

A Survey of Policy Specification Approaches

Nicodemos Damianou, Arosha K Bandara, Morris Sloman and Emil C Lupu
{ncd, bandara, mss, ecl1}@doc.ic.ac.uk

Department of Computing
Imperial College of Science Technology and Medicine
London, SW7 2BZ, UK

Abstract

Policies are rules governing the choices in behaviour of a system. They are often used as a means of implementing flexible and adaptive systems for management of internet services, distributed systems, and security systems. There is also a need for a common specification of security policy for large-scale, multi-organisational systems where access control is implemented in a variety of heterogeneous components. In this paper we survey both security and management policy specification approaches. We also cover the issues relating to detecting and resolving conflicts which can arise in the policies and some ideas on how to refine high level goals and service level agreements into implementable policies. The paper briefly outlines some of the research issues that have to be solved for large-scale adoption of policy-based systems

Keywords

General terms: Policy specification

Additional keywords and phrases: Security policy, access control, role based access control, authorization, security management, policy based management,

Contents

1	Introduction	3
2	Security Policy	4
2.1	Logic-Based Languages	5
2.1.1	First Order Logic	5
2.1.2	Stratified Logic	5
2.1.3	Deontic Logic	6
2.2	Role-Based Access Control (RBAC) Specification	7
2.2.1	Role Specification Languages	9
2.2.2	TRBAC	9
2.3	Other Security Specification Approaches	10
2.3.1	SPL	10
2.3.2	XACML	11
2.3.3	LaSCO	12
2.3.4	Trust Specification	12
3	Management Policy	13
3.1	IETF/DMTF Policy Core Information Model	13
3.2	Policy Description Language (PDL)	16
3.3	Configuration Management	17
4	Enterprise and Collaboration Policy	18
4.1	Open Distributed Programming Reference Model (ODP-RM)	18
4.2	Law-Governed Interaction (LGI)	19
4.3	Event-Trigger-Rules	20
5	Ponder	22
5.1	Domains	22
5.2	Ponder primitive policies	22
5.3	Ponder Composite Policies	23
6	Analysis and Refinement	25
6.1	Policy Analysis	25
6.2	Policy Refinement	26
7	Commercial Tools	29
7.1	Nortel Optivity	29
7.2	Orchestream Enterprise	29
7.3	HP Openview PolicyXpert	30
7.4	Cisco CiscoAssure	30
7.5	Allot Communications NetPolicy	30
7.6	Computer Associates Infrastructure Management and eTrust Solutions	30
7.7	Tivoli Management Framework – Access Manager	31
8	Research Issues	31
9	Conclusions	32
	References	33

1 Introduction

Policy-based management has become a promising solution for managing enterprise-wide networks and distributed systems. These are typically large-scale systems which require management solutions that are both self-adapting and that dynamically change the behaviour of the managed system. The main motivation for the recent interest in policy-based services, networks and security systems is to support dynamic adaptability of behaviour by changing policy without recoding or stopping the system [Sloman 2001]. This implies that it should be possible to dynamically update the policy rules interpreted by distributed entities to modify their behaviour.

Policies are rules governing the choices in behaviour of a system [Sloman 1994b]. *Obligation policies* are event-triggered condition-action rules that can be used to define adaptable management actions. These policies thus define the conditions for performing a wide range of management actions such as change Quality of Service, when to perform storage server backups, register new users in a system, or install new software. *Authorisation policies* are used to define what services or resources a subject (management agent, user or role) can access. In addition, security management policies are needed to define the actions to be taken when security violations, such as a series of login failures occur for a particular user, or an attack on the system is detected. Furthermore, the heterogeneity of security mechanisms used to implement access control makes security management an important and difficult task.

Policies are persistent so that a one-off command to perform an action is not a policy. Scripts and mobile agents, based on powerful interpreted languages such as Java, can also be used to support adaptability as well as to introduce new functionality into distributed network components. 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. In today's Internet-based environments security concerns tend to increase when mobile code mechanisms are introduced to enable such adaptation, and so many researchers favour a more constrained form of rule-based policy adaptation.

Large-scale systems may contain millions of users and resources. It is not practical to specify policies relating to individual entities – instead, it must be possible to specify policies relating to groups of entities and also to nested groups such as sections within departments, within sites in different countries in an international organisation. It is also useful to group the policies pertaining to the rights and duties of a role or position within an organisation such as a network operator, nurse in a ward or mobile computing 'visitor' in a hotel.

Policies are derived from business goals, service level agreements or trust relationships within or between enterprises. The refinement of these abstract policies into policies relating to specific services and then into policies implementable by specific devices supporting the service is not easy, and not amenable to automation. Although the technologies for building management systems and implementing security 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.

This paper provides a survey of the work on policy specification for both security authorisation and policy-driven management of security and networked systems. In section 2 we cover the various approaches to security policy specification, including logic-based languages, role-based access control and various access control and trust specification techniques. In section 3 we focus on approaches to management policy specification, including standards work, event-triggered policies and configuration management policies. Section 4 covers Enterprise and Collaboration policy, including the Open Distributed Processing Enterprise viewpoint, law-governed systems and event-triggered rules for electronic commerce. In section 5 we describe our Ponder policy specification language, which combines many of the above concepts and can be used for both security and management policies. In section 6 we present an overview of the issues relating to analysing policies for conflicts and refining high level goals into implementable policies, which is followed by a discussion on research issues for policy-based systems. WWW references for much of the work on policy and many of the papers described here can be found from <http://www-dse.doc.ic.ac.uk/Research/policies>.

We do not discuss routing policies as these have been described in a recent survey [Stone, Lundy et al. 2001].

2 Security Policy

Access control is concerned with permitting only authorised users (subjects) to access services and resources (targets). It limits the activity of legitimate users who have been successfully authenticated. Authorisation or access control policy defines the high-level rules specifying the conditions under which subjects are permitted to access targets [Samarati and Vimercati 2000]. However, in many systems there is no real policy specification, only the implementation in terms of low-level mechanisms such as access control lists. 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. Access control policies have been traditionally divided into discretionary and mandatory policies.

Discretionary access control (DAC) policies restrict access to objects based on the identity of the subjects and/or groups to which they belong, and are discretionary in the sense that a subject can pass its access permissions on to another 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. 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.

Mandatory access control (MAC) policies enforce access control on the basis of fixed regulations mandated by a central authority, as typified by the Bell-LaPadula, lattice-based model [Bell and LaPadula 1973]. 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 conceptual framework of the Bell-LaPadula model forms the basis of other derived models one of which is the Biba model. The Biba model uses similar controls as those used in the Bell-LaPadula model for providing *integrity* of data [Biba 1977].

Non-discretionary access control (NDAC) identifies the situations in which authority is vested in some users, but there are explicit controls on delegation and propagation of authority [Abrams 1993]. Any global and persistent access control policy relying on access control decision information not directly controlled by the security administrator is non-discretionary. Administrative policies [Sandhu and Samarati 1994] determine who is authorised to modify the allowed access rights and exist only within discretionary policies. Whereas, 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.

Over the years other sophisticated security models have been proposed to formalise security policies required for commercial applications. The **Clark-Wilson model** [Clark and Wilson 1987] is a well known one for 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 that do not lead to execution of conflicting tasks. This principle has since been adopted as an important constraint in security systems.

A security policy model that specifies clear and concise access rules for clinical information systems can be found in [Anderson 1996]. This model is based on access control lists and the authors claim it can express Bell-LaPadula and other lattice-based models. Finally the **Chinese-wall policy** [Brewer and Nash 1989] was developed as a formal model of a security policy applicable to financial information systems, to prevent information flows that cause conflict of interest for individual consultants. The basis of the model is that people are only allowed to access information that 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.

2.1 Logic-Based Languages

Logic-based languages have proved attractive for the specification of security policy, as they have a well-understood formalism, which is amenable to analysis. However they can be difficult to use and are not always directly translatable into efficient implementation. This section starts by presenting some of these specification languages classified by the type of logic used in their definition. We go on to discuss languages that were specifically developed to support the Role-Based Access Control (RBAC) model.

