPL-1999-126, Extended Enterprise Laboratory, HP Laboratories, Bristol, UK, 18 October 1999.

Moore, B., E. Ellesson, J. Strassner and A. Westerinen (2001), *Policy Core Information Model - Version 1 Specification, RFC 3060, available from http://www.ietf.org*, February 2001.

Myers, A. C. and B. Liskov (1997). *A Decentralized Model for Information Flow Control*. In Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP), Saint-Malo, France, pp. 129-142,, October 1997.

Netscape (2000), *Netscape Directory Server version 4.2, available from http://home.netscape.com/directory/v4.0/*.

OASIS (2001) (Organization for the Advancement of Structured Information Standards), *XACML language proposal, version 0.8, available from http://www.oasis-open.org/committees/xacml*, 10 January 2002.

OMG (1999a) Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.3.1*, October 1999.

OMG (1999b) Object Management Group, *Object Constraint Language Specification, version 1.3, Chapter 7 in OMG Unified Modelling Language Version 1.3*, June 1999.

OMG (2001) Object Management Group, *CORBA Security Service Specification, version 1.7*, March 2001.

Ortalo, R. (1998). *A Flexible Method for Information System Security Policy Specification*. In Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS 98), Louvain-la-Neuve, Belgium, Springer-Verlag, September 1998.

Ribeiro, C., A. Zuquete and P. Ferreira (2001a). *SPL: An access control language for security policies with complex constraints*. In Proceedings of the Network and Distributed System Security Symposium (NDSS'01), San Diego, California, February 2001.

Ribeiro, C., A. Zuquete and P. Ferreira (2001b). *Enforcing Obligation with Security Monitors*. In Proceedings of the Third International Conference on Information and Communications Security (ICICS 2001), Xian, China, 13-16 November 2001.

Richters, M. and M. Gogolla (1998). *On Formalising the UML Object Constraint Language OCL*. In Proceedings of the 17th International Conference on Conceptual Modelling (ER '98), Singapore, Springer-Verlag, November 1998.

Rowley, A. (1998). *A Security Architecture for Distributed Groupware.* PhD Thesis, Department of Computer Science, Queen Mary and Westfield College. London, U.K., September 1998.

Ryan, V., S. Seligman and R. Lee (1999), *Schema for Representing Java (tm) Objects in an LDAP Directory, RFC 2713*, October 1999.

Samarati, P., E. Bertino, S. D. C. D. Vimercati and E. Ferrari (1998). *Exception-Based Information Flow Control in Object-Oriented Systems*. ACM Transactions on Information and System Security, vol. 1(1), pp. 26-65, November 1998.

Samarati, P. and S. Vimercati (2000). *Access Control: Policies, Models, and Mechanisms*. In Foundations of Security Analysis and Design (Tutorial Lectures). R. Focardi and R. Gorrieri eds, Springer -Verlag, pp. 137-196, September 2000.

Sandhu, R., D. Ferraiolo and R. Kuhn (2000). *The NIST Model for Role-Based Access Control: Towards A Unified Standard*. In Proceedings of the 5th ACM Workshop on Role-Based Access Control, Berlin, Germany, pp. 47-61, 26-28 July 2000.

Sandhu, R. and P. Samarati (1994). *Access Control: Principles and Practice*. IEEE Communications Magazine, vol. 32(9), pp. 40-48.

Sandhu, R. S. (1998). *Role Activation Hierarchies*. In Proceedings of the Third ACM/NIST Role Based Access Control Workshop, Fairfax, Virginia, USA, ACM Press, 22-23 October 1998.

Sandhu, R. S., E. J. Coyne, H. L. Feinstein and C. E. Youman (1996). *Role-Based Access Control Models*. IEEE Computer, vol. 29(2), pp. 38-47.

Sandhu, R. S. and P. Samarati (1994). *Authentication, Access Control, and Intrusion Detection*. Part of the paper appeared under the title "Access Control: Principles and Practice" in IEEE Communications, vol. 32(9), pp. 40-48, 1994.

Segall, B. and D. Arnold (1997). *Elvin has left the building: A publish/subscribe notification service with quenching*. In Proceedings of the Queensland AUUG Summer Technical Conference, Brisbane, Australia, available from http://www.dstc.edu.au/Elvin/doc/papers/auug97/AUUG97.html.

Sloman, M. and K. Twidle (1994a). *Domains: A Framework for Structuring Management Policy*. In Chapter 16 in Network and Distributed Systems Management (Sloman, 1994ed), pp. 433-453.

Sloman, M. S. (1994b). *Policy Driven Management for Distributed Systems*. Journal of Network and Systems Management, vol. 2(4), pp. 333-360, December 1994.

Sloman, M. S., Jeff Magee, K. Twidle and J. Kramer (1993). *An Architecture For Managing Distributed Systems*. In Proceedings of the Fourth IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, Portugal, IEEE Computer Society Press, pp. 40-46, 22-24 September 1993.

Slonneger, K. and B. Kurtz (1995). *Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach*, Addison-Wesley Publishing Company, 1995.

Snir, Y., Y. Ramberg, J. Strassner, R. Cohen and B. Moore (2001), *Policy QoS Information Model*, *available from www.ietf.org/internet-drafts/draft-ietf-policy-qos-info-model-04.txt*, November 2001.

Steen, M. W. A. and J. Derrick (1999). *Formalising ODP Enterprise Policies*. In Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC '99), University of Mannheim, Germany, IEEE Publishing, September 1999.

Steen, M. W. A. and J. Derrick (2000). *ODP Enterprise Viewpoint Specification*. Computer Standards and Interfaces, vol. 22, pp. 65-189, September 2000.

Stone, G. N., B. Lundy and G. G. Xie (2001). *Network Policy Languages: A Survey and a New Approach*. IEEE Network, vol. 15(1), pp. 10-21, January 2001.

Strassner, J., E. Ellesson, B. Moore and R. Moats (2002), *Policy Core LDAP Schema*, *IETF Internet draft work in progress, available from http://www.ietf.org*, January 2002.

Strassner, J. and S. Schleimer (1998), *Policy Framework Definition Language (version 00)*, *IETF Internet draft work in progress, available from http://www.ietf.org*, 17 November 1997.

Sun (1999a) Microsystems, Inc, *Getting Started with the Java Dynamic Management Kit 4.0*, *available from http://java.sun.com/docs/index.html*, December 1999.

Sun (1999b) Microsystems, Inc., *Java Management Extensions Instrumentation and Agent Specification, v1.0*, *available from http://java.sun.com/docs/index.html*, December 1999.

Sun (1999c) Microsystems, Inc, *Remote Method Invocation Specification*, *available from http://java.sun.com/docs/index.html*, September 1999.

Sun (1999d) Microsystems, Inc., *Java Naming and Directory Interface, Application Programming Interface, available from http://java.sun.com/docs/index.html*, July 1999.

Sun (2000) Microsystems, Inc., *Java Message Service version 1.0.2*, *available from http://java.sun.com/docs/index.html*, December 2000.

Tennenhouse, D. L., J. M. Smith, W. D. Sincoskie, D. J. Wetherall and G. J. Minden (1997). *A Survey of Active Network Research*. IEEE Communications Magazine, vol. 35(1), pp. 80-86, 1 January 1997.

Thomas, R. K. (1997). *Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments*. In Proceedings of the Second ACM/NIST Role Based Access Control Workshop, Fairfax, Virginia, USA, ACM Press, pp. 13-19, November 1997.

Thomsen, D., D. O'Brien and J. Bogle (1998). *Role Based Access Control Framework for Network Enterprises*. In Proceedings of the 14th Annual Computer Security Applications Conference, December 1998.

Tonouchi, T. (2001), *Hyperbolic Domain Browser Implementation*, *available from http://www-dse.doc.ic.ac.uk/policies/software.html*, October 2001.

Varadharajan, V. and P. Allen (1996). *Joint Action Based Authorisation Schemes*. ACM Operating Systems Review, vol. 30(3), pp. 32-45, July 1996.

Vaziri, M. and D. Jackson (1999). *Some Shortcomings of OCL, the Object Constraint Language of UML*. Response to Object Management Group's Request for Information on UML 2.0, MIT Laboratory for Computer Science, 7 December 1999.

Verma, D., M. Beigi and R. Jennings (2001). *Policy Based SLA Management in Enterprise Networks*. In Proceedings of the Policy Workshop 2001, HP Labs, Bristol, UK, Springer-Verlag, 29-31 January 2001.

Verma, D. C. (2001). *Policy-Based Networking: Architecture and Algorithms*, New Riders Publishing, 2000.

Virmani, A., J. Lobo and M. Kohli (2000). *Netmon: network management for the SARAS softswitch*. In Proceedings of the 2000 IEEE/IFIP Network Operations and Management Seminar (NOMS 2000), Hawaii, April 2000.

Wahl, M., T. Howes and S. Kille (1997a), *Lightweight Directory Access Protocol (v3)*, *RFC 2251, available from http://www.ietf.org*, December 1997.

Wahl, M., T. Howes and S. Kille (1997b), *Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions*, *RFC 2252, available from http://www.ietf.org*, December 1997.

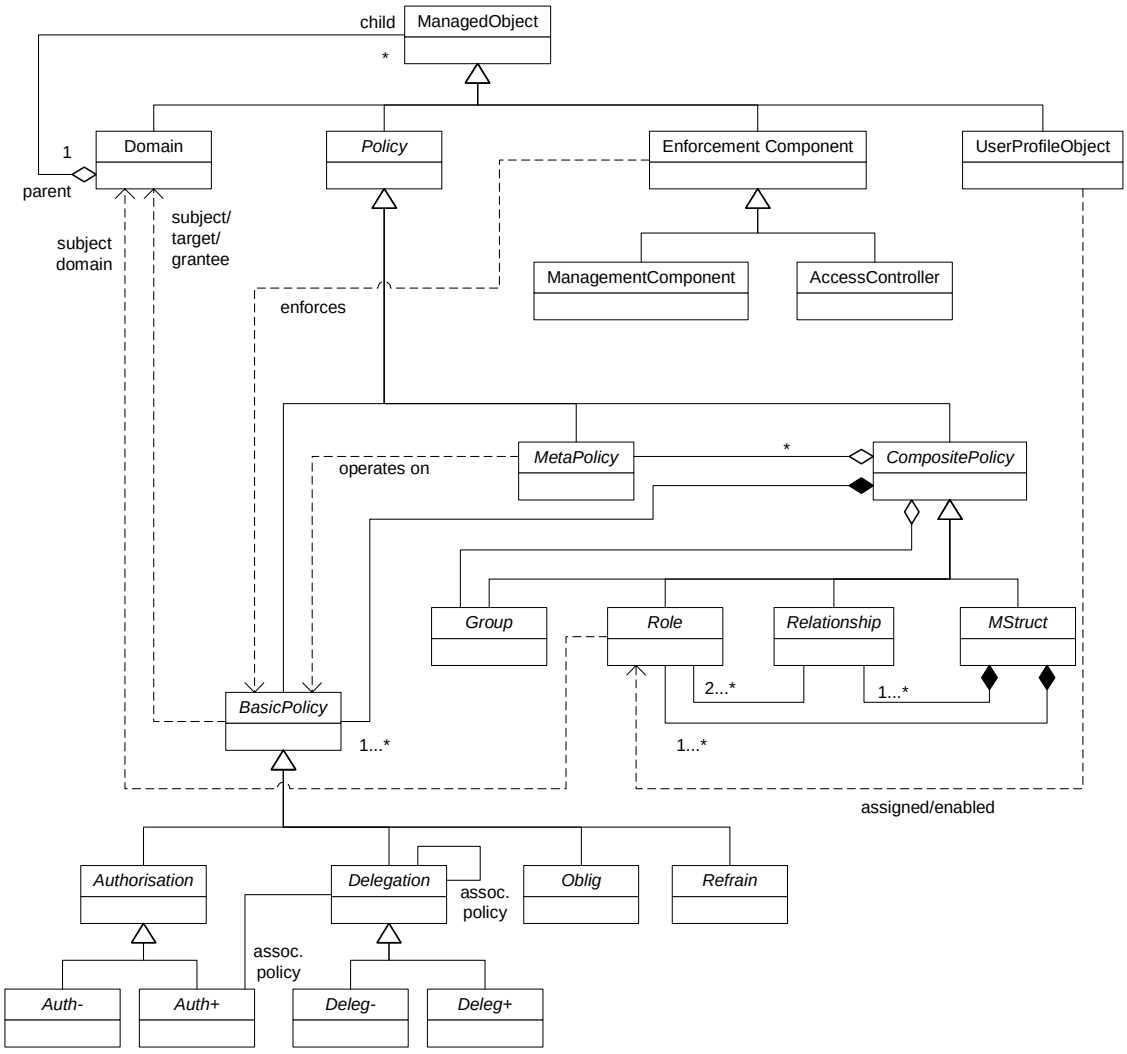
Weis, R. (1994a). *Policy Definition and Classification: Aspects, Criteria and Examples*. In Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations & Management, Toulouse, France, 10-12 October 1994.