2.1.1 First Order Logic

There are several examples that illustrate the application of first order logic to the specification of security policy. These include the logical notation introduced in [Chen and Sandhu 1995]; the Role Definition Language (RDL) presented in [Hayton, Bacon et al. 1998] and RSL99 [Ahn and Sandhu 1999]. Because all these approaches are based on the Role Based Access Control (RBAC) model, we will defer a more detailed discussion to Section 2.2 where they can be considered together with other RBAC specification techniques.

In addition to these RBAC examples, there are some examples of the application of Z to defining security policies for a system. Z is a formal specification language that combines features of first order predicate logic with set theory [Spivey 1989]. In [Boswell 1995], the use of Z to specify and validate the security model for the NATO Air Command and Control System is described. The aim of this work was to develop a model for both mandatory and discretionary access controls based on the Bell-LaPadula approach mentioned previously.

One of the main problems encountered when using first order logic for policy specification arises when negation is used together with recursive rules. This leads to logic programs that cannot be decided and cannot be evaluated in a flounder-free manner [Dantsin, Eiter et al. 1997]. Although it is possible to avoid the use of negation or recursion, this is not practical since it diminishes significantly the expressive power of the logical language. In the next section, we discuss an alternative solution, stratified logic, which helps overcome the problems associated with using recursion and negation together.

2.1.2 Stratified Logic

When using first order logic based languages to describe policies, many approaches considered here are described in terms of stratified logic, which permits a constrained use of recursion and negation while disallowing those combinations which lead to undecidable programs. A stratified program is one where it is possible to order the clauses such that for any clause containing a negated literal in its body, there is a clause later in the program that defines the negated literal. Another way of describing stratified theories makes use of directed dependency graphs. These are graphs that comprise a node for each predicate symbol appearing in the program and a directed edge from the node representing any predicate that appears in the body of a clause to the node representing the predicate defined in the head of the clause. The edges are labelled positively or negatively, where a negative symbol indicates that the predicate at the tail end of the edge appear in negated form in a clause of the program. Using this technique, a program is stratified if the dependency graph contains no cycles having a negative edge.

The concepts of stratified theories and stratification were originally developed in the context of databases [Chandra and Harel 1985] and were later adopted into the area of logic programming as described in [Apt, Blair et al. 1988; van Gelder 1988]. Programs that use stratified logic can use negation to extend their expressive power and can be evaluated in a flounder free manner. Indeed there are numerous studies that identify stratified logic as a class of first order logic that supports logic programs that are decidable [Jager and Stark 1993; Dantsin, Eiter et al. 1997]. Moreover, such programs are decidable in polynomial time

[Jajodia, Samarati et al. 1997]. A more detailed analysis of the computational complexity and expressive power of stratified logic can be found in [Dantsin, Eiter et al. 1997].

The **authorisation specification language** (ASL) [Jajodia, Samarati et al. 1997] is an example of a stratified first order logic language for specifying access control policies. 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. Although the language provides support for role-based access control, it does not scale well to large systems because there is no way of grouping rules into structures for reusability. 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 e.g. *-read* would indicate read not permitted. 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, Samarati 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.

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 logic adopted by Barker, defines a normal clause as an expression of the following form: $H \leftarrow L_1, L_2, \dots, L_m$ ($m \geq 0$). The head of the clause, H , is an atom and L_1, L_2, \dots, L_m is a conjunction of literals that constitutes the body of the clause. If the conjunction of literals L_1, L_2, \dots, L_m 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. As described previously, 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 and Rosenthal 2001] the authors 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 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 $R2$.

2.1.3 Deontic Logic

Deontic logic was developed starting in the 1950s by Von Wright [von Wright 1951], Castaneda [Castaneda 1981] and [Alchourron 1971] by extending modal logic with operators for permission, obligation and prohibition. Known as Standard Deontic Logic (SDL), traditionally it has been used in analysing the structure of normative law and normative reasoning in law. Because SDL provides a means of analysing and identifying ambiguities in sets of legal rules, there are many examples of the application of SDL to represent legislative documents [Sergot, Sadri et al. 1986; Jones and Sergot 1995]. An excellent overview of the applications of SDL is in [Wieringa and Meyer 1998].

Before looking at the details of how SDL has been applied to policy specification, it would be useful to summarise the basic axioms of the language. These are as follows:

[SDL0] Tautologies of propositional calculus

[SDL1] $O(p \rightarrow q) \rightarrow (Op \rightarrow Oq)$

If there is an obligation that p implies q , then an obligation to do p implies an obligation to do q .

[SDL2] $Op \rightarrow Pp$

If there is an obligation to do p then p is permitted.

[SDL3] $Pp \leftrightarrow \neg O\neg p$

Iff p is permitted then there is no obligation to not do p . In other words, iff p is permitted then there is no refrain policy with respect to p .

[SDL4] $Fp \leftrightarrow \neg Pp$

Iff p is forbidden then there is no permission to do p .

[SDL5] $p, (p \rightarrow q) / q$ (Modus Ponens)

If we can show that p holds and that q is implied by p , then it is possible to infer that q must hold.

[SDL6] p / Op (O-necessitation)

If we can show that p holds, then it is possible to infer that the obligation to do p also holds.

At first glance it would appear that SDL is ideally suited to specifying policy because it provides operators for all the common policy constructs like permission, prohibition (c.f. positive and negative authorisation) and obligation. However, closer examination of some of the axioms reveals inconsistencies between the definition of policy rules and the behaviour of SDL. For example, SDL2 indicates that an obligation can imply a permission and SDL3 indicates that a permission implies no obligation to not do an action. However, these implications between permissions and obligations do not exist in many management systems in which the obligation and authorisation policies may be specified independently and implemented in completely different ways.

Some of the earliest work using deontic logic for security policy representation can be found in [Glasgow, Macewen et al. 1992]. The focus of this work was to develop a means of specifying confidentiality policies together with conditional norms.

In [Cholvy and Cuppens 1997], SDL is used to represent security policies with the aim of detecting conflicts in the policy specifications. This approach is based on translating the SDL representation into first order predicate logic before performing the necessary conflict detection analysis. In an extension to this work, [Cuppens and Saurel 1996] also describes how delegation can be represented using deontic logic notation.

Ortalo describes a language to express security policies in information systems based deontic logic [Ortalo 1998]. In his approach he accepts the axiom $Pp = \neg O\neg p$ ("permitted p is equivalent to not p being not obliged") as a suitable definition of permission. As mentioned previously, this axiom is not appropriate for the modelling of obligation and authorisation policies because the two need to be separated.

An inherent problem with the deontic logic approach is the existence of a number of paradoxes. For example, Ross' paradox can be stated as $Op \rightarrow O(p \wedge q)$, i.e. if the system is obliged to perform the action *send message*, then it is obliged to perform the action *send message* or the action *delete message*. Although there is some work that offers resolutions to these paradoxes [Prakken and Sergot 1997], the existence of paradoxes can make it confusing to discuss policy specifications using deontic logic notations.

2.2 Role-Based Access Control (RBAC) Specification

Roles permit the grouping of a set of permissions related to a position in an organisation such as finance director, network operator, ward-nurse or physician. This allows permissions to be defined in terms of the position rather than the person assigned to the permission, so policies do not have to be changed when people

are reassigned to different positions within the organisation. Another motivation for RBAC has been to reuse role specification by a form of inheritance whereby one role (often a superior in the organisation) can inherit the rights of another role and thus avoid the need to repeat the specification of permissions.

Sandhu et al. [Sandhu, Coyne et al. 1996] have specified four conceptual 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 means of 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 a senior role to inherit permissions from a junior role. For example the finance director of a company inherits the permissions of the accounts manager, as the latter is the junior role. Although the propagation of permissions along role hierarchies further simplifies administration by considerably reducing the number of permissions in the system, it is not always desirable. Organisational hierarchies do not usually correspond to permission-inheritance hierarchies. The person in the senior role may not have the specific skills needed for the more junior role. For example a managing director would not usually be able to perform the functions of a systems administrator much lower down in the organisational hierarchy. Such situations lead to exceptions and complicate the specification of role hierarchies [Moffett 1998]. Another problem with RBAC is that it does not cater for multiple instances of a role with similar policies but relating to different target objects. For example there may be two different instances of a network operator role, responsible for the North and South regions of the network respectively.

$RBAC_2$ 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-dependent 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, Ferraiolo 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, Beznosov 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].

Chen et al. [Chen and Sandhu 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 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 and Sandhu 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:

$$\begin{aligned} \text{Role set: } R &= \{r_1, r_2, \dots, r_n\} \\ \text{User set: } U &= \{u_1, u_2, \dots, u_m\} \\ \text{Check-condition: } &\text{onelement}(R) \in \text{role-set}(\text{onelement}(U)) \rightarrow \\ &\text{allother}(R) \cap \text{role-set}(\text{onelement}(U)) = \emptyset \end{aligned}$$

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

2.2.1 Role Specification Languages

The Role Definition Language RDL [Hayton, Bacon et al. 1998] is based on Horn clauses and was developed as part of the Cambridge University Oasis architecture for secure interworking services. 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 2.3.4. 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, Bacon 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
```

RSL99 [Ahn and Sandhu 1999] is another role specification language which extends the ideas introduced in [Chen and Sandhu 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.

2.2.2 TRBAC

Since time is not defined as part of the state of an RBAC system as defined in [Chen and Sandhu 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, Bonatti 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.

```
(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)
```

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 example above, from [Bertino, Bonatti 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.

2.3 Other Security Specification Approaches

In this section we cover other approach to specifying security policy which includes an event-based language, the use of XML, a graphical approach and finally trust policy specification.

2.3.1 SPL

The **security policy language** (SPL) [Ribeiro, Zuquete 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, Zuquete 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, Zuquete 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 that 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, Zuquete 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;
}

```

2.3.2 XACML

XACML [OASIS 2001] 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 (entirely unrelated to the Oasis work at Cambridge described in section 2.2.1). The language permits access control rules to be defined for securely browsing XML documents that can update individual document elements. Similar to existing policy languages, XACML is used to specify a *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="http://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. The main advantage of XML is that it is a widely adopted standard with browsers to display and edit the policy specification. Although XACML supports a fine granularity of access control specification, 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 3.1

2.3.3 LaSCO

LaSCO [Hoagland, Pandey 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.

2.3.4 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 trusted 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 and Sloman 2000] and are used for applications such as web based labelling, signed email, active networks and e-commerce.

In [Blaze, Feigenbaum et al. 1998; Blaze, Feigenbaum 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 and Vimercati 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, Mass 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 and Sloman 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 having 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>

```

3 Management Policy

There has been considerable recent interest in using policies for specifying dynamically adaptable management strategies that can be easily modified to change the management approach without recoding the management system. Most of the policy-based management approaches use condition-action rules, either with or without event triggering, although some also use interpreted scripting languages. We describe these various approaches in this section.

3.1 IETF/DMTF Policy Core Information Model

The area of network policy specification has recently seen a lot of attention both from the research and the commercial communities. *Network policy* is the rules that 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. The IETF approach does not have an explicit specification of authorisation policy, but simple admission control policies can be specified by using an action to either allow or deny a message or request to be forwarded 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 ((sourceIPAddress = 192.168.12.17 AND destinationIPAddress = 192.168.24.8) OR
    (sourceIPAddress = 192.168.24.8 AND destinationIPAddress = 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 and Ellesson 1998] have been abandoned. The **policy core information model** (PCIM) [Moore, Ellesson 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 information model defines how to represent managed objects and policies in a system but does not solve the problem of actually 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 [Snir, Ramberg 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 7.

Figure 1 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 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.

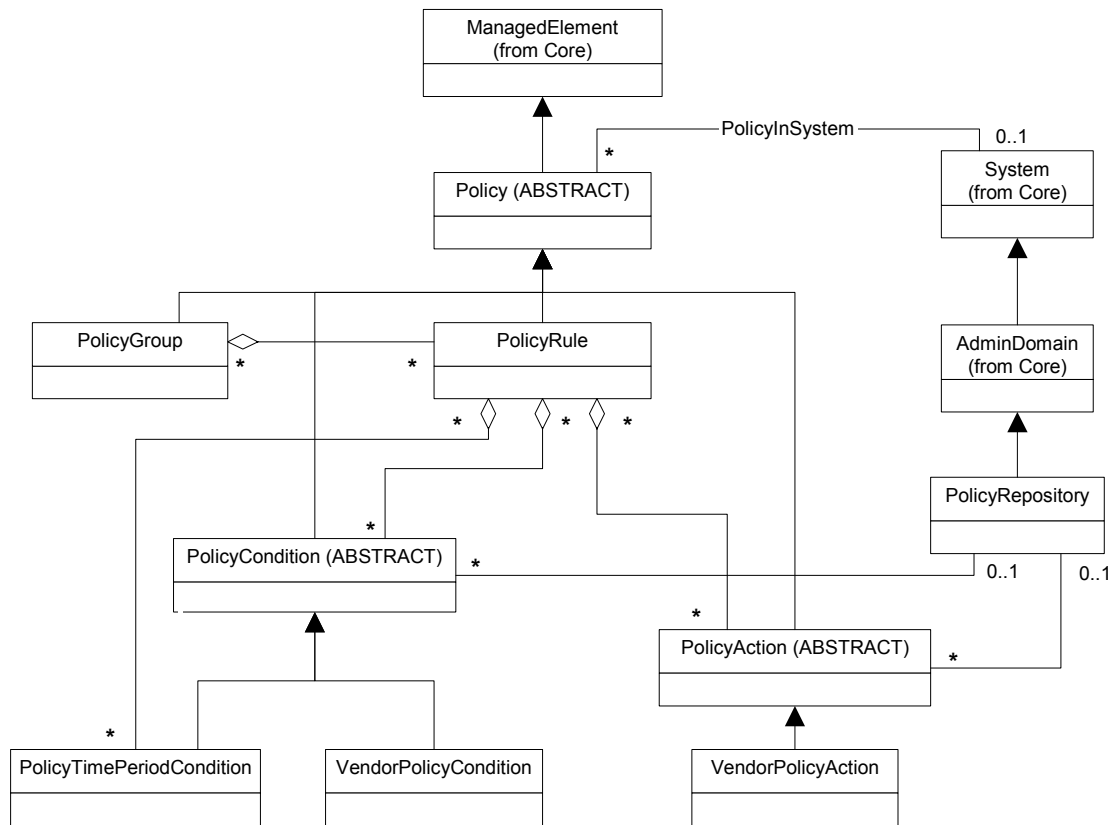


Figure 1: IETF policy core information model

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, Elleson 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, Lundy 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, Lundy 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 policy analysis. PPL does not provide any way of composing policies in groups, and there is no use of roles.

3.2 Policy Description Language (PDL)

The **policy description language** (PDL) is an event-based language from Bell-Labs [Lobo, Bhatia et al. 1999] in which 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: policy defined events, which are only generated by policy defined event propositions, and 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 and Lobo 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 overload 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, Lobo et al. 2000], and extensions to the language to specify workflows for network management can be found in [Kohli and Lobo 1999]. The language has been used to program Lucent switching products [Virmani, Lobo et al. 2000] and proves to be powerful in a variety of network operations and management scenarios.

3.3 Configuration Management

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 primarily at Unix, and to a lesser extent Windows, operating systems connected via 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 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*. The second line 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 is a powerful and concise declarative scripting language, suitable for system administrators to automate common administrative tasks on Unix systems. However, it cannot be used to specify authorisation policies and lacks support for object oriented concepts such as inheritance and parameterised instantiation. Its creators admit the need for extensions to enable enterprise-level policy specification [Burgess and Sandnes 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, Smith 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. A comparison of different approaches to implementing policies as scripts can be found in [Martinez et. al. 2002].

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

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 and Derrick 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 make 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*).

4.2 Law-Governed Interaction (LGI)

Another approach to defining policies for a distributed heterogeneous system is the proposal for Law-Governed Interaction presented in [Minsky 1991]. The motivation for this work is to provide a framework for managing the interactions between agents distributed across the network by controlling the flow of messages between these agents. In order to define the message delivery policy rules, referred to as *laws* in this approach, are specified using a simple Prolog notation. These rules are then interpreted by trusted agents, called controllers, that act as proxies for every agent in the system.

In the LGI approach, the enforcement of the law is triggered by the occurrence of events that are specified as part of each law. Known as controlled events, these could occur at any point in time and the system described in the literature does not assume any a priori knowledge of how events may be generated by the systems. A law is defined as a prescription for the behaviour of the system when it is in a given state, known as the control-state, and a controlled-event occurs. This prescription for the behaviour of the system, referred to as the ruling of the law, is denoted by the sequence of primitive operations that must be performed when the law is enforced.

A law is specified with a set of rules that define the final ruling upon the occurrence of particular events in a given state. The LGI approach specifies a *do(...)* operation that is used to add primitive operations to the final ruling of the law. The following primitive operations and controlled-events are specified in the literature.

- *sent(x, m, y)*: this event occurs when agent *x*, sends a message *m*, to agent *y*.
- *forward(x, m, y)*: this operation is performed when the law being enforced by agent *x* rules that message *m* should be sent to agent *y*.
- *arrived(x, m, y)*: this event occurs when message *m*, sent by agent *x*, arrives at agent *y*.
- *deliver(x, m, y)*: this operation is performed when the law being enforced by agent *y* rules that the message *m*, sent by agent *x*, should be delivered to agent *y*.

Of course this set can be extended to suit the particular domain in which LGI is being applied. As an example, consider a simplified token ring protocol law specification presented in [Minsky 1991]. As shown in Figure 3 the set of primitive operations is extended to include *recant(token)*, which causes the agent to clear the token property from its control state; and *set(token)*, which adds the token to the control state of the agent. Additionally, a predicate *next(D)* is used which is defined to hold when *D* is the next agent in the ring. Note that the @ symbol is used to indicate that the predicate is to be evaluated with respect to the control state of the agent.