Yialelis, N. (1996). *Domain-Based Security for Distributed Object Systems.* PhD Thesis, Department of Computing, Imperial College. London, U. K., August 1996.

Appendix A

Information Model

A.1 Class Diagram



Appendix B

Syntax Specification

B.1 Grammar

This is the syntax of the policy language specified in SableCC [Gagnon 1998].

```
Package ponderToolkit.compiler.syntax;   // Root Java package for generated Java files

/************************************************************************
   HELPERS
************************************************************************/
Helpers

   all =              [0 .. 127];
   digit =            ['0' .. '9'];
   non_digit =        ['_' + [['a' .. 'z'] + ['A' .. 'Z']]];
   u_case =           ['A' .. 'Z'];
   l_case =           ['a' .. 'z'];
   letter =           l_case | u_case;
   cr =               13;
   lf =               10;
   tab =              9;
   eol =              cr | lf | cr lf;
   non_eol =          [all - [10 + 13]];
   not_star =         [all - '*'];
   not_quotes =       [all - '"'];
   not_star_slash =   [not_star - '/'];
   not_brace =        [all - '}'];
   path_seq =         (non_digit ( digit | non_digit)*);
   l_comment =        '//' non_eol* eol; /* or eof for that matter */
   c_comment =        '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
   not_gt =           [all - '>'];
   spec_chars =       '<<<' (not_gt* (('>'|'>>') not_gt)* not_gt*)+ '>>>';
   digit_seq =        digit+;
   fractional_seq =   digit_seq? '.' digit_seq ;
   sign =             '+' | '-';
   exponent_part =    ('e' | 'E') sign? digit_seq;
   float_seq =        fractional_seq exponent_part? |      digit_seq exponent_part ;
   integer_seq =      digit digit*;
   string_seq =       '"' (not_quotes | '\"')* '"';
   boolean_seq =      'TRUE' | 'FALSE' | 'true' | 'false' ;

/************************************************************************
   STATES
************************************************************************/
States
   normal,     // default state. When not in an ocl-expression.
   ocl,        // entered when in an ocl-expression
   pre_ocl;    // entered when you are about to enter an ocl-expression

/************************************************************************
   TOKENS
************************************************************************/
Tokens

   /****************************** Keywords *******************************/
   auth_plus =     'auth+';
   auth_minus =    'auth-';
```

```
boolean =       'boolean';
catch =         'catch';
deleg_plus =    'deleg+';
deleg_minus =   'deleg-';
do =            'do';
domain =        'domain';
extends =       'extends';
extern =        'extern';
grantee =       'grantee';
group =         'group';
hops =          'hops';
import =        'import';
in =            'in';
int =           'int';
inst =          'inst';
meta =          'meta';
mstruct =       'mstruct';
oblig =         'oblig';
on =            'on';
out =           'out';
real =          'real';
refrain =       'refrain';
rel =           'rel';
result =        'result';
role =          'role';
spec =          'spec';
string =        'string';
type =          'type';
user =          'user';
valid =         'valid';
when =          'when';

// the following keywords, if in normal state they take you to pre_ocl state
{normal->pre_ocl} raises =        'raises';
{normal->pre_ocl} constraint =    'constraint';

// the following are recognised only when in normal or pre_ocl state. They are not
// recognised when in ocl state. That's because in ocl they can be used as identifiers
{normal, pre_ocl->ocl} subject =  'subject';
{normal, pre_ocl->ocl} target =   'target';
{normal, pre_ocl->ocl} event =    'event';
{normal, pre_ocl->ocl} action =   'action';

// the following keywords are from the OCL Grammar. Not recognised in normal state
{normal->ocl, pre_ocl, ocl}
if =        'if';
then =      'then';
else =      'else';
endif =     'endif';
and =       'and';
or =        'or';
xor =       'xor';
implies =   'implies';
not =       'not';

// the following keywords are only recognised when in OCL state
// apart from 'set' which is also a type in Ponder
      set =        'set';
{ocl} bag =        'bag';
{ocl} sequence =   'sequence';
{ocl} collection = 'collection';

/********************** Characters and special symbols ******************/
{normal, pre_ocl->ocl, ocl}
equals =       '=';
at_sign =      '@';
blank =        (eol | tab | ' ')+;
l_par =        '(';
r_par =        ')';
{normal, pre_ocl->ocl, ocl->normal}       // if in pre_ocl it takes you to ocl_state
l_bra =        '{';                       // and if in ocl it takes you to normal
{normal, pre_ocl, ocl->normal}            // if in ocl it takes you to normal state
r_bra =        '}';
l_brk =        '[';
r_brk =        ']';
dot =          '.';
comma =        ',';
```

```
   {normal, pre_ocl, ocl->normal}            // if in ocl it takes you to normal state
   semicolon =    ';';
   excl_mark =    '!';
   arrow =        '->';
   bar =          '|';
   bar_bar =      '||';
   amper_amper =  '&&';
   caret =        '^';

   //  the following symbols are from OCL Grammar
   not_eq =       '<>';
   lt =           '<';
   lteq =         '<=';
   gt =           '>';
   gteq =         '>=';
   plus =         '+';
   minus =        '-';
   star =         '*';
   slash =        '/';
   colon =        ':';
   dot_dot =      '..';

   /***************************** Other useful tokens ***********************/
   comment =      l_comment | c_comment;
   spec_seq =     spec_chars;
   ident =        letter (letter | digit | '_')*;
   abs_path =     '/.' | ('/' ( path_seq  | (path_seq ('/' path_seq)+) ) ('/' | '/-')? );
   rel_path =     (('../'|'./') path_seq?) | ('../'|'./')? ((path_seq '/' '-'?) | (path_seq
('/' path_seq)+ ('/' | '/-')?));
   int_value =    integer_seq;
   real_value =   float_seq;
   string_value = string_seq;
   boolean_value =boolean_seq;

Ignored Tokens
   blank,
   comment;

/*****************************************************************************
   PRODUCTIONS
*****************************************************************************/
Productions

   // Two kinds of entities can be defined: types and instances.
   // They can be preceded by a general command
   ponder_specification =
        specification*;

   specification =
        {import_or_domain}     import_or_domain |
        {type_or_instance}     type_or_instance ;

   // These are the commands that can be specified outside structures :
   //   - An import statement
   //   - A domain statement that is used to declare the working domain
   import_or_domain =
        {import}       import_statement       |
        {domain}       domain_statement       ;

   // A structure can be specified as a type or as an instance.
   // This forces to specify 'type' and 'inst' keyword within a structure e.g. group
   // even if there is an outer definition.
   type_or_instance =
        {type_section}         type type_definition+  |
        {instance_section}     inst inst_declaration+ ;

   /*************************************************************************
        Types
   *************************************************************************/
   // These are all the possible structure types that can be specified
   //   - A Policy type
   //   - A Grouping structure type
   //   - A Meta-Policy (meta_pol) type
   type_definition =
        {policy}       policy_type    |
        {group}        group_type     |
```

```
     {role}         role_type      |
     {rel}          rel_type       |
     {mstruct}      mstruct_type   |
     {meta}         meta_type      ;

// All the (basic/simple) policy types that can be specified
//   - authorisation
//   - obligation
//   - refrain
//   - delegation
policy_type =
     {auth_plus}    pos_auth_type  |
     {oblig}        oblig_type     |
     {neg}          neg_pol_type   |       // auth- and refrain
     {deleg}        deleg_type     ;

/*************************************************************************
     Instances
*************************************************************************/
inst_declaration =
     {policy}       policy_inst    |
     {group}        group_inst     |
     {role}         role_inst      |
     {rel}          rel_inst       |
     {mstruct}      mstruct_inst   |
     {meta_pol}     meta_inst      ;

policy_inst =
     {auth_plus}    pos_auth_inst  |
     {oblig}        oblig_inst     |
     {neg}          neg_pol_inst   |       // auth- and refrain
     {deleg}        deleg_inst     ;

// The instantiation Command
instantiation =
     ident_or_path equals actual_call_decl semicolon;

/*************************************************************************
     Basic Policies
*************************************************************************/
/******************** positive authorisation policy **********************/
pos_auth_type =
     auth_plus formal_call_decl l_bra pos_auth_type_body* r_bra;

// The definition of an instance of a positive authorisation can be directly
// specified or through an instantiation command.
// The same is true with all the other types of structures
pos_auth_inst =
     {def}           auth_plus ident_or_path l_bra pos_auth_type_body* r_bra    |
     {instantiation} auth_plus instantiation+                                   ;

pos_auth_type_body =
     {common}       policy_elements                     |
     {action}       action pos_auth_actions semicolon   ;

// Positive auth actions can have filters associated with them
pos_auth_actions =
           pos_auth_action_decl pos_auth_action_tail*   |
     {all} star                                         ;       // all actions

// Authorisation actions are separated by commas.
pos_auth_action_tail =
     comma pos_auth_action_decl;

// Positive auth actions can have filters associated with them
pos_auth_action_decl =
     auth_action filter*;

// For auth, parameters can be omitted from the action eventhough the action
// may have parameters. This indicates that we don't care about the parameters
// The action can be prefixed with the name of the object on which the action
// is called. This must be the target.
auth_action =
     auth_action_prefix? ident auth_parameters_decl?;

auth_action_prefix =
     ident_or_path dot;
```

```
// authorisation parameters can only be identifiers (labels/placeholders). If
// a restriction is to be placed on a parameter of the action, then the identifier
// can be used in the constraint of the policy and the condition shall be
// placed there.
auth_parameters_decl =
     l_par ident_list? r_par;

ident_list =
     ident ident_list_tail*;

ident_list_tail =
     comma ident;

// The specification of the filter for a positive authorisation policy
filter =
     filter_condition? l_bra filter_body+ r_bra;

filter_condition =
     if expression;

filter_body =
     {in}           in             ident   equals expression semicolon |
     {result}       result                 equals expression semicolon ;

/*************** negative authorisation & refrain policy ******************/
neg_pol_type =
     neg_sign formal_call_decl l_bra neg_type_body* r_bra;

neg_pol_inst =
     {def}           neg_sign ident_or_path l_bra neg_type_body* r_bra   |
     {instantiation} neg_sign instantiation+                             ;

neg_type_body =
     {common}       policy_elements                      |
     {action}       action neg_pol_actions semicolon     ;

neg_pol_actions =
               auth_action neg_pol_action_tail*     |
     {all}     star                                 ;       // all actions

neg_pol_action_tail =
     comma auth_action;

neg_sign =
     {auth_minus}   auth_minus     |
     {refrain}      refrain        ;

/************************** obligation policy ***************************/
oblig_type =
     oblig formal_call_decl l_bra oblig_type_body* r_bra;

oblig_inst =
     {def}           oblig ident_or_path l_bra oblig_type_body* r_bra    |
     {instantiation} oblig instantiation+                                ;

oblig_type_body =
     {pol_elememts} policy_elements_aux semicolon |
     {common}       common_element_spec           |
                    oblig_type_body_aux semicolon ;

oblig_type_body_aux =
     {event}        event_spec             |
     {action}       do      oblig_actions  |
     {exception}    catch   exception_spec ;

// an obligation action consists of sequences of actions separated with
// concurrency constraints.
oblig_actions =
     basic_oblig_action next_oblig_action? ;

next_oblig_action =
     concurrency_op oblig_actions ;

basic_oblig_action =
     {single}       oblig_action_decl             |
```

```
     {parenth}      l_par oblig_actions r_par     ;

oblig_action_decl =
     oblig_action_name l_par actual_parameters? r_par ;

oblig_action_name =
     object_prefix? ident;

object_prefix =
     {path}     ident_or_path  dot ;

// Concurrency operators for obligation actions :
//   - a1 -> a2     - a1 must follow a2
//   - a1 | a2      - a1 is performed; if it fails a2 is performed
//                          otherwise execution stops
//   - a1 || a2     - a1 and a2 may be performed concurrently.
//                          Execution continues when either has finished
//   - a1 && a2     - a1 and a2 may be performed concurrently.
//                          Execution continues when both have finished
concurrency_op =
     {seq}          arrow          |
     {or}           bar            |
     {parallel}     bar_bar        |
     {and}          amper_amper    ;

// The specification of events in the body of an obligation policy
event_spec =
     on event_expr ;

// An exception is a single action.
// This can be a script action. An exception "parameter" from the runtime
// exception system is passed as an argument to the exception action.
exception_spec =
     ident l_par actual_parameters? r_par;

/************************* delegation policy ****************************/
// The heading of a delegation policy type declaration has the following format:
//   -       deleg+/- <del-name> (<auth+/deleg+-policy>) [([<formal-parameters>])]
// If the policy is deleg+ then constraints on the validity of the actual delegation
// plus the maximum number of delegation hops can be specified.
deleg_type =
     {pos}   deleg_plus  deleg_formal_call_decl l_bra deleg_plus_type_body* r_bra |
     {neg}   deleg_minus deleg_formal_call_decl l_bra deleg_type_body* r_bra ;

deleg_formal_call_decl =
     ident_or_path  [lp1]:l_par type_ident? ident [rp1]:r_par
                    [lp2]:l_par formal_parameters? [rp2]:r_par ;

// Delegation policy instance
deleg_inst =
     {def}           deleg_inst_def       |
     {instantiation} deleg_instantiation  ;

deleg_inst_def =
     {pos}   deleg_plus     [id1]:ident_or_path l_par auth_plus?
                            [id2]:ident_or_path r_par l_bra deleg_plus_type_body* r_bra |
     {neg}   deleg_minus    [id1]:ident_or_path l_par auth_plus?
                            [id2]:ident_or_path r_par l_bra deleg_type_body* r_bra ;

// The delegation policy instantiation statement
// The authorisation policies from which to delegate must be specified
// before the rest of the parameters
deleg_instantiation =
     {pos}   deleg_plus     deleg_actual_call_decl+ |
     {neg}   deleg_minus    deleg_actual_call_decl+ ;

deleg_actual_call_decl =
     [id1]:ident_or_path equals [id2]:ident_or_path deleg_actual_params semicolon;

deleg_actual_params =       [lp1]:l_par ident_or_path [rp1]:r_par
                            [lp2]:l_par actual_parameters? [rp2]:r_par ;

// The delegation policy should still allow the specification of a target
// to override the target of the authorisation policies. So, a separate "grantee"
// entry should be there.
deleg_type_body =
     {common}        policy_elements                                    |
```

```
     {grantee}       grantee set_type?     subj_target    semicolon      |
     {access_rights} action deleg_access_rights           semicolon      ;

// The body of a positive delegation policy also allows the specification of the
// valid-clause to indicate delegation constraints.
deleg_plus_type_body =
           deleg_type_body                                |
     {valid} valid           constraint_spec semicolon    |
     {hops}  hops            deleg_hops                   ;

// The access rights to be delegated.
// These are just like actions of a negative authorisation policy (no filters)
deleg_access_rights =
     neg_pol_actions;

// If the policy is deleg+ then the maximum number of delegation hobs can
// be specified. This must be an integer value
deleg_hops =
     {ident}        ident          |
     {int_value}    int_value      ;

/****************** common basic-policy body contents ********************/
// Common single-policy body contents.
// Individual policies can also contain the definition of common elements like
// constants, constraints.
// Obligation policies can also contain event declarations.
// Each command must be separated by a semicolon.
policy_elements =
                    policy_elements_aux semicolon |
     {common_elem}  basic_common_element_spec     ;

policy_elements_aux =
     {subject}      subject set_type?      subj_target           |
     {target}       target  set_type?      subj_target           |
     {constraint}   when                   constraint_spec       ;

// A subject is a domain-scope-expression, and can be assigned to a name
subj_target =
     {dse} subj_target_name? domain_scope_expr ;

subj_target_name =
     ident equals;

/*************************************************************************
     Composite Policies
*************************************************************************/
/****************************** group ************************************/
group_type =
     group comp_type_formal_call_decl l_bra group_body* r_bra ;

group_inst =
     {def}           group ident_or_path l_bra group_body* r_bra |
     {instantiation} group instantiation+                        ;