```

R1: sent(S,M,r) :- token@,do(forward(S,M,r)).
    {To send a message to r a node must have the term token in its cState}

R2: arrived(S,M,r) :- do(deliver(t4)).
    {Any message that arrives at r is delivered.}

R3: sent(S,yourTurn,D) :- token@, next(D)@,
                        do(recant(token)),
                        do(forward(S,yourTurn,D)).
    {The owner of the token can give it up by sending the yourTurn message to the next object on the ring}

R4: arrived(S,yourTurn,D) :- do(set(token)),
                            do(deliver(memo(yourTurn))).
    {When the yourTurn message arrives at an object, the token is added to it, and an appropriate memo is delivered.}

```

Figure 3: Law specification for a token ring protocol [Minsky 1991]

In this specification, R1 states that if the `forward(...)` operation is added to the ruling and the token is present in the control state of the agent, the `sent(...)` event is generated. R2 is used to specify the policy that any message that arrives at a recipient agent is delivered. R3 specifies that in order for the `yourTurn` message to be `sent(...)`, the destination `D` must be the next node in the ring, the current agent must have the token and the law must prescribe that the agent `recant(...)` the token and `forward(...)` the `yourTurn` message. Finally, R4 specifies that the `arrived(...)` event is generated once the token is `set(...)` into the control state of the recipient agent and the `deliver(yourTurn)` operation has been added to the ruling of the law.

In this approach, permissions and prohibition are specified as a set of rules that are similar to positive and negative authorisations. Additionally, the approach supports a common global set of constraints, similar to obligation policies, which are implemented by means of filters in every node that check that all interactions are consistent with a global law [Minsky and Pal 1997]. There are other examples of the use of LGI systems to specify business rules in electronic commerce applications [Ungureanu and Minsky 2000] and to support the provision of security policies in heterogeneous systems [Ungureanu and Minsky 1998].

4.3 Event-Trigger-Rules

Work done at the University of Florida presents a specification scheme similar to the ECA rule based approaches already discussed here, with some key differences [Su, Lam et al. 2001]. The development of this approach, called Event-Trigger-Rule (ETR) paradigm, is motivated by the need for rule based processing capabilities in the distributed environment of electronic commerce enterprises [Su, Lam et al. 2001].

The ETR paradigm is a generalisation of the ECA approach where the event specification and conditions and actions of the rule are specified as separate entities. Specifying a trigger then associates the event and rule together into a policy. This is in contrast to the ECA rule specification approach, where the event specification, associated conditions and actions be combined into a single rule.

In the ETR approach, events can be classified into 3 types – method associated events, explicit events and timer events. Method events are associated with a particular method invocation and can be raised either before, after or on-commit of the method. This distinction is referred to as the coupling mode of the event. Each of these coupling modes raises synchronous events that will cause a rule to be evaluated before execution of the program continues. Additional coupling modes are instead-of (raises a synchronous event that allows the rule to replace the method invocation) and decoupled (raises an asynchronous event). Explicit events are those raised by the application during execution and timer events are those associated with a particular time of interest. Figure 4 illustrates a method associated event specification.

```

IN      InventoryManager
EVENT   update_quantity_event(String item, int quantity)
TYPE    METHOD
COUPLING_MODE BEFORE
OPERATION UpdateQuantity(String item, int quantity)

```

Figure 4: Specification of a method-associated event

A rule specifies some operations that should be performed if certain conditions apply. The conditional part of an ETR rule is defined as a guarded expression, where the guard is used to control evaluation of the conditional expression. This allows the entire rule to be skipped if any part of the guard expression evaluated to false, thus avoiding potential exception conditions (e.g. if required variables are not initialised). Additionally, the rule specifies an action block (cf. a ‘then’ block) and an alternative action block (cf. an ‘else’ block). The complete syntax of a rule specification is presented in Figure 5.

```

    RULE    rule_name(parameter list)
  [ RETURNS return_type]
  [DESCRIPTION description_text]
  [ TYPE DYNAMIC/STATIC]
  [ STATE ACTIVE/SUSPENDED]
  [ RULEVAR rule variable declarations]
  [ CONDITION guarded expression]
  [ ACTION operation block]
  [ ALTACTION operation block]
  [ EXCEPTION exception and handling block]

```

Figure 5: Syntax of the rule specification

When specifying a rule, it is possible to define local variables using the `RULEVAR` clause and also handle errors using the `EXCEPTION` clause. The `STATE` clause specifies if the rule will be active or suspended after its definition. A suspended rule will not be triggered until it is made active. The specification syntax also provides optimisation hints to the runtime environment using the `TYPE` clause. A dynamic rule can be changed at runtime whereas a static rule is less likely to be changed. This information is used when generating the runtime representation of the rule to provide optimal performance.

The final component of the ETR approach is the trigger. Triggers are used to specify which event(s) causes the processing of a particular rule. Figure 6 illustrates the syntax of a trigger.

```

    TRIGGER trigger_name(parameter list)
  TRIGGEREVENT set of event connected by OR
  [ EVENTHISTORY event expression]
  RULESTRUC set of rules
  [ RETURNS return_type: rule_in_RULESTRUC]

```

Figure 6: Syntax of the trigger specification

The `TRIGGEREVENT` clause is used to specify the set of events, combined using an `OR` connective, that will cause the rule(s) specified in `RULESTRUC` to be evaluated. The event specification can be augmented using the `EVENTHISTORY` clause to define other event expressions that need to have occurred prior to the one defined in the `TRIGGEREVENT` clause. When specifying the rules to be triggered in the `RULESTRUC` clause, it is possible to combine several rules using one of 4 constructs: sequential (rules are triggered one after the other), parallel (rules are triggered concurrently), `AND`-synchronised (all members of a set of rules must complete evaluation before another, specified, rule is triggered), and `OR`-synchronised (any two members of a set of rules must complete evaluation before another, specified, rule is triggered).

The literature that discusses the ETR approach presents many applications of this technique. These include the development of a knowledge management network [Lee 2000] and in a dynamic business process management service described in [Su, Lam et al. 2001].

Based on its similarity to the ECA rule approaches like PDL, it is easy to see how the ETR approach could be used to specify obligation policies in a distributed system. However, because of the manner in which events are defined and the ability to associate them to method invocations, it is also possible to specify authorisation policies, albeit less succinctly, using this notation. Additionally, by separating the event specifications from the rules, the ETR approach allows the user to reuse the events in multiple triggers and thus associate them with different rules as necessary. Despite the ability to specify different types of policy, and reuse parts of the specification in multiple rules, this approach does not support other useful features like policy extension (defining policies that inherit features from some parent policy) or policy groupings (organising policies that relate to the same activity together).

We now describe a language that combines many of the concepts of both security and management policy specification within a framework that supports object-oriented reuse, as well as role-based management.

5 Ponder

The Ponder language for specifying Management and Security policies [Damianou, Dulay et al. 2001] evolved out of work on policy management at Imperial College over a period of about 10 years. Ponder is a declarative, object-oriented language that can be used to specify both security and management policies. Ponder authorisation policies can be implemented using various access control mechanisms for firewalls, operating systems, databases and Java [Corradi, Montanari et al. 2000]. It supports obligation policies that are event triggered condition-action rules for policy based management of networks and distributed systems. Ponder can also be used for security management activities such as registration of users or logging and auditing events for dealing with access to critical resources or security violations. Key concepts of the language include domains to group the object to which policies apply, roles to group policies relating to a position in an organisation [Lupu and Sloman 1997b], relationships to define interactions between roles and management structures to define a configuration of roles and relationships pertaining to an organisational unit such as a department.

5.1 Domains

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. 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 objects. A domain 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 i.e. domains can overlap. An advantage of specifying policy scope in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies. Domains have been implemented as directories in an extended LDAP Service.

5.2 Ponder primitive policies

Authorisation policies define what activities a member of the subject domain can perform on the set of objects in the target domain. These are essentially access control policies, to protect resources and services from unauthorized access. 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.

The language provides reuse by supporting the definition of policy types to which any policy element can be passed as a formal parameter. Multiple instances can then be created and tailored for the specific environment by passing actual parameters as shown in Figure 7.

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

inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, Nregion/switches);
inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /Nregion/routers);
```

The two policy instances created from a *PolicyOpsT* type allow members of */NetworkAdmins* and */QoSAdmins* (subjects) to load, remove, enable or disable objects of type *PolicyT* within the */Nregion/switches* and */Nregion/routers* domains (targets) respectively.

Figure 7: Example of Ponder authorisation policies

Policies can also be declared directly without using a type as shown in the negative authorisation policy in Figure 8, which indicates the use of a time-based constraint to limit the applicability of the policy

```

inst auth- /negativeAuth/testRouters {
  subject /testEngineers/trainee ;
  action performance_test() ;
  target <routerT> /routers ;
  when time.between ("0900", "1700")
}

```

Trainee test engineers are forbidden to perform performance tests on routers between the hours of 0900 and 1700. The policy is stored within the /negativeAuth domain.

Figure 8: Direct policy declaration

Ponder also supports a number of other basic policies for specifying security policy: *Information filtering* policy can be used to transform input or output parameters in an interaction. 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. *Delegation* policy permits subjects to grant privileges, which they possess (due to an existing authorisation policy), to grantees to perform an action on their behalf e.g., passing read rights to a printer spooler in order to print a file. *Refrain* policies define the actions that subjects must refrain from performing (must not perform) on target objects even though they may actually be permitted to perform the action. Refrain policies act as restraints on the actions that subjects perform and are implemented by subjects. See [Damianou, Dulay et al. 2001] for more details and examples of these policies.

Obligation policies are event-triggered condition-action rules, similar to Lucent's PDL, and define the activities subjects (human or automated manager components) must perform on objects in the target domain. 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 loginFailure {
  on
  subject s = /NRegion/SecAdmin ;
  target <userT> t = /NRegion/users ^ {userid} ;
  do t.disable() -> s.log(userid) ;
}

```

This policy is triggered by 3 consecutive loginfail events with the same userid. 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.

Figure 9: Example Ponder obligation policy

5.3 Ponder Composite Policies

Ponder composite policies facilitate policy management in large, complex enterprises. They provide the ability to group policies and structure 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.

Roles provide a semantic grouping of policies with a common subject, generally pertaining to a position within an organisation. Ponder roles include most of the functionality of RBAC roles described in section 2.2, but include obligations. Specifying organizational 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 authorizations of that position. A role can also specify the policies that apply to an automated component acting as a subject in the system e.g. a security manager agent. Organisational positions can be represented as domains and we consider a role to be the set of authorisation, obligation, refrain and delegation policies with the *subject domain* of the role as their subject. A role is just a group of policies in which all the policies have the same subject, which is defined implicitly, as shown in Figure 10.

```

type role 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
}

```

The role type `ServiceEngineer` models a service engineer role in a mobile telecommunications service. A service engineer 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.

Figure 10: Example role policy

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 within the relationship. It thus provides an abstraction for defining policies that are not the roles themselves but are part of the interaction between the roles. The syntax of a relationship is very similar to that of a role but a relationship can include definitions of the roles participating in the relationship. However roles cannot have nested role definitions. Participating roles can also be defined as parameters within a relationship type definition as shown below.

```

type rel ReportingT (ProjectManagerT pm, SecretaryT secr) {
  inst oblig reportWeekly {
    on      timer.day ("monday") ;
    subject secr ;
    target pm ;
    do      mailReport() ;
  }
  // . . . other policies
}

```

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.

Figure 11: Example relationship type

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. A management structure is thus a composite policy containing the definition of roles, relationships and other nested management structures as well as instances of these composite policies, has some similarity to an Enterprise Community mentioned in section 4.1.

Figure 12 shows a simple management structure for a software development company consisting of a project manager, software developers and a project contact secretary. Figure 13 gives the definition of the structure.

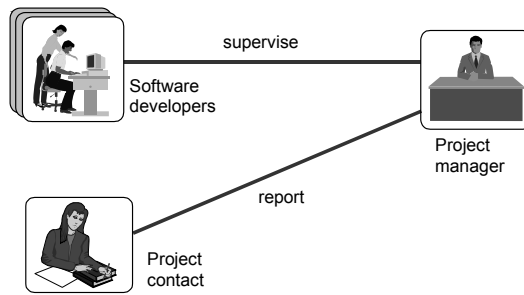


Figure 12: Simple management structure

```

type mstruct BranchT (...) {
  inst role projectManager = ProjectManagerT(...);
        role projectContact = SecretaryT(...);
        role softDeveloper = SoftDeveloperT(...);