group_body =
     {common}  comp_pol_body       |
     {nesting} comp_nested_elem    ;

/****************************** role *************************************/
role_type =
     role comp_type_formal_call_decl l_bra role_body* r_bra ;

role_inst =
     {def}           role ident_or_path l_bra role_body* r_bra subject_domain?  |
     {instantiation} role role_instantiation+                                   ;

role_body =
     {common}       comp_pol_body    |
     {nesting}      comp_nested_elem ;

role_instantiation =
     ident_or_path equals actual_call_decl subject_domain? semicolon;

subject_domain =
     at_sign ident_or_path;
```

```
/*************************** relationship ******************************/
// A relationship can contain nested role types and instances in addition to
// those that it may contain as a composite-policy sub-type.
rel_type =
     rel comp_type_formal_call_decl l_bra rel_body* r_bra ;

rel_inst =
     {def}           rel ident_or_path l_bra rel_body* r_bra     |
     {instantiation} rel instantiation+                          ;

rel_body =
     {common}  comp_pol_body       |
     {nesting} rel_nested_elem     ;

rel_nested_elem =
     {type} type rel_type_nested_elem+ |
     {inst} inst rel_inst_nested_elem+ ;

rel_type_nested_elem =
     {common} comp_type_nested_elem |
     {role}   role_type             ;

rel_inst_nested_elem =
     {common} comp_inst_nested_elem |
     {role}   role_inst             ;

/*********************** management structure *****************************/
// A management structure can contain nested role and relationship types and
// instances in addition to those that it may contain as a composite-policy
// sub-type.
mstruct_type =
     mstruct comp_type_formal_call_decl l_bra mstruct_body* r_bra;

mstruct_inst =
     {def}           mstruct ident_or_path l_bra mstruct_body* r_bra     |
     {instantiation} mstruct instantiation+                              ;

mstruct_body =
     {common}  comp_pol_body       |
     {nesting} mstruct_nested_elem ;

mstruct_nested_elem =
     {type} type mstruct_type_nested_elem+ |
     {inst} inst mstruct_inst_nested_elem+ ;

mstruct_type_nested_elem =
     {common}       comp_type_nested_elem  |
     {role}         role_type              |
     {rel}          rel_type               |
     {mstruct}      mstruct_type           ;

mstruct_inst_nested_elem =
     {common}       comp_inst_nested_elem  |
     {role}         role_inst              |
     {rel}          rel_inst               |
     {mstruct}      mstruct_inst           ;

/************ Common body contents for composite policies *****************/
// All composite policty structures can contain nested basic policies, groups
// and meta-policies.
comp_pol_body =
     common_element_spec    ;

comp_nested_elem =
     {type} type comp_type_nested_elem+ |
     {inst} inst comp_inst_nested_elem+ ;

comp_type_nested_elem =
     comp_type_nested_elem_aux ;

comp_type_nested_elem_aux =
     {group}        group_type     |
     {policy}       policy_type    |
     {meta}         meta_type      ;

comp_inst_nested_elem =
     comp_inst_nested_elem_aux ;
```

```
comp_inst_nested_elem_aux =
     {group}        group_inst     |
     {policy}       policy_inst    |
     {meta}         meta_inst      ;

/*************************************************************************
     Meta-Policies
*************************************************************************/
// The meta policy: "meta" ident (<formal-parameters>) "raises" action
// The raises clause is ommitted if the meta policy contains only concurrency
// constraints
meta_type =
                    meta formal_call_decl raises action_call l_bra meta_body r_bra |
     {concurrency}  meta formal_call_decl l_bra meta_body_conc r_bra               ;

meta_body =
     meta_expression next_meta_expression* semicolon ;

meta_expression =
                    expression                          |
     {assign}       l_brk ident r_brk equals expression  ;

next_meta_expression =
     semicolon meta_expression;

meta_inst =
     {def}           meta ident_or_path raises action_call l_bra meta_body r_bra |
     {def_conc}      meta ident_or_path l_bra meta_body_conc r_bra               |
     {instantiation} meta instantiation+                                         ;

meta_body_conc =
       concurrency_expression next_concurrency_expression* semicolon ;

next_concurrency_expression =
     semicolon concurrency_expression ;

// a concurrency expression consists of sequences of activities separated with
// concurrency constraints.
concurrency_expression =
     activity next_activity? ;

next_activity =
     concurrency_op concurrency_expression ;

activity =
     {single}       activity_decl                       |
     {parenth}      l_par concurrency_expression r_par   ;

activity_decl =
     activity_prefix? next_activity_prefix* ident ;

activity_prefix =
     path dot ;

next_activity_prefix =
     ident dot ;

/*************************************************************************
     Parameters
*************************************************************************/
/************************* formal parameters ****************************/
// The formal parameters call declaration used in the definition of types
// The name can be a path instead of an identifier, if there is a need to
// specify the domain where the definition should go. Composite types can also
// include an extends-clause for inheritance.
formal_call_decl =
     ident_or_path  l_par formal_parameters? r_par ;

comp_type_formal_call_decl =
     formal_call_decl extends_type? ;

formal_parameters =
     formal_param formal_param_tail* ;

formal_param_tail =
```

```
     comma formal_param;

// The type of the parameter can be ommited indicating a "don't care" situation
// With this, we can use formal parameters for event-expressions too
formal_param =
     type_decl? ident;

// All the possible types that can be specified/declared.
// For user-defined types, the name of the type can be specified optionally
type_decl =
     {int}          int                              |
     {real}         real                             |
     {string}       string                           |
     {boolean}      boolean                          |
     {domain}       domain                           |
     {set}          set     set_type?                |
     {subject}      subject set_type?                |
     {target}       target  set_type?                |
     {grantee}      grantee set_type?                |
     {event}        event                            |
     {action}       action                           |
     {constraint}   constraint                       |
     {auth_plus}    auth_plus                        |
     {auth_minus}   auth_minus                       |
     {oblig}        oblig                            |
     {refrain}      refrain                          |
     {deleg_plus}   deleg_plus                       |
     {deleg_minus}  deleg_minus                      |
     {role}         role                             |
     {rel}          rel                              |
     {group}        group                            |
     {mstruct}      mstruct                          |
     {meta}         meta                             |
     {type_ident}   type_qualifier? type_ident    ; // any type

type_ident =
     ident_or_path;

type_qualifier =
     {user_defined} user    |       // a user-defined (policy) type
     {extern}       extern  ;       // an external (IDL) type

set_type =
     lt ident gt;

// The type to be extended can also be specified as a path
// The syntax of the extends clause is the same as that of the actual_call_decl
extends_type =
     extends extends_clause next_extends_clause* ;

extends_clause =
     {with_params}  actual_call_decl |
     {path}         ident_or_path    ;

next_extends_clause =
     comma extends_clause ;

/************************* actual parameters ****************************/
// The actual parameters call declaration used in the definition of instances
// The name can be a path instead of an identifier, if there is a need to
// specify the domain where the definition should go.
actual_call_decl =
     ident_or_path  l_par actual_parameters? r_par       ;

actual_parameters =
     actual_param actual_param_tail*;

actual_param  =
     {expr}  expression                     |
     {dse}   l_brk domain_scope_expr r_brk ;

actual_param_tail =
     comma actual_param;

/*************************************************************************
     Common Elements Specification
*************************************************************************/
```

```
// Common Elements can be defined within any of the defined structures (types
// or instances). Common elements that can be specified are:
//          - Events
//          - Constraints
//          - Constants (int, real, string, boolean etc)
//          - Domains
// Basic Common Elements are those that can be specified in Authorisation,
// Delegation and Refrain policies - Events make no sense in those policies
basic_common_element_spec =
     {constraint}          constraint     constraint_def+ |
     {spec}                spec           external_spec+  |
     {constant}            constant_def                   |
     {import_or_domain}    import_or_domain               ;

common_element_spec =
     {basic} basic_common_element_spec     |
     {event} event event_def+              ;

/************************** event definition ****************************/
// The definition of an event.
// e.g. circuitFailure(h,x,y) = envAlarm(h) -> rFailure(x,y)
event_def =
     ident event_params? equals event_expr semicolon ;

// An Event expression is:
//   - basic event
//   - e1 && e2     - occurs when both e1 and e2 occur irrespective of their order
//   - e1 | e2      - occurs when e1 or e2 occurs
//   - e1 -> e2     - occurs when e1 occurs before e2
//   - e1 + time    - occurs a specified period of time after e1 occurs
//   - {e1 ; e2}!e3 - occurs when e1 occurs followed by e2 with no interleaving of e3
//  - n * e         - occurs when e occurs n times
event_expr =
     {basic}        basic_event                                          |
     {next}         basic_event next_event                               |
     {multiple}     int_value star event_expr                            |
     {no_inter}     l_bra [e1]:event_expr semicolon [e2]:event_expr r_bra
                    excl_mark [e3]:event_expr                           ;

next_event =
     {op}    event_op event_expr    |
     {time}  plus int_value         ;

event_op =
     {amper} amper_amper    |
     {bar}   bar            |
     {arrow} arrow          ;

// A basic event is:
//   - the name of the event with any optional parameters: eventName ['(' ... ')']
//   - a method call on the timer object to specify a time event
basic_event =
     {parenth}      l_par   event_expr r_par       |
     {ident}        ident   auth_parameters_decl?  | // same as authorisation-parameters
     {timer}        ident   dot action_call        ;

event_params =
     l_par formal_parameters? r_par;

/************************ constraint definition ***************************/
constraint_def =
     ident constraint_params? equals constraint_spec semicolon ;

constraint_params =
     l_par formal_parameters? r_par;

constraint_spec = expression;


/************************ constants definition ****************************/
constant_def =
     constant_def_aux ;

constant_def_aux =
     {int}          int                   basic_const_assign+    |
     {real}         real                  basic_const_assign+    |
```

```
     {string}       string                basic_const_assign+    |
     {boolean}      boolean               basic_const_assign+    |
     {set}          set     set_type?     set_const_assign+      |
     {extern_type}  extern type_ident     type_const_assign+     | // an external type
     {user_type}    user   type_ident     type_const_assign+     ; // a user-defined type

basic_const_assign =
     ident equals expression semicolon;

set_const_assign =
     ident equals domain_scope_expr semicolon;

type_const_assign =
     ident equals expression semicolon;

/*********************** external specification *************************/
// A specification that is external to Ponder.
external_spec =
     ident spec_seq semicolon ;

/*****************************************************************************
     Import and Domain Statements
*****************************************************************************/
// An Import statement can end with a "/-" meaning, import all, from the
// specified domain path
import_statement =
     import import_path+ ;

import_path =
     {path} path    semicolon      |
     {ident} ident   semicolon      ;

domain_statement =
     domain domain_statement_body+ ;

domain_statement_body =
     {var_decl}     ident equals path      semicolon |
     {decl}         path                  semicolon |
     {lib_action}   ident dot action_call  semicolon ;

/*****************************************************************************
     Expressions
*****************************************************************************/
/********************** domain scope expressions *************************/
domain_scope_expr =
     {basic}        basic_domain_scope_expr                        |
     {composite}    basic_domain_scope_expr next_domain_scope_expr ;

next_domain_scope_expr =
     domain_op domain_scope_expr   ;

basic_domain_scope_expr =
     {domain_object} domain_object                        |
     {single_obj}    l_bra  domain_object  r_bra          |
     {star}          star   int_value?     domain_object  |
     {at_sign}       at_sign int_value?    domain_object  |
     {par}           l_par domain_scope_expr r_par        ;

domain_op =
     {plus}         plus    |
     {minus}        minus   |
     {intersect}    caret   ;

domain_object =
     ident_or_path next_domain_object*;

next_domain_object =
     {obj_attr}     dot     object_attr    |
     {action_call}  dot     action_call    |       // for domain library objects
     {arrow}        arrow   feature_call   ;

object_attr =
     {subject}      subject |
     {target}       target  ;

ident_or_path =
     {ident} ident   |
```

```
     {path}  path    ;

path =
     {abs} abs_path |
     {rel} rel_path ;

/****************************** expressions ******************************/
// Expressions in Ponder Follow the OCL Syntax (version 1.3)
expression =
     logical_expression;

if_expression =
     if [if_branch]:expression then [then_branch]:expression
     else [else_branch]:expression endif ;

logical_expression =
     relational_expression next_logical_expression* ;

next_logical_expression =
     logical_operator relational_expression ;

relational_expression =
     add_expression next_relational_expression? ;

next_relational_expression =
     relational_operator add_expression ;

add_expression =
     mult_expression next_add_expression* ;

next_add_expression =
     add_operator mult_expression ;

mult_expression =
     unary_expression next_mult_expression* ;

next_mult_expression =
     mult_operator unary_expression ;

unary_expression =
     {unary} unary_operator postfix_expression |
     {postfix} postfix_expression ;

postfix_expression =
     primary_expression next_postfix_expression* ;

next_postfix_expression =
     {action_call}  dot     action_call    |
     {arrow}        arrow   feature_call   ;

primary_expression =
     {lit_col}      literal_collection     |
     {literal}      literal                |
     {action}       action_call            |
     {parenth}      l_par expression r_par  |
     {if}           if_expression          ;

feature_call_parameters =
     l_par declarator? expression r_par ;

literal =
     {path}         path           |      // added. Not part of OCL grammar.
     {string}       string_value   |
     {real}         real_value     |
     {int}          int_value      |
     {boolean}      boolean_value  |
     {subject}      subject        |
     {target}       target         ;

simple_type_specifier =
     ident_or_path ;

literal_collection =
     collection_kind l_bra expression_list_or_range? r_bra ;

expression_list_or_range =
```

```
     expression expression_list_or_range_tail? ;

expression_list_or_range_tail =
     {list}  expression_list_tail+  |
     {range} dot_dot expression     ;

expression_list_tail =
     comma expression ;

feature_call =
     ident qualifiers? feature_call_parameters?   ;

action_call =
     ident   action_call_params? ;

action_call_params =
     l_par actual_parameters? r_par ;

qualifiers =
     l_brk actual_parameter_list r_brk ;

// "iterate" alternative added (missing in ocl specification)
declarator =
     {standard} ident declarator_tail* declarator_type_declaration? bar |
     {iterate} [iterator]:ident [iter_type]:declarator_type_declaration
                    semicolon [accumulator]:ident
                    [acc_type]:declarator_type_declaration equals expression bar;

declarator_tail =
     comma ident ;

declarator_type_declaration =
     colon simple_type_specifier ;

actual_parameter_list =
     expression actual_parameter_list_tail* ;

actual_parameter_list_tail =
     comma expression ;

logical_operator =
     {and}          and     |
     {or}           or      |
     {xor}          xor     |
     {implies}      implies ;

collection_kind =
     {set}          set            |
     {bag}          bag            |
     {sequence}     sequence       |
     {collection}   collection     ;

relational_operator =
     {equals}       equals  |
     {not_eq}       not_eq  |
     {gt}           gt      |
     {lt}           lt      |
     {gteq}         gteq    |
     {lteq}         lteq    ;

add_operator =
     {plus}  plus    |
     {minus} minus   ;

mult_operator =
     {mult}  star    |
     {div}   slash   ;

unary_operator =
     {minus} minus   |
     {not}   not     ;
```