  inst rel supervise = SupervisionT (projectManager, softDeveloper);
        rel report = ReportingT (projectContact, projectManager);
}
inst mstruct branchA = BranchT(...);
  mstruct branchB = BranchT(...);
  
```

This declares instances of the 3 roles shown in Figure 12. Two relationships govern the interactions between these roles. A supervise relationship between the softDeveloper and the projectManager, and a reporting relationship between the ProjectContact and the projectManager. Two instances of the BranchT type are created for branches within the organisation that exhibit the same role-relationship requirements.

Figure 13: Software company management structure

Ponder allows specialisation of policy types, through inheritance. When a type extends another, it inherits all of its elements, adds new elements and overrides elements with the same name. This is particularly useful for specialisation of composite policies. For example it would be possible to define a new type of mobile systems project manager, from a project manager role cf. Figure 10 with additional policies.

In Ponder a person can be assigned to multiple roles but rights from one role cannot be used to perform actions relating to another role. A person can also have policies that pertain to him/her as an individual and have nothing to do with any roles. In RBAC inheritance is based on policy instances and all policies are defined in terms of roles. This means RBAC requires a much more complicated role structure to separate the policies that are inherited from those that are private.

A compiler has been implemented for the Ponder language. Various backends have also been implemented to generate firewall rules, Windows access control templates, Java security policies [Corradi, Montanari et al. 2000] and Java obligation policy rules for interpretation by a policy agent. We also have a system to automatically disseminate policies to the relevant agents that will interpret them i.e. to subjects for obligation and refrain policies and access control agents for authorisation and filter policies.

6 Analysis and Refinement

6.1 Policy Analysis

Conflicts between policies can arise due to a specification containing policies that have opposite modalities, with the same subjects and targets, but both permit and forbid the same actions. In this situation, an agent interpreting the policies will not be able to perform an action appropriately because one policy negates the effect of the other. For a policy-based system to work effectively, it is important to have a means of detecting and resolving any conflicts that arise. In this section, we discuss different types of conflicts and presents strategies for resolving them.

A classification of policy conflicts is presented in [Lupu and Sloman 1999], which discussed both modality conflicts and application specific conflicts. Modality conflicts can be categorised into 3 distinct types:

- i) **Authorisation conflicts** arise when a positive and a negative authorisation policy is defined for the overlapping subjects, targets and actions.
- ii) **Obligation conflicts** arise when one policy obliges a subject to perform a given action whilst at the same time another policy forbids the action from being performed. In the context of Ponder, this situation would arise if an obligation and a refrain policy were defined on overlapping subjects and targets with identical actions.
- iii) **Unauthorised obligation conflicts** arises when a subject is obliged to perform an action that it does not have the authorisation to do. In a system with a default negative authorisation policy in which actions have to be explicitly authorised, this could occur if an obligation policy is defined without an associated authorisation policy.

Application specific conflicts are those that arise because of constraints defined for the particular application in which the policies are being used. For example, a system that enforces the principle of separation of duties would define a conflict if the same person who submits an expense report is also allowed to approve it.

[Jajodia, Samarati et al. 1997] identifies that conflicts can be either static or dynamic. The distinction is that analysing the syntax of a policy statement can identify static conflicts. These conflicts will occur irrespective of the state of the system enforcing the policies – this is often the case for simple modality conflicts. Dynamic conflicts are those that occur at run-time and arise because a particular state of the system results in a conflict. These are harder to detect in advance given that it is necessary to analyse the system in all possible states to do so.

Jajodia et al., proposes that a conflict, once detected could be handled in one of three ways. The most obvious and simplest one is for the system to declare an error condition whenever a conflict arises. However, this solution is not particularly interesting since it does not allow for the system to automatically recover from the conflicting scenario. Other solutions are to allow the positive policy to override; or to let the negative policy override. The latter strategy is adopting an approach of ‘do no harm’, based on the assumption that the negative policy (i.e. the one that prevents an action being performed) has a more benign effect on the system than its conflicting counterpart. As would be expected, the positive policy override strategy is the exact converse of the negative override approach described.

In addition to the negative and positive override strategies mentioned above, [Lupu and Sloman 1999] also identifies some alternatives. One approach suggested is to assign explicit priorities to every policy. This way when a conflict arises the agent enforcing the policy could simply compare the priority values and enforce the policy that has the highest priority. However, this approach could easily lead to inconsistent behaviour of the system if, as is common in distributed systems, multiple people are responsible for defining policies and assigning their priorities. Other strategies suggested include giving priority to the policy that is ‘closest’ to the managed object; or using the specificity of the policy definition to determine the priority.

Work done by Chomicki and Lobo [Chomicki, Lobo et al. 2000] describes how conflicts can arise between ECA rules and action constraints defined in the policy Description Language (PDL). Here, a policy monitor is defined to detect conflicts between the ECA rules and any action constraints. In order to resolve the conflict, the monitor will either choose to ignore certain events, thus preventing the ECA rule from activating and causing the conflict; or will cancel any actions that are specified in an action constraint. The latter scenario is an example of a negative policy override strategy.

6.2 Policy Refinement

The need for policy refinement techniques have been apparent from the outset of research into policy based systems management. [Moffet and Sloman 1993] introduces the idea of policy hierarchies and the application of policy refinement to derive lower-level, more specific policies from high-level ones. The motivation to refine policies can be defined as follows:

- To determine the resources that are required to satisfy the needs of the policy.
- To translate high-level policies into operational policies that can be enforced by the system.
- To allow analysis that would verify that the set of lower level policies actually meet the requirements of the high-level policy.

[Moffet and Sloman 1993] suggests goal refinement and arbitrary refinement of objectives as possible approaches to policy refinement. Goal refinement is a technique that has been further developed in the area of requirements engineering. Work done by Darimont et al. present patterns of goal refinement that allow high-level goals to be stated in terms of a combination of lower level ones [van Lamsweerde, Darimont et al. 1995; van Lamsweerde 1996; van Lamsweerde 1999]. In this work, by specifying goals in terms of temporal logic rules, it is demonstrated how to derive provable refinement patterns. The approach taken for proving a given pattern is to assume that each of the sub-goals holds and then show that it is possible to infer the truth of the base goal from the conjunction (or disjunction) of the sub-goals.

The goals are specified using a language called KAOS, which supports both a formal and informal definition of the system. The informal definition is specified in natural language whilst the formal definition uses the temporal logic notation introduced by [Manna and Pnueli 1992]:

X	X holds in the current state	• X,	X held in the previous state
o X,	X will hold in the next state	◆ X,	X held at some time in the past
◇ X,	X will eventually hold	■ X,	X always held in the past
□ X,	X will always hold in future	Y W X,	Y holds <i>unless</i> X holds
Y U X,	Y holds <i>until</i> X holds		

Once a refinement pattern has been derived, it can be applied to any matching scenario without the need to recreate the proof again. The following example of a goal refinement pattern is based on that presented in [Darimont and van Lamsweerde 1996].

Consider a lift control system, where the upward progress of the lift requires that the car move through consecutive floors. The goal that describes the achievement of this upward progress can be stated in KAOS as follows:

Goal Achieve [LiftUpwardProgress]
FormalDef ($\forall lc: \text{LiftCar}, f: \text{Floor}$) [$\text{At}(lc, f) \Rightarrow \diamond \text{At}(lc, f+1)$]

An achievement goal, which states that if some condition P is true then Q must eventually be true, can be represented in the temporal logic notation as $P \Rightarrow \diamond Q$ (\diamond denotes *eventually*). So in this example, we state that if a lift car lc is at floor f , then eventually that lift car must be at floor $f+1$.

A possible refinement pattern for such a goal would be to decompose it into 3 sub-goals of the form shown in Figure 14.

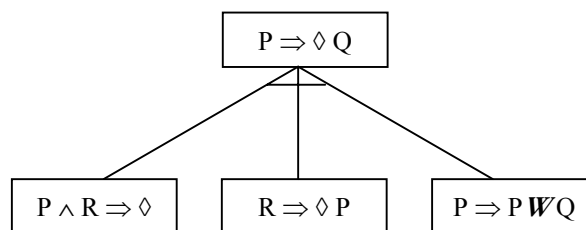


Figure 14: Achieve goal refinement pattern

The first sub-goal requires that we identify a new condition R such that if P and R are true, then Q will eventually be true. In this example, R could be the condition where the lift doors are closed on the current floor. This goal can be stated in KAOS as:

Goal Achieve [ProgressWhenDoorsClosed]
FormalDef ($\forall lc: \text{LiftCar}, f: \text{Floor}$) [$\text{At}(lc, f) \wedge \text{DoorsClosed}(lc) \Rightarrow \diamond \text{At}(lc, f+1)$]

The second and third sub-goals require that if the lift car is on floor f , then eventually the doors will be in a closed state on floor f ; and finally if the lift is on floor f , then it will continue to be there unless it is considered to be on floor $f+1$ (W denotes *unless*). This last goal captures the requirement that the motion of the lift from one floor to another is an atomic operation.

Goal Achieve [DoorsClosed]

FormalDef $(\forall lc: \text{LiftCar}, f: \text{Floor}) [\text{At}(lc, f) \Rightarrow \diamond \text{DoorsClosed}(lc)]$

Goal Achieve [LiftWaiting]

FormalDef $(\forall lc: \text{LiftCar}, f: \text{Floor}) [\text{At}(lc, f) \Rightarrow \text{At}(lc, f) \text{ } \cancel{W} \text{At}(lc, f+1)]$

The formal proof for this refinement pattern, together with a library of available goal refinements is presented in [Darimont 1995].

The underlying logic technique used in this goal refinement approach is called goal regression. Originally developed as a means of deriving plans of action for intelligent agents, goal regression provides a way of deriving the set of actions, that when applied to the system that is in some initial state, S_0 , results in the system being in some goal state S_G . This could be usefully applied in a policy refinement technique to derive a set of actions (or policies that use those actions) that have the equivalent effect on the system state as some other, high-level action.

As mentioned previously, the objective of policy refinement is to transform high-level policy specifications into more specific policies that would be better suited for use in different execution environments. Most of the above work is aimed at refining goals into implementation specifications but could be adapted to policy refinement. Before delving into the details of how policy refinement techniques work it would be useful to identify the desired properties of a refinement. In order to describe these properties a policy refinement is defined as follows:

Definition: (Policy Refinement) If there exists a set of policies $P_{rs}: p_1, p_2, \dots, p_n$, such that the enforcement of a combination of these policies results in a system behaving in an identical manner to a system that is enforcing some base policy P_b , it can be said that P_{rs} is a refinement of P_b . The set of policies $P_{rs}: p_1, p_2, \dots, p_n$ is referred to as the *refined policy set*.

Using this definition and drawing on work done to identify the properties of goal refinements [Darimont and van Lamsweerde 1996] the following properties are proposed:

1. **Correctness:** a refinement is said to be correct if there exists a subset of the refined policy set such that the conjunction of all the members of that subset is also a refinement of the base policy.
2. **Consistency:** refinement is said to be consistent if there are no conflicts between any of the policies in the refined policy set.
3. **Minimality:** a refinement is said to be minimal if it is correct and if removing any policy from the refined policy set causes the refinement to be incorrect.

A policy refinement can be said to complete iff all the properties defined above hold. The goal refinement approach also specifies a fourth property, non-trivial, that requires there to be more than one element in the refined set. However, in the policy refinement domain it may be acceptable to have a single policy that is a refinement of some base policy, provided that the refinement uses subjects, targets and actions that map to different physical entities.

It is also important to distinguish what is meant by a policy refinement *pattern* in contrast to a policy refinement as defined above. Once again it is possible to apply the concept presented in the goal refinement work of Darimont et al. [Darimont and van Lamsweerde 1996], and define that a policy refinement pattern is a *single level refinement* that *directly relates* some base policy P_b to a refined policy set P_{rs} such that P_{rs} is a complete refinement of P_b .

Other work on requirements refinement has concentrated on deriving goal-based specifications based on initial, high-level requirements documents. [Antón 1996] presents a framework for deriving goal specifications as part of the requirements engineering process. In subsequent work it is also shown how goal specifications can be derived from UML design artefacts such as use case documentation [Antón, Dempster et al. 2000].

There has also been work done to refine system architecture requirements from goal specifications. In [Brandozzi and Perry 2001], a technique for mapping KAOS specifications into an architecture model is described together with details of the Architecture Prescription Language (APL), which is used to represent the results of the transformation.

Whilst all of this work offers some useful guidelines on how develop a policy refinement technique, they cannot be directly applied to this problem domain. Indeed there are few examples of practical approaches for policy refinement. One such example is described in work done by Hewlett-Packard Laboratories, which outlines a policy-authoring environment that provides a policy wizard tool, called POWER, for refining policies [Casassa Mont, Baldwin et al. 1999]. Here, a domain expert first develops a set of policy templates that are then used by the authoring tools to constrain the possible refinements that a user can derive. These templates are expressed as Prolog programs and the policy authoring tools have an integrated inference engine that interprets these programs to guide the user through the refinement process. Finally, the POWER system provides a policy-mapping scheme for translating the refined policies into a deployable form.

Although these tools do not provide direct support for important policy analysis features such as conflict detection, we believe that the goal refinement approach outlined shows promise and merits further study.

7 Commercial Tools

Verma [Verma, Beigi 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, Baldwin 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, O'Brien 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, Kuhn 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 and space considerations prevent us from discussing each tool in detail. However, because these tools have a significant influence on the adoption of policy-based management solutions, we feel it is necessary to consider some of the major commercial policy-based network management products in greater depth.

It should be noted that the information presented here is based on public-domain documentation, available at the time of writing, from the vendors' website and industry surveys. For the latest information, surveys of commercial tools, together with product comparisons are available on the web (see <http://www-dse.doc.ic.ac.uk/Research/policies> for more information).

7.1 Nortel Optivity

Nortel's Optivity Policy Services (OPS) tool is a multi-platform network management solution. Designed to provide traffic prioritisation services, the tool supports both Nortel BayRS and Cisco IOS router platforms, and provides configuration features through a Java interface. In its initial release, policy information was stored in an Oracle backend but the approach has been dropped in favour of an LDAP schema in the current version. The documentation available does not specify the specification notation used to store the policies. However, given that the policies take the form of *if <condition> then <action>* rules, it would be reasonable to assume that their specification is based on the IETF-CIM specification approach.

Optivity supports the COPS-PR protocol for exchanging policy information between elements in the policy-based management infrastructure. The policies specified can use packet information (IP addresses, VLAN tags, protocol information etc.) together with flow rate information and scheduling to determine when a particular action should be performed. The system uses the DiffServ approach [Blake, Black et al. 1998] to implement traffic prioritisation so the actions performed on the network traffic are determined by the DiffServ codepoint that is assigned to the packet at the edge of the managed network.

7.2 Orchestream Enterprise

The Orchestream Enterprise solution is a Quality of Service management tool that leverages the DiffServ approach. Policies are specified using the IETF condition/action notation and can be stored in a LDAP

repository. In order to enhance the policy management capabilities, the Orchestream tool allows a network administrator to organise the devices to be managed in a hierarchy. This means that all the lower level devices will inherit policies specified below a given level.

Like the Nortel solution, Orchestream supports COPS for sharing policy information with other nodes in the network. Additionally, devices can be configured using SNMP, HTTP and some other proprietary protocols. Policies can be triggered based on conditions that are specified using source/destination addresses, port numbers, IP protocol type, as well as on external events that are implemented through a custom software API. The tool does not allow conditions to be based on higher-level protocol information such as MAC addresses of VLAN tags. The more recent emphasis has been on network provisioning and integration with operational support systems rather than only policy based management.

7.3 HP Openview PolicyXpert

HP's PolicyXpert tool is a multi-platform policy based management solution, designed for integration into the company's Openview network management suite. In its current release, version 2.1, PolicyXpert supports traffic management actions ranging from priority marking to DiffServ code points. Like many of the other tools considered here, policies are defined using the *if <condition> then <action>* paradigm where conditions can be based on packet information, time of day or higher-level protocol information like HTTP URL or VLAN ID. The tool supports many of the prevalent standards, including COPS, DiffServ and RSVP.

7.4 Cisco CiscoAssure

Cisco's policy based management offering, CiscoAssure Policy Manager, is also aimed at QoS service management. Although policies are specified using the condition/action approach defined by the IETF-CIM standard, the tool policies themselves are stored in a flat-file database [Conover 1999]. The user interface allows administrators to easily specify multiple conditions for triggering policies. Like the other tools considered here, conditions can be specified using a combination of IP addresses (source and destination), application ports, and the protocol being used (IP, TCP or UDP). Policy actions are applied to routers by using the Command Line Interface (CLI) language that is supported by Cisco hardware. Multi-vendor interoperability is provided with an implementation of COPS. In addition to supporting QoS related management operations, this tool allows the administrator to define access control policies for the devices being managed.

7.5 Allot Communications NetPolicy

NetPolicy aims to provide policy based management capabilities for a range of Allot Communication's network hardware in addition to Cisco routers. However, results of tests performed on an early version of this tool concluded that the Cisco support was incomplete [Conover 1999]. Once again, policies are specified using the condition/action notation, and the conditions can be defined in terms of the packet information parameters mentioned previously. The policy repository is implemented using LDAP and policy information is passed to target devices using either COPS or CLI. Additionally, NetPolicy supports management operations on simple access control lists.