B.2 Predefined Libraries

Timer

The timer library object is used to specify events. It contains functions that can be used to specify time-point events, repeated events based on the duration, and repeated events at specific time-points. For both, Timer and Time objects, the following are true:

- *Date* is a string of the form: "dd:mm:yyyy". Any of the sub-strings of date can be specified as '*' which is used as a wildcard character. So, "01:*:2000" means the 1st of each month in the year 2000.
- *Time* is a string of the form: "hh:mm:ss".
- *Period* is a string from one of the following: "msec", "sec", "min", "hour", "day", "week", "year".
- *DayOfWeek* is a string from one of the following: "mon", "tue", "wed", "thu", "fri", "sat", "sun".
- *Month* is a string from one of the following: "jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec";

Function Name	Parameters	Operation
At	(Date), Time	Specifies an event that occurs at a specific date and time. The date cannot include wildcard characters, but can be ommitted to mean any date. The function thus has an overloaded version with only a Time parameter. E.g. Timer.at("08:00:00") specifies an event that occurs at 8:00am.
Every	Number (duration), Period	Specifies an event that occurs repeatedly every specified period of time. E.g. Timer.every(5, "min") specifies an even that repeats every 5 minutes.
EveryDate	Date	Specifies an event that occurs repeatedly every specific date. E.g. Timer.everyDate("01:*:*") specifies an event that occurs every 1st of each month.
EveryDay	DayOfWeek, (Date)	Specifies an event that occurs repeatedly on the specific day of the week. The date parameter can be left blank to indicate any date. E.g. Timer.everyDay("mon", "*:01:*") specifies an event that occurs on every Monday during January.
EveryAt	Number (duration), Period, Time	Specifies an event that occurs repeatedly every specified period of time at a specific time. E.g. Timer.everyAt(2, "day", "18:00:00") specifies an event that occurs other day at 6:00pm.
EveryDateAt	Date, Time	Specifies an event that occurs repeatedly every specified date at a specific time. E.g. Timer.everyDateAt("01:12:*", "12:00:00") specifies an event that occurs every first of December at noon.
EveryDayAt	DayOfWeek, Date, Time	Specifies an event that occurs repeatedly every specified day of the week at a specific time. E.g. Timer.everyDayAt("wed", "*:*:*", "12:00:00") specifies an event that occurs every Wednesday at noon.

Time

The time library object is used to provide utility functions for time-based constraints.

Function Name	Parameters	Operation
between	(Date), Time – to specify first time-point (Date), Time – to specify second time-point	Specifies a time range. This function is overloaded. It also accepts only 2 parameters of type Time (instead of four) – the dates can be ignored. E.g. Time.between("01:01:*", "12:00:00", "01:05:*, "12:00:00") specifes a range between 1st of January at 12:00am and 1st of May at 12:00am
after	(Date), Time	Specifies a time range after a specified time-point. The function is overloaded to accept only 1 parameter, the Time – the date can be ommitted. E.g. Time.after("18:00:00"), means after 6:00pm
before	(Date), Time	Specifies a time range before a specified time-point. The function is overloaded to accept only 1 parameter, the Time – the date can be ommitted. E.g. Time.before("01:10:2000", "02:30:00"), means before the 1st of October 2000 at 2:30am.
date	None	Returns a string for the current date
month	None	Returns a string for the current month
dayOfWeek	None	Returns a string for the current day
time	None	Returns a string for the current time
duration	Number, Period	Specifies a duration. E.g. Time.duration(5, "hour") to indicate a duration of 5 hours.

Domain

The following are functions that are defined on any domain object.

Function Name	Parameters	Operation
get	String: The name of an object in the domain	Returns the object within the current domain whose name is given. E.g. Printers.get("printer1"), returns the object "printer1" from a domain called Printers.
getDomain	String: The relative path to a sub-domain of the current domain	Returns a sub-domain of the current domain, whose relative path is given. E.g. Printers.get("floor4/color"), returns the sub-domain: <Printers>/floor4/color.

Appendix C

Formal Semantics

C.1 Abstract Syntax

In the following, keywords appear in bold as **keyword**, and non-terminals start with a capital letter.

```
Program ::=          Statement Program | ε
Statement ::=        ConstDef | TypeDef | Inst | RuntimeCommand
TypeDef ::=          BasicPolTypeDef | GroupTypeDef | RoleTypeDef
BasicPolTypeDef ::=  AuthTypeDef | DelegTypeDef | ObligTypeDef | RefrainTypeDef

AuthTypeDef ::=      type auth+ path/id ( (VarType id)* ) { Bauth+ } |
                     type auth- path/id ( (VarType id)* ) { Bauth- }
Bauth+ ::=           Bauth
                     action PAuthActionList
Bauth- ::=           Bauth
                     action NAuthActionList
Bauth ::=            subject Dse
                     target Dse
                     when Expr

DelegTypeDef ::=     type deleg+ path/id (path id) ( (VarType id)* ) { Bdeleg+ } |
                     type deleg- path/id  (path id) ( (VarType id)* ) { Bdeleg }
Bdeleg+ ::=          Bdeleg
                     valid Expr
                     hops intValue
Bdeleg ::=           subject Dse
                     target Dse
                     grantee Dse
                     action NAuthActionList
                     when Expr

ObligTypeDef ::=     type oblig path/id ( (VarType id)* ) { Boblig }
Boblig ::=           on EventExpr
                     subject Dse
                     target Dse
                     do ObligAction
                     when Expr

RefrainTypeDef ::=   type refrain path/id ( (VarType id)* ) { Brefrain }
Brefrain ::=         subject Dse
                     target Dse
                     action NAuthActionList
                     when Expr

GroupTypeDef ::=     type group path/id ( (VarType id)* ) { Bgroup }
                     [extends path2/id2 ((Expr | Dse)* )]
Bgroup ::=           TypeDef | Inst | Bgroup | ε

RoleTypeDef ::=      type role path/id ( (VarType id)* ) { Brole }
                     [extends path2/id2 ( (Expr | Dse)* )]
Brole ::=            BasicPolTypeDef | GroupTypeDef | BasicPolInst | GroupInst | Brole | ε

ConstDef ::=         PrimType id = Expr |
                     set [id] id = Dse |
                     domain id = path |
                     TypeName id = path
```

```
PosAuthActionList ::= (id FilterExpr (, id FilterExpr )* ) | *
FilterExpr ::=        if Expr { (id = Expr)* result = Expr }
NegAuthActionList ::= (id (, id )* ) | *

EventExpr ::=         BasicEvent                    |
                      ( EventExpr )                 |
                      EventExpr && EventExpr        |
                      EventExpr | EventExpr         |
                      EventExpr → EventExpr         |
                      EventExpr + int               |
                      int * EventExpr               |
                      { EventExpr ; EventExpr } ! EventExpr
BasicEvent ::=        id [((id)* )]  |  Timer . methodName ((Expr|Dse)*))

ObligAction ::=       BasicAction                   |
                      ( ObligAction )               |
                      ObligAction  → ObligAction    |
                      ObligAction || ObligAction    |
                      ObligAction && ObligAction    |
                      ObligAction | ObligAction
BasicAction ::=       Prefix . actionName ( [(Expr | Dse)*] )
Prefix ::=            subject | target

Inst ::=              BasicPolInst | GroupInst | RoleInst
BasicPolInst ::=      inst auth+ path1/id1 = path2/id2 ( (Expr | Dse)* ) |
                      inst auth- path1/id1 = path2/id2 ( (Expr | Dse)* ) |
                      inst deleg+ path1/id1 = path2/id2 ( path/id3 )  ( (Expr | Dse)* ) |
                      inst deleg- path1/id1 = path2/id2 ( path/id3 )  ( (Expr | Dse)* ) |
                      inst oblig path1/id1 = path2/id2 ( (Expr | Dse)* ) |
                      inst refrain path1/id1 = path2/id2 ( (Expr | Dse)* )

GroupInst ::=         inst group path1/id1 = path2/id2 ( (Expr | Dse)* )
RoleInst ::=          inst role path1/id1 = path2/id2 ( (Expr | Dse)* ) @ path

PrimType ::=          int | double | char | string | boolean
TypeName ::=          id | path
VarType ::=           PrimType | set [id] | domain | TypeName
Expr ::=              Value | Var | Expr.methodName((Expr|Dse)*) | Expr.fieldName | OCLexpr
Value ::=             PrimValue | SetValue | RefValue
PrimValue ::=         intValue | realValue | stringValue | charValue | booleanValue | path
SetValue ::=          {RefValue*}
RefValue ::=          r_i
Dse  ::=              Dse SetOp Dse | path | * n path | @ n path |
                      path.methodName((Expr|Dse)*) | path->featureCall
SetOp ::=             + | - | ^

// All expressions that can be assigned to policy elements
EExpr ::=             Expr | Dse | PAuthActionList | NAuthActionList

// The commands that can be used at runtime to manage the system. Added to the grammar
RuntimeCommand ::=    enable(path) | disable(path) | exec(id, id, id, Expr*) |
                      exec(id, {id}, id, Expr*) | exec({id}, {id}, id, Expr*) |
                      execFilter(id, id, id, Expr*, Expr) | field(id, id) |
                      applyFilter(id, id, id, Expr*, Expr) | allows(id, id, id, id) |
                      disallows(id, id, id, id) | delegate(id, id, id*)
```

C.2 Structural Operational Semantics

Program (statement) execution is sequential

$$\frac{\langle P,\ \sigma,\ \Delta\rangle \longrightarrow \langle P_1,\ \sigma_1,\ \Delta_1\rangle}{\langle P\ S,\ \sigma,\ \Delta\rangle \longrightarrow \langle P_1\ S,\ \sigma_1,\ \Delta_1\rangle}\ \text{(program1)}$$

$$\frac{\langle P,\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma_1,\ \Delta_1\rangle}{\langle P\ S,\ \sigma,\ \Delta\rangle \longrightarrow \langle S,\ \sigma_1,\ \Delta_1\rangle}\ \text{(program2)}$$

Type and instantiation described for auth+. Same rules apply to auth-, oblig and refrain.

$$\frac{\Delta_1 = \Delta[\text{path/t} \mapsto \text{type auth+ path/t } (T_1\ x_1,\ \ldots,\ T_n\ x_n)\ \{\ \text{Bauth+}\ \}]}{\langle \text{type auth+ path/t } (T_1\ x_1,\ \ldots,\ T_n\ x_n)\ \{\ \text{Bauth+}\ \},\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma,\ \Delta_1\rangle}\ \text{(type auth+)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ \Delta(\text{path/t}) = \text{type auth+ path/t } (T_1\ x_1,\ \ldots,\ T_n\ x_n)\ \{\ \text{Bauth+}\ \} \\ \langle e_i,\ \sigma,\ \Delta\rangle \longrightarrow^{*} \langle val_i,\ \sigma_1,\ \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ \text{Bauth+}_1 = \text{Bauth+}[z_1/x_1,\ \ldots,\ z_n/x_n][\text{state} \mapsto \text{disabled}] \\ \sigma_3 = \sigma_2[r_1 \mapsto \ll \text{Bauth+}_1 \gg^{\text{path/t}}] \\ \Delta_1 = \Delta[\text{path}_1/a \mapsto \ll \text{Bauth+}_1 \gg^{\text{path/t}}] \end{array}}{\langle \text{inst auth+ path}_1/a = \text{path/t}(e_1,\ \ldots,\ e_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle r_1,\ \sigma_3,\ \Delta_1\rangle}\ \text{(inst auth+)}$$

Rules showing when an action execution is granted

$$\frac{\begin{array}{l} \exists\ pol_1 \in \text{Policies}(\Delta,\ \text{auth+})\ (\langle \text{allows}(pol_1,s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{true}\ ,\ \sigma,\ \Delta\rangle) \\ \forall\ pol_2 \in \text{Policies}(\Delta,\ \text{auth-})\ (\langle \text{disallows}(pol_2,\ s,\ t,\ a),\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{false}\ ,\ \sigma,\ \Delta\rangle) \end{array}}{\langle \text{exec}(s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \text{grant}}\ \text{(exec grant)}$$

$$\frac{\begin{array}{l} \exists\ pol_1 \in \text{Policies}(\Delta,\ \text{auth+})\ (\langle \text{allows}(pol_1,s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \\ \langle \text{applyFilter}(s,t,a,v_1\ldots v_n,\text{filterExpr}),\ \sigma,\ \Delta\rangle) \\ \forall\ pol_2 \in \text{Policies}(\Delta,\ \text{auth-})\ (\langle \text{disallows}(pol_2,\ s,\ t,\ a),\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{false}\ ,\ \sigma,\ \Delta\rangle) \end{array}}{\langle \text{exec}(s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{applyFilter}(s,t,a,v_1\ldots v_n,\text{filterExpr}),\ \sigma,\ \Delta\rangle}\ \text{(exec grant filter)}$$

Rules showing when an action execution is denied

$$\frac{\exists\ pol \in \text{Policies}(\Delta,\ \text{auth-})\ (\langle \text{disallows}(pol,\ s,\ t,\ a),\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{true}\ ,\ \sigma,\ \Delta\rangle)}{\langle \text{exec}(s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \text{deny}}\ \text{(exec deny)}$$

$$\frac{}{\langle \text{exec}(s,t,a,v_1\ldots v_n),\ \sigma,\ \Delta\rangle \longrightarrow \text{deny}}\ \text{(exec default)}$$

Rules showing the application of authorisation filters to the access decision process

$$\frac{\begin{array}{l} \text{filter}_1 == \text{if } \text{expr}_c\ \{p_1 = \text{expr}_1\ \ldots\ p_n = \text{expr}_n;\ \text{result} = \text{expr}_r\} \\ \langle \text{expr}_c,\ \sigma,\ \Delta\rangle \longrightarrow \langle \text{false},\ \sigma_1,\ \Delta\rangle \end{array}}{\begin{array}{l} \langle \text{applyFilter}(s,\ t,\ ba,\ v_1\ldots v_n,\text{filter}_1\ldots\text{filter}_k),\ \sigma,\ \Delta\rangle \longrightarrow \\ \langle \text{applyFilter}(s,\ t,\ ba,\ v_1\ldots v_n,\text{filter}_2\ldots\text{filter}_k),\sigma_5,\ \Delta\rangle \end{array}}\ \text{(auth exec filter next)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ \text{filter}_1 == \text{if } \text{expr}_c\ \{p_1 = \text{expr}_1\ \ldots\ p_n = \text{expr}_n;\ \text{result} = \text{expr}_r\} \\ \langle \text{expr}_c,\ \sigma_1,\ \Delta\rangle \longrightarrow \langle \text{true},\ \sigma_2,\ \Delta\rangle \\ \sigma_3 = \sigma_2[p_1 \mapsto v_1]\ldots[p_n \mapsto v_n] \\ \text{expr'}_i = \text{expr}_i[v_i/p_i] \\ \langle \text{expr'}_I,\ \sigma_3,\ \Delta\rangle \longrightarrow^{*} \langle v'_i,\ \sigma'_3,\ \Delta\rangle \text{ for } i \in \{1\ldots n\} \end{array}}{\begin{array}{l} \langle \text{applyFilter}(s,\ t,\ a,\ v_1\ldots v_n,\text{filter}_1\ldots\text{filter}_k),\ \sigma,\ \Delta\rangle \longrightarrow \\ \langle \text{execFilter}(s,t,a,v'_1\ldots v'_n,\ \text{filterExpr}),\sigma_5,\ \Delta\rangle \end{array}}\ \text{(auth exec filter)}$$

$$\frac{}{\langle \text{execFilter}(s,t,a,v_1\ldots v_n,\text{filterExpr})\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma,\ \Delta\rangle}\ \text{(auth exec filter done)}$$

Rules to describe when an auth+ policy allows an action execution

$$\frac{\begin{array}{l}pol \in Policies(\Delta, auth+)\\ pol(state) = enabled\\ \langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s , \sigma, \Delta\rangle\\ s \in set_s\\ \langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t , \sigma, \Delta\rangle\\ t \in set_t\\ a \in pol(action)\\ r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle true , \sigma_2, \Delta\rangle\\ filter(a, pol) == ""\end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle true, \sigma, \Delta\rangle}\ (allows\ true)$$

$$\frac{\begin{array}{l}pol \in Policies(\Delta, auth+)\\ pol(state) = enabled\\ \langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s , \sigma, \Delta\rangle\\ s \in set_s\\ \langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t , \sigma, \Delta\rangle\\ t \in set_t\\ a \in pol(action)\\ r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle true , \sigma_2, \Delta\rangle\\ filter(a, pol) != ""\end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle applyFilter(s,t,a,v_1 \ldots v_n, filter(a,pol)), \sigma, \Delta\rangle}\ (allows\ true\ filter)$$

$$\frac{pol \notin Policies(\Delta, auth+)}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle}\ (allows\ false\ 1)$$

$$\frac{\begin{array}{l}\langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s , \sigma, \Delta\rangle\\ s \notin set_s\end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle}\ (allows\ false\ 2)$$

$$\frac{\begin{array}{l}\langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t , \sigma, \Delta\rangle\\ t \notin set_t\end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle}\ (allows\ false\ 3)$$

$$\frac{a \notin pol(action)}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle}\ (allows\ false\ 4)$$

$$\frac{\begin{array}{l}r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle false, \sigma_2, \Delta\rangle\end{array}}{\langle allows(pol, s,t,a, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle}\ (allows\ false\ 5)$$

Rules to describe when an auth+ policy allows an action execution

$$\frac{\begin{array}{l}pol \in Policies(\Delta, auth-)\\ pol(state) = enabled\\ \langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s , \sigma, \Delta\rangle\\ s \in set_s\\ \langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t , \sigma, \Delta\rangle\\ t \in set_t\\ a \in pol(action)\\ r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle true , \sigma_2, \Delta\rangle\end{array}}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle true, \sigma, \Delta\rangle}\ (disallows\ true)$$

$$\frac{pol \notin Policies(\Delta, auth\text{-})}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \text{ (disallows false 1)}$$

$$\frac{\langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s , \sigma, \Delta\rangle \quad s \notin set_s}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \text{ (disallows false 2)}$$

$$\frac{\langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set_t , \sigma, \Delta\rangle \quad t \notin set_t}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \text{ (disallows false 3)}$$

$$\frac{a \notin pol(action)}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \text{ (disallows false 4)}$$

$$\frac{\begin{array}{l} r_1, r_2 \text{ are new in } \sigma \\ \sigma_1 = \sigma[r_1 \mapsto t] \\ \sigma_2 = \sigma_1[r_2 \mapsto s] \\ \langle pol(constraint), \sigma_2, \Delta\rangle \longrightarrow \langle false, \sigma_2, \Delta\rangle \end{array}}{\langle disallows(pol, s,t,a), \sigma, \Delta\rangle \longrightarrow \langle false, \sigma, \Delta\rangle} \text{ (disallows false 5)}$$

Constraint, Subject and Target evaluations

$$\frac{pol(constraint) = expr \quad \langle expr, \sigma, \Delta\rangle \longrightarrow^* \langle val, \sigma_1, \Delta\rangle}{\langle pol(constraint), \sigma, \Delta\rangle \longrightarrow \langle val, \sigma_1, \Delta\rangle} \text{ (constraint)}$$

$$\frac{pol(subject) = dse \quad \langle dse, \sigma, \Delta\rangle \longrightarrow^* \langle set, \sigma_1, \Delta\rangle}{\langle pol(subject), \sigma, \Delta\rangle \longrightarrow \langle set, \sigma_1, \Delta\rangle} \text{ (subject)}$$

$$\frac{pol(target) = dse \quad \langle dse, \sigma, \Delta\rangle \longrightarrow^* \langle set, \sigma_1, \Delta\rangle}{\langle pol(target), \sigma, \Delta\rangle \longrightarrow \langle set, \sigma_1, \Delta\rangle} \text{ (target)}$$

Constant definitions

$$\frac{\langle expr, \sigma, \Delta\rangle \longrightarrow^* \langle val, \sigma_1, \Delta\rangle}{\langle PrimType\ id = expr, \sigma, \Delta\rangle \longrightarrow \langle \sigma_1[id \mapsto val], \Delta\rangle} \text{ (primType)}$$

$$\frac{\langle dse, \sigma, \Delta\rangle \longrightarrow^* \langle \{Obj_1 \ldots Obj_n\}, \sigma, \Delta\rangle}{\langle set\ [id_1]\ id_2 = dse, \sigma, \Delta\rangle \longrightarrow \langle \sigma[id \mapsto \{Obj_1 \ldots Obj_n\}], \Delta\rangle} \text{ (setType)}$$

$$\frac{}{\langle domain\ id = path, \sigma, \Delta\rangle \longrightarrow \langle \sigma[id \mapsto path], \Delta\rangle} \text{ (domain)}$$

$$\frac{}{\langle TtypeName\ id = path, \sigma, \Delta\rangle \longrightarrow \langle \sigma[id \mapsto path], \Delta\rangle} \text{ (typeName)}$$

Expressions

$$\frac{}{\langle val, \sigma, \Delta\rangle \longrightarrow \langle val, \sigma, \Delta\rangle} \text{ (value)}$$

$$\frac{\sigma(id) = val}{\langle id, \sigma, \Delta\rangle \longrightarrow \langle val, \sigma, \Delta\rangle} \text{ (var)}$$

$$\frac{evalOCL(\sigma, \Delta, OCLexpr) = val}{\langle OCLexpr, \sigma, \Delta\rangle \longrightarrow \langle val, \sigma, \Delta\rangle} \text{ (OCL expr)}$$

$$\frac{\langle expr,\ \sigma,\ \Delta\rangle \longrightarrow \langle r_i,\ \sigma_1,\ \Delta\rangle}{\langle expr.f,\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma_1(r_i,\ f)),\ \sigma_1,\ \Delta\rangle} \quad \text{(field access 1)}$$

$$\frac{\langle expr,\ \sigma,\ \Delta\rangle \longrightarrow \langle path,\ \sigma_1,\ \Delta\rangle \quad obj = Object(\Delta,\ path)}{\langle expr.f,\ \sigma,\ \Delta\rangle \longrightarrow \langle field(obj,\ f),\ \sigma_1,\ \Delta\rangle} \quad \text{(field access 2)}$$

$$\frac{\langle expr,\ \sigma,\ \Delta\rangle \longrightarrow \langle id,\ \sigma_1,\ \Delta\rangle}{\langle expr.f,\ \sigma,\ \Delta\rangle \longrightarrow \langle field(\sigma_1(id),\ f),\ \sigma_1,\ \Delta\rangle} \quad \text{(field access 3)}$$

$$\frac{}{\langle field(r_t,\ f,),\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma,\ \Delta\rangle} \quad \text{(field access 4)}$$

$$\frac{val_j \text{ is ground for } j \in \{1..k\text{-}1\} \quad \langle expr_k,\ \sigma,\ \Delta\rangle \longrightarrow \langle expr'_k,\ \sigma',\ \Delta\rangle}{\langle expr_1.m(val_2,\ldots,val_{k-1},\ expr_k,\ldots expr_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle expr_1.m(val_2,\ldots,val_{k-1},\ expr'_k,\ldots expr_n),\ \sigma',\ \Delta\rangle} \quad \text{(method call 1)}$$

$$\frac{\textit{this} \text{ identifies the object in the context of which the call is made} \quad \langle expr_1,\ \sigma,\ \Delta\rangle \longrightarrow \langle r_i,\ \sigma_1,\ \Delta\rangle}{\langle expr_1.m(val_2\ldots val_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle exec(\textit{this},\ \sigma_1(r_i),\ m,\ val_1..val_n)),\ \sigma_1,\ \Delta\rangle} \quad \text{(method call 2)}$$

$$\frac{\textit{this} \text{ identifies the object in the context of which the call is made} \quad \langle expr_1,\ \sigma,\ \Delta\rangle \longrightarrow \langle path,\ \sigma_1,\ \Delta\rangle \quad obj = Object(\Delta,\ path)}{\langle expr_1.m(val_2\ldots val_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle exec(\textit{this},\ obj,\ m,\ val_1..val_n)),\ \sigma_1,\ \Delta\rangle} \quad \text{(method call 3)}$$

$$\frac{\textit{this} \text{ identifies the object in the context of which the call is made} \quad \langle expr,\ \sigma,\ \Delta\rangle \longrightarrow \langle id,\ \sigma_1,\ \Delta\rangle}{\langle expr_1.m(val_2\ldots val_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle exec(\textit{this},\ \sigma_1(id),\ m,\ val_1..val_n)),\ \sigma_1,\ \Delta\rangle} \quad \text{(method call 4)}$$

Domain Scope Expressions

$$\frac{eval(path,\ \Delta) = \{Obj_1\ldots Obj_n\}}{\langle path,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_1\ldots Obj_n\},\ \sigma,\ \Delta\rangle} \quad \text{(dse path)}$$

$$\frac{eval^*(path,\ n,\ \Delta) = \{Obj_1\ldots Obj_n\}}{\langle *\ n\ path,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_1\ldots Obj_n\},\ \sigma,\ \Delta\rangle} \quad \text{(dse *path)}$$

$$\frac{eval@(path,\ n,\ \Delta) = \{Obj_1\ldots Obj_n\}}{\langle @\ n\ path,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_1\ldots Obj_n\},\ \sigma,\ \Delta\rangle} \quad \text{(dse @path)}$$

$$\frac{obj = Object(\Delta,\ path)}{\langle path.m(val_2\ldots val_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle exec(\textit{this},\ obj,\ m,\ val_1..val_n)),\ \sigma,\ \Delta\rangle} \quad \text{(dse action call)}$$

$$\frac{evalOCL(\sigma,\ \Delta,\ path\text{->}featureCall) = \{Obj_1\ldots Obj_n\}}{\langle path\text{->}featureCall,\ \sigma,\ \Delta\rangle \longrightarrow \langle\ \{Obj_1\ldots Obj_n\},\ \sigma,\ \Delta\rangle} \quad \text{(dse feature call)}$$

$$\frac{\langle dse_1,\ \sigma,\ \Delta\rangle \longrightarrow \langle dse'_1,\ \sigma,\ \Delta\rangle}{\langle dse_1\ setOp\ dse_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle dse'_1\ setOp\ dse_2,\ \sigma,\ \Delta\rangle} \quad \text{(comp dse 1)}$$

$$\frac{\langle dse_1,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_1\ldots Obj_n\},\ \sigma,\ \Delta\rangle \quad \langle dse_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle dse'_2,\ \sigma,\ \Delta\rangle}{\langle dse_1\ setOp\ dse_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_1\ldots Obj_n\}\ setOp\ dse'_2,\ \sigma,\ \Delta\rangle} \quad \text{(comp dse 2)}$$

$$\frac{\langle dse_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle \{Obj_k\ldots Obj_m\},\ \sigma,\ \Delta\rangle}{\langle \{Obj_1\ldots Obj_n\}\ setOp\ dse_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle applySetOp(setOp,\ \{Obj_1\ldots Obj_n\},\ \{Obj_k\ldots Obj_m\}),\ \sigma,\ \Delta\rangle} \quad \text{(comp dse 3)}$$

Policy Life-Cycle commands (The rules apply to all types of basic policies)

$$\frac{\begin{array}{l} pol \in Policies(\Delta, auth+) \\ pol = \Delta(path) \\ pol_1 = pol[state \mapsto enabled] \\ \Delta_1 = \Delta[path \mapsto pol_1] \end{array}}{\langle enable(path), \sigma, \Delta \rangle \longrightarrow \langle \sigma, \Delta_1 \rangle} \text{ (enable auth+)}$$

$$\frac{\begin{array}{l} pol \in Policies(\Delta, auth+) \\ pol = \Delta(path) \\ pol_1 = pol[state \mapsto disabled] \\ \Delta_1 = \Delta[path \mapsto pol_1] \end{array}}{\langle disable(path), \sigma, \Delta \rangle \longrightarrow \langle \sigma, \Delta_1 \rangle} \text{ (disable auth+)}$$

Delegation; the same rules apply to negative delegation policies.

$$\frac{\Delta_1 = \Delta[path/t \mapsto type\ deleg+\ path/t\ (T_0\ a)\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bdeleg+\ \}]}{\langle type\ deleg+\ path/t\ (T_0\ a)(T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bdeleg+\ \}, \sigma, \Delta\rangle \longrightarrow \langle \sigma, \Delta_1\rangle} \text{ (type deleg+)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ authPol := \Delta(path_2/a) \\ \Delta(path/t) = type\ deleg+\ path/t\ (T_0\ a)\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bdeleg+\ \} \\ \langle e_i, \sigma, \Delta\rangle \longrightarrow \langle val_i, \sigma_1, \Delta\rangle \\ \sigma_2 = \sigma_1[z_0 \mapsto authPol]\ [z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ Bdeleg+_1 = Bdeleg+[z_0/a, z_1/x_1, \ldots, z_n/x_n][state \mapsto disabled] \\ dPol := \ll Bdeleg+_1 \gg^{path/t} \\ \sigma_3 = \sigma_2[r_i \mapsto dPol] \\ \Delta_1 = \Delta[path_1/d \mapsto dPol] \\ adPolTypeDef := \text{“}type\ auth+\ path_1/t_2()\ \{subject\ dPol(subject)\ ;\ target\ AS = \\ AuthService;\ action\ delegate(g, actionList);\ when\ dPol(constraint)\ and \\ dPol(action)\text{->}includes(actionList)\ and\ (AS.cascading(path_1/d, g) < dPol(hops))\ \}\text{”} \\ \langle adPolTypeDef, \sigma_3, \Delta_1\rangle \longrightarrow \langle \sigma_3, \Delta_2\rangle \end{array}}{\begin{array}{l} \langle inst\ deleg+\ path_1/d = path/t\ (path_2/a)\ (e_1, \ldots, e_n), \sigma, \Delta\rangle \longrightarrow \\ \langle inst\ auth+\ path_1/a_1 = path_1/t_2(), \sigma_3, \Delta_2\rangle \end{array}} \text{ (inst deleg+)}$$

$$\frac{\begin{array}{l} dPol \in Policies(\Delta, deleg+) \\ dPol = \Delta(path/d) \\ \langle dPol(subject), \sigma, \Delta\rangle \longrightarrow \langle set_s, \sigma, \Delta\rangle \\ s \in set_s \\ \langle dPol(grantee), \sigma, \Delta\rangle \longrightarrow \langle set_g, \sigma, \Delta\rangle \\ g \in set_g \\ actionList \in dPol(action) \\ aPol = dPol(assocPol)\ \textit{// returns associated policy} \\ \langle type\ auth+\ path/t_2()\ \{subject\ g;\ target\ dPol(target);\ action\ actionList;\ when \\ dPol(valid)\ and\ aPol(constraint), \sigma_1, \Delta\rangle \longrightarrow \langle \sigma_1, \Delta_1\rangle \\ adPolTypeDef := \text{“}type\ auth+\ path_1/t_3()\ \{subject\ g;\ target\ AS = AuthService; \\ action\ delegate(g_1, actionList);\ when\ dPol(constraint)\ and\ dPol(action)\text{-} \\ \text{>}includes(actionList)\ and\ (AS.cascading(path_1/d, g) < dPol(hops))\ \}\text{”} \\ \langle adPolTypeDef, \sigma_1, \Delta_1\rangle \longrightarrow \langle \sigma_1, \Delta_2\rangle \end{array}}{\begin{array}{l} \langle delegate(s, g, actionList), \sigma, \Delta\rangle \longrightarrow \langle inst\ auth+\ path_1/a_2 = path/t_2();inst\ auth+ \\ path_1/a_3 = path/t_3();, \sigma_1, \Delta_1\rangle \end{array}} \text{ (delegate)}$$

Obligation

$$\frac{\begin{array}{l}\text{new event instance } e^i \text{ occurs}\\ \sum H'_i = (\sum H_i)[H_i \mapsto H_i + e^i]\\ pol \in Policies(\Delta, oblig)\\ pol(action) = target.ba\\ pol(state) = enabled\\ H_k = ES(pol)\\ occ(H_k, pol(event)) = true \text{ and } T_k \text{ is the trace of } pol(event) \text{ in the history } H_k\\ \sigma' = \sigma[x_i \mapsto val_i]\ \forall\ x_i \in eAttrs(pol(event)) \text{ and } \forall\ val_i \in eValues(T_k)\\ \langle pol(constraint), \sigma', \Delta\rangle \longrightarrow \langle true, \sigma'_1, \Delta\rangle\\ \langle pol(subject), \sigma'_1, \Delta\rangle \longrightarrow \langle set_s, \sigma'_2, \Delta\rangle\\ \langle pol(target), \sigma'_2, \Delta\rangle \longrightarrow \langle set_t, \sigma'_3, \Delta\rangle\\ \sum H''_i = (\sum H'_I)[H_k \mapsto \varepsilon]\\ \hline <e_i, \sigma'_3, \Delta> \longrightarrow^* <v_i, \sigma'_4, \Delta>\end{array}}{\langle \text{do target.ba}(e_1 \ldots e_n), \sigma, \Delta, \sum H_i \rangle \longrightarrow_0 \langle \text{exec}(set_s, set_t, ba, v_1 \ldots v_n), \sigma'_4, \Delta, \sum H''_i\rangle}\ (\text{event exec target})$$

$$\frac{\begin{array}{l}\text{new event instance } e^i \text{ occurs}\\ \sum H'_i = (\sum H_i)[H_i \mapsto H_i + e^i]\\ pol \in Policies(\Delta, oblig)\\ pol(action) = subject.ba\\ pol(state) = enabled\\ H_k = ES(pol)\\ occ(H_k, pol(event)) = true \text{ and } T_k \text{ is the trace of } pol(event) \text{ in the history } H_k\\ \sigma' = \sigma[x_i \mapsto val_i]\ \forall\ x_i \in eAttrs(pol(event)) \text{ and } \forall\ val_i \in eValues(T_k)\\ \langle pol(constraint), \sigma', \Delta\rangle \longrightarrow \langle true, \sigma'_1, \Delta\rangle\\ \langle pol(subject), \sigma'_1, \Delta\rangle \longrightarrow \langle set_s, \sigma'_2, \Delta\rangle\\ \sum H''_i = (\sum H'_I)[H_k \mapsto \varepsilon]\\ \hline <e_i, \sigma'_2, \Delta> \longrightarrow^* <v_i, \sigma'_3, \Delta>\end{array}}{\langle \text{do subject.ba}(e_1 \ldots e_n), \sigma, \Delta, \sum H_i \rangle \longrightarrow_0 \langle \text{exec}(set_s, set_s, ba, v_1 \ldots v_n), \sigma'_3, \Delta, \sum H''_i\rangle}\ (\text{event exec subject})$$

$$\frac{\begin{array}{l}s_i = head(set_s)\\ set'_s = tail(set_s)\\ \langle \text{exec}(s_i, s_i, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle\end{array}}{\langle \text{exec}(set_s, set_s, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \text{exec}(set'_s, set'_s, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle}\ (\text{exec on subject})$$

$$\frac{\begin{array}{l}s_i = head(set_s)\\ set'_s = tail(set_s)\\ \langle \text{exec}(s_i, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle\end{array}}{\langle \text{exec}(set_s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \text{exec}(set'_s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle}\ (\text{exec 1})$$

$$\frac{\begin{array}{l}s_i = head(set_s) = tail(set_s)\\ \langle \text{exec}(s_i, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle\end{array}}{\langle \text{exec}(set_s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle}\ (\text{exec 2})$$

$$\frac{\begin{array}{l}t_i = head(set_t)\\ set'_t = tail(set_t)\\ \langle \text{exec}(s, t_i, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle\end{array}}{\langle \text{exec}(s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \text{exec}(s, set'_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle}\ (\text{exec 3})$$

$$\frac{t_i = \text{head}(set_t) \quad set'_t = \text{tail}(set_t) \quad \langle exec(s, t_i, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 fail}{\langle exec(s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle exec(s, set'_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle} \text{ (exec 4)}$$

$$\frac{t_i = \text{head}(set_t) = \text{tail}(set_t) \quad \langle exec(s, t_i, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle}{\langle exec(s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle} \text{ (exec 5)}$$

$$\frac{t_i = \text{head}(set_t) = \text{tail}(set_t) \quad \langle exec(s, t_i, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 fail}{\langle exec(s, set_t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \sigma, \Delta, \sum H\rangle} \text{ (exec 6)}$$

$$\frac{\exists\, pol \in Policies(\Delta, refrain)\ (\langle disallows(pol, s, t, ba), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle true, \sigma, \Delta, \sum H\rangle)}{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 fail} \text{ (exec refrain fail)}$$

$$\frac{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow deny}{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 fail} \text{ (exec fail)}$$

$$\frac{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow grant}{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle val, \sigma, \Delta, \sum H\rangle} \text{ (exec success)}$$

$$\frac{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta\rangle \longrightarrow \langle execFilter(s, t, ba, v_1 \ldots v_n, filterExpr), \sigma, \Delta\rangle}{\langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle execFilter(s, t, ba, v_1 \ldots v_n, filterExpr), \sigma, \Delta, \sum H\rangle} \text{ (exec success filter)}$$

$$\frac{\begin{array}{l} filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n; result = expr_r\} \\ \langle expr_c, \sigma, \Delta\rangle \longrightarrow \langle false, \sigma_1, \Delta\rangle \end{array}}{\langle exec(s, t, ba, v_1 \ldots v_n, filter_1 \ldots filter_k), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle exec(s, t, ba, v_1 \ldots v_n, filter_2 \ldots filter_k), \sigma_5, \Delta, \sum H\rangle} \text{ (exec filter next)}$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ filter_1 == \text{if } expr_c \{p_1 = expr_1 \ldots p_n = expr_n; result = expr_r\} \\ \langle expr_c, \sigma_1, \Delta\rangle \longrightarrow \langle true, \sigma_2, \Delta\rangle \\ \langle exec(s, t, ba, v_1 \ldots v_n), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle val, \sigma, \Delta, \sum H\rangle \\ \sigma_3 = \sigma_2[z_1 \mapsto val] \\ \langle expr_r, \sigma_3, \Delta\rangle \longrightarrow^* \langle val', \sigma_4, \Delta\rangle \end{array}}{\langle execFilter(s, t, ba, v_1 \ldots v_n, filter_1 \ldots filter_k), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle val', \sigma_4, \Delta, \sum H\rangle} \text{ (exec filter)}$$

Execution of the $\rightarrow$ concurrency operator

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A'_1,\ \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \rightarrow A_2,\ \sigma,\ \Delta,\ \sum H\ \rangle \longrightarrow_0 \langle \mathtt{do}\ A'_1 \rightarrow A_2,\ \sigma_1,\ \Delta_1,\ \sum H'\rangle} \quad \text{(exec seq 1)}$$

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \rightarrow A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A_2,\ \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec seq 2)}$$

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \mathtt{fail}}{\langle \mathtt{do}\ A_1 \rightarrow A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \mathtt{fail}} \quad \text{(exec seq fail 1)}$$

$$\frac{\langle \mathtt{do}\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \mathtt{fail}}{\langle \mathtt{do}\ A_1 \rightarrow A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \mathtt{fail}} \quad \text{(exec seq fail 2)}$$

Execution of the | concurrency operator

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A'_1,\ \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1\ \mid\ A_2,\ \sigma,\ \Delta,\ \sum H\ \rangle \longrightarrow_0 \langle \mathtt{do}\ A'_1\ \mid\ A_2,\ \sigma_1,\ \Delta_1,\ \sum H'\rangle} \quad \text{(exec or 1)}$$

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1\ \mid\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec or 2)}$$

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \mathtt{fail}}{\langle \mathtt{do}\ A_1\ \mid\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle} \quad \text{(exec or 3)}$$

Execution of the && and || concurrency operators. We use the symbol $\otimes$ to denote both of these operators.

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle A'_1,\ \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \otimes A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A'_1 \otimes A_2,\ \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec conc 1)}$$

$$\frac{\langle \mathtt{do}\ A_1,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \otimes A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A_2,\ \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec conc 2)}$$

$$\frac{\langle \mathtt{do}\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle A'_2,\ \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \otimes A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A_1 \otimes A'_2,\ \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec conc 3)}$$

$$\frac{\langle \mathtt{do}\ A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \sigma_1,\ \Delta,\ \sum H'\rangle}{\langle \mathtt{do}\ A_1 \otimes A_2,\ \sigma,\ \Delta,\ \sum H\rangle \longrightarrow_0 \langle \mathtt{do}\ A_1,\ \sigma_1,\ \Delta,\ \sum H'\rangle} \quad \text{(exec conc 4)}$$

Additional rules for the execution of the && concurrency operator.

$$\frac{\langle \texttt{do } A_1, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \texttt{fail}}{\langle \texttt{do } A_1 \texttt{ \&\& } A_2, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \texttt{fail}} \text{ (exec conc fail 1)}$$

$$\frac{\langle \texttt{do } A_2, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \texttt{fail}}{\langle \texttt{do } A_1 \texttt{ \&\& } A_2, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \texttt{fail}} \text{ (exec conc fail 2)}$$

Additional rules for the execution of the || concurrency operator

$$\frac{\langle \texttt{do } A_1 \texttt{ || } A_2, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{do } A_1, \sigma_1, \Delta, \sum H'\rangle}{\langle \texttt{do } A_1 \texttt{ || } A_2 \oplus A_3, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{do } A_2 \oplus A_3, \sigma_1, \Delta, \sum H'\rangle} \text{ (exec conc 5)}$$

$$\frac{\langle \texttt{do } A_1 \texttt{ || } A_2, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{do } A_2, \sigma_1, \Delta, \sum H'\rangle}{\langle \texttt{do } A_1 \texttt{ || } A_2 \oplus A_3, \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{do } A_1 \oplus A_3, \sigma_1, \Delta, \sum H'\rangle} \text{ (exec conc 6)}$$

The rules which show when a refrain policy disallows an action execution

$$\frac{\begin{array}{l}\texttt{pol} \in \texttt{Policies}(\Delta, \texttt{refrain})\\ \texttt{pol}(\texttt{state}) = \texttt{enabled}\\ \langle \texttt{pol}(\texttt{subject}), \sigma, \Delta\rangle \longrightarrow \langle \texttt{set}_s\ , \sigma, \Delta\rangle\\ s \in \texttt{set}_s\\ \langle \texttt{pol}(\texttt{target}), \sigma, \Delta\rangle \longrightarrow \langle \texttt{set}_t\ , \sigma, \Delta\rangle\\ t \in \texttt{set}_t\\ a \in \texttt{pol}(\texttt{action})\\ r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle \texttt{pol}(\texttt{constraint}), \sigma_2, \Delta\rangle \longrightarrow \langle \texttt{true}\ , \sigma_2, \Delta\rangle\end{array}}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{true}, \sigma, \Delta, \sum H\rangle} \text{ (refrain true)}$$

$$\frac{\texttt{pol} \notin \texttt{Policies}(\Delta, \texttt{refrain})}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{false}, \sigma, \Delta, \sum H\rangle} \text{ (refrain false 1)}$$

$$\frac{\begin{array}{l}\langle \texttt{pol}(\texttt{subject}), \sigma, \Delta\rangle \longrightarrow \langle \texttt{set}_s\ , \sigma, \Delta\rangle\\ s \notin \texttt{set}_s\end{array}}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{false}, \sigma, \Delta, \sum H\rangle} \text{ (refrain false 2)}$$

$$\frac{\begin{array}{l}\langle \texttt{pol}(\texttt{target}), \sigma, \Delta\rangle \longrightarrow \langle \texttt{set}_t\ , \sigma, \Delta\rangle\\ t \notin \texttt{set}_t\end{array}}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{false}, \sigma, \Delta, \sum H\rangle} \text{ (refrain false 3)}$$

$$\frac{a \in \texttt{pol}(\texttt{action})}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{false}, \sigma, \Delta, \sum H\rangle} \text{ (refrain false 4)}$$

$$\frac{\begin{array}{l}r_1, r_2 \text{ are new in } \sigma\\ \sigma_1 = \sigma[r_1 \mapsto t]\\ \sigma_2 = \sigma_1[r_2 \mapsto s]\\ \langle \texttt{pol}(\texttt{constraint}), \sigma_2, \Delta\rangle \longrightarrow \langle \texttt{false}\ , \sigma_2, \Delta\rangle\end{array}}{\langle \texttt{disallows}(\texttt{pol}, s,t,a), \sigma, \Delta, \sum H\rangle \longrightarrow_0 \langle \texttt{false}, \sigma, \Delta, \sum H\rangle} \text{ (refrain false 5)}$$

Group composite policy. The type definition rules are the same for all other types of composite policy structures.

$$\frac{\Delta_1 = \Delta[path/t \mapsto \texttt{type group } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bgroup\ \}]}{\langle \texttt{type group } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bgroup\ \},\ \sigma,\ \Delta\rangle \longrightarrow \langle \sigma,\ \Delta_1\rangle}\ (\text{type group})$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ \Delta(path_1/t_1) = \texttt{type group } path_1/t_1\ (T_1\ x_1,\ldots,T_n\ x_n)\ \{\ Bgroup_1\ \} \texttt{ extends } \ldots \\ \langle e_i,\ \sigma,\ \Delta\rangle \longrightarrow^* \langle val_i,\ \sigma_1,\ \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ Bgroup_2 = Bgroup_1[z_1/x_1,\ \ldots,\ z_n/x_n] \\ Bgroup_3 = override(Bgroup,\ Bgroup_2) \\ \Delta_1 = \Delta[path \mapsto \texttt{type group } path/t\ (T_1\ x_1,\ldots,T_n\ x_n)\ \{\ Bgroup_3\ \}] \end{array}}{\langle \texttt{type group } path/t\ (T_1\ x_1,\ldots,T_n\ x_n) \texttt{ extends } path_1/t_1(e_1,\ldots,e_n)\ \{\ Bgroup\ \} \longrightarrow \langle \sigma_2,\ \Delta_1\rangle}\ (\text{group type extends})$$

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ \Delta(path/t) = \texttt{type group } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Bgroup\ \} \\ \langle e_i,\ \sigma,\ \Delta\rangle \longrightarrow^* \langle val_i,\ \sigma_1,\ \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ Bgroup_1 = Bgroup[z_1/x_1,\ \ldots,\ z_n/x_n] \\ Bgroup_2 = replacePath(Bgroup_1,\ path/g) \\ \Delta_1 = createDomain(\Delta,\ path_1/g) \\ \langle Bgroup_2,\ \sigma_2,\ \Delta_1\rangle \longrightarrow^* \langle \sigma_3,\ \Delta_2\rangle \\ \sigma_4 = \sigma_3[r_1 \mapsto \ll Bgroup_2 \gg^{path/t}] \end{array}}{\langle \texttt{inst group } path_1/g = path/t(e_1,\ \ldots,\ e_n),\ \sigma,\ \Delta\rangle \longrightarrow \langle r_1,\ \sigma_4,\ \Delta_2\rangle}\ (\text{inst group})$$

Role Instantiation

$$\frac{\begin{array}{l} z_i \text{ are new identifiers in } \sigma \\ r_1 \text{ is new in } \sigma \\ \Delta(path/t) = \texttt{type role } path/t\ (T_1\ x_1, \ldots, T_n\ x_n)\ \{\ Brole\ \} \\ \langle e_i,\ \sigma,\ \Delta\rangle \longrightarrow^* \langle val_i,\ \sigma_1,\ \Delta\rangle \\ \sigma_2 = \sigma_1[z_1 \mapsto val_1]\ldots[z_n \mapsto val_n] \\ Brole_1 = Brole[z_1/x_1,\ \ldots,\ z_n/x_n] \\ Brole_2 = replacePath(Brole_1,\ path/g) \\ Brole_3 = replaceSubject(Brole_1,\ path_2) \\ \Delta_1 = createDomain(\Delta,\ path_1/g) \\ \langle Brole_3,\ \sigma_2,\ \Delta_1\rangle \longrightarrow^* \langle \sigma_3,\ \Delta_2\rangle \\ \sigma_4 = \sigma_3[r_1 \mapsto \ll Brole_3 \gg^{path/t}] \end{array}}{\langle \texttt{inst role } path_1/g = path/t(e_1,\ \ldots,\ e_n)\ @path_2,\ \sigma,\ \Delta\rangle \longrightarrow \langle r_1,\ \sigma_4,\ \Delta_2\rangle}\ (\text{inst role})$$

Signatures of Runtime Commands

enable(path) ::= path → void

disable(path) ::= path → void

exec(s, t, a, $(v)^*$) ::= subject → target → action name → $(value)^*$ → ANY

exec(s, set_t, a, $(v)^*$) ::= subject → target set → action name → $(value)^*$ → void

exec(set_s, set_t, a, $(v)^*$) := subject set → target set → action name → $(value)^*$ → void

execFilter(s, t, a, $(v)^*$, filter) := subject → target, action name → $(value)^*$ → FilterExpr → ANY

applyFilter(s, t, a, $(v)^*$, filter) := subject → target → action name → $(value)^*$ → FilterExpr → ANY

field(t, f) := target → fieldName → ANY

allows(pol, s, t, a, $(v)^*$) ::= policy → subject → target → action name → $(value)^*$ → (true ∪ false) ∪ execFilter

disallows(pol, s, t, a) ::= policy → subject → target → action name → (true∪false)

delegate(s, g, actionList) ::= subject → grantee → (action name)* → void

C.3 Alloy Model for the Domain System

```
model DomainSystem {

   // The model consists of:
   // - Managed objects called Objects,
   // - Domain entries and
   // - Names (which are assumed to be drawn from a fixed set of names
   domain {
      Object,
      DomainEntry,
      fixed Name
   }

   // Managed objects are partitioned into Non-domain objects and domain objects
   // The root of a domain structure is a domain object
   // The entries relation returns the set of entries of a domain object
   // The name relation of a domain entry returns the name of that entry
   // The contents relation of a domain entry returns the actual object (the contents of
   // the entry
   // The parent relationship returns the domain objects which are parents for an object
   // Children gives the children (objects) of a domain
   state {
      partition NonDomainObj, DomainObj : static Object
      Root: fixed DomainObj!
      entries: DomainObj -> DomainEntry
      name: DomainEntry -> static Name!
      contents: DomainEntry -> static Object!
      parent (~children) : Object -> DomainObj
   }

   // Define parent: Get the domain entries for which the object is an entry, and from
   // those, get the domain objects which contain those entries
   def parent {
      all o | o.parent = o.~contents.~entries
   }

   // Any two entries of the same domain, must have distinct names
   inv UniqueNames {
      all d | all e1, e2: d.entries | e1.name = e2.name -> e1 = e2
   }

   // The root has no parent! All other objects do (not excluding multiple parents)
   inv Parents {
      no Root.parent
   }

   // Make sure that the graph is acyclic. If you follow the parent relation, no domain
   // is an ancestor of itself
   inv Acyclic {
      no d | d in d.+parent
   }

   // Every object is reachable from the root. This also means that there is only one root
   // (which was already specified in the State paragraph)
   inv Reachable {
      Object in Root.*children
   }

   // make sure that structures nire than one level deep are not ruled out
   cond TwoDeep {
      some Root.children.children
   }

   // Every non-domain object is an entry in a domain object
   assert NonDomainObjHasEntry {
      all nd: NonDomainObj | some d | nd in d.entries.contents
   }

   // Operation to add a set of entries es to a domain d
   op NewDomainEntries (d: DomainObj, es: DomainEntry') {
      no es & DomainEntry
      d.entries' = d.entries + es
```

```
      all x: DomainObj + DomainObj' - d | x.entries' = x.entries
   }

   // Operation to create a new object o with name n, in a domain d
   op Create (d: DomainObj!, o: Object'!, n: Name) {
      n !in d.entries.name
      some e: DomainEntry' | NewDomainEntries (d, e) && e.contents' = o && e.name' = n
   }

   // Operation to delete an entry with name n from domain d
   op DeleteEntry (d: DomainObj!, n: Name) {
      n in d.entries.name
      some e: DirEntry | e.name = n
      d.entries' = d.entries - e
   }

   // Specify the properties of the NewDomainEntries operation
   assert EntriesCreated {
      all d: DomainObj, e: DomainEntry' | NewDomainEntries (d,e) ->
             DomainEntry' = DomainEntry + e
   }

   // Specify the properties of the Create operation
   assert CreateWorks {
      all d, o, n | Create (d,o,n) -> o in d.children'
   }

   // Specify the properties of the DeleteEntry operation
   assert EntryDeleted {
      all d: DomainObj, n: Name, e: DomainEntry | DeleteEntry (d,n) ->
             DomainEntry' = DomainEntry – e &&
             e.name = n
   }

}
```