7.6 Computer Associates Infrastructure Management and eTrust Solutions

As part of their UniCenter solution, Computer Associates provide a number of policy based management tools for both security and systems management operations. The eTrust Access Control module provides strong login control that can filter login access through stringent criteria including terminal ID, network/port address, day/time combinations. Implementations of the tool are available for both Unix and Windows platforms. From the available product descriptions, it appears that the Computer Associates approach specifies and stores policies using a proprietary notation.

Other than the security policy management provided by eTrust, other tools in the UniCenter suite also make use of policies. For example, the Service Level Management tool allows the user to specify policies that determine the conditions under which non-conformance to a SLA should be flagged and when reports should be published.

7.7 Tivoli Management Framework – Access Manager

The systems management framework developed by Tivoli is an extensive suite of applications that provides support for everything from configuration management to user access management. The Access Manager tool allows the administrator to configure authorisation policy templates that will map to access control lists in the underlying system. If managed objects are organised into a hierarchy (e.g. web server files system), the policies defined at a given level are propagated to lower level objects. Inherited policies can be overridden by specifying explicit policies on any given object.

Tivoli's Access Manager tool is distributed in two variations – the eCommerce Access Manager and the Operating System Access Manager. The eCommerce Access Manager is designed to manage authorisation policies for web-based resources. It integrates with other Tivoli tools to provide single log-in functionality for organisations that have multiple web applications. The Operating System Access Manager can be used to define access control policies for a range of Unix-based operating systems (Solaris, HP-UX, Linux etc).

In addition to the tools considered here, there are products from a number of companies that provide similar features. Lucent's RealNet Rules, IPHighway's Open Policy, Systor Security Administration Manager, Access360 Enforce and Spectrum Management PBNM are some examples of these. Based on our investigation of the different tools available, we can summarise their features as follows.

A common component of 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. Many 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.

8 Research Issues

Despite significant efforts in developing different policy specification techniques, there remain a number of issues to be addressed. In particular, since one of the objectives of using policy-based systems is to fulfil organisational goals, the ability to refine such goals into concrete policy specifications would be useful. As we have discussed in this paper, it is desirable to maintain the properties of correctness, consistency and minimality when performing any refinement transformation. Of course, this is only possible if the chosen policy specification technique provides support for checking whether these properties hold. To this end, we are currently investigating a mapping of Ponder specifications into a more formal, logic-based representation. In order to exploit the properties of decidability and lower computational complexity, it is intended that the formal representation would be based on first-order stratified logic. This notation could then be used in conjunction with a goal regression approach, similar to that used in goal-oriented requirements refinement [Darimont and van Lamsweerde 1996], to develop a usable policy refinement technique.

As part of solving the policy refinement problem, it will be necessary to address some of the outstanding issues related to policy analysis and conflict detection. Unless we have a means of checking for conflicts in

a policy specification, it will be impossible to maintain the required consistency property in a given refinement. In Ponder modality conflicts arise between positive and negative policies that apply to the same subjects, targets and actions [Lupu and Sloman 1999]. These can be detected by syntactic analysis of the policies as the conflict can be determined by detecting overlap of subjects, targets and actions. However, the analysis detects only potential conflicts rather than actual conflicts since constraints may limit the applicability of the policy to disjoint sets of circumstances e.g., different times of day. While modality conflicts can be detected syntactically, other conflicts can only be determined by understanding the actions being performed by the policies. For example, there will be a conflict between two policies that result in the same packet being placed on 2 different queues. Similarly, separation of duty conflicts arise from authorisation policies which permit the same person to approve payments and sign cheques. Generally, these conflicts are application specific and to detect them it is necessary to specify the conditions that result in conflict. The approach is therefore to specify constraints on the set of policies (i.e. meta-policies) using a suitable notation and then analyse the policy set against these constraints to determine if there are any conflicts [Lupu and Sloman 1999]. Whether conflicts occur or not may depend on run-time parameters specified in constraints such as time or the current state of the components to which the policy apply. It is thus rather difficult to determine all possible conflicting conditions in advance and so it is still necessary to detect conflicts at run-time. Furthermore, when conflicting policies are detected it is not obvious how to resolve the conflicts automatically. Explicit priority may work in some cases. In some situations, negative authorisation policies should override positive ones, but in other situations the positive authorisation is an exception to a more general negative authorisation. In some situations more specific policies that apply to a department may override general policies applying to the whole organisation. We have been experimenting with meta-policies that define application specific precedence relationships between conflicting policies.

Although some progress has been made in dealing with policy conflicts [Lupu and Sloman 1999; Verma 2001], significant challenges remain to be addressed. In particular, how can one detect conflicts when arbitrary conditions restrict the applicability of the policies? Sometimes, it is possible to compare restrictions placed by the constraints. For example, it is possible to detect if two time intervals overlap or if the policies apply when subjects are in different states e.g., active or standby. However, the problem remains unsolved in the general case. Other challenges concern the different levels of abstraction at which policy is specified. Conflicts between organisational goals will inevitably lead to conflicts between the policies derived from these goals. Some policies will trigger complex management procedures, which require the execution of actions that may be specified as part of different policies. This renders the task of ensuring the consistency of a policy specification much more complex.

Policies can be used to support adaptability at multiple levels in a network (i) within network-aware applications, (ii) within application-aware networks, and (iii) at the hardware level to support adaptability in the packet forwarding “fastpath” of network elements. Research is needed on defining interfaces for the exchange of policies between these levels. For example an application specific policy may be more efficiently interpreted within a network component or an application may need to adapt its behaviour as a result of adaptation within the network. However it is not easy to map the semantics of the policies between the different levels. The application may not be aware of what components exist within the network and so how can it specify policies to be interpreted by them? A similar problem arises when there is a need to interact and exchange policy information between multiple interacting services or administrative domains.

An interesting variation of the above is to consider a policy feedback loop where the system is monitored to see whether it is performing according to high-level policies or to determine changes in the systems due to faults, new applications or users appearing and hence to dynamically modify the lower level policies in order to adapt the behaviour.

9 Conclusions

In this paper, we have presented an overview of the available specification approaches for security, management and enterprise collaboration policies. Broadly speaking, many of the security policy specification languages are based on logic-based approaches whereas management and enterprise collaboration policy specification notations adopt a more informal approach. A common theme across all these specification notations is that they are specialised towards representing either security policies or management policies, not a combination of the two.

We present Ponder as an object-oriented, declarative policy specification language that allows both security and management policies to be represented. Additionally, it supports policy templates that can be configured with application specific parameters as needed and meta-policies that can be used to modify the behaviour of a policy at run-time. Ponder also allows policies to be organised according to roles, management structures and groups.

Conflict analysis and refinement are key areas for further investigation. We believe that it is necessary to define a formal, logic-based representation of policies in order to support the analysis process. Additionally, the availability of a logic-based policy notation will facilitate investigations into the use of goal regression in developing policy refinement techniques. We are developing a mapping between Ponder and a formal, logic-based representation of policy. It is hoped that the provision of this mapping will maintain the ease with which policies are defined whilst allowing more complex formal analysis and refinement techniques to be applied to the specifications.

References

- ABRAMS, M. D. 1993. Renewed Understanding of Access Control Policies. *16th National Computer Security Conference*, Baltimore, Maryland, U.S.A.
- AHN, G.-J. AND R. SANDHU 1999. The RSL99 Language for Role-based Separation of Duty Constraints. *Fourth ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, USA, ACM Press.
- ALCHOURRON, C. E. 1971. Normative Systems. New York, Wien: xvii+208.
- ANDERSON, R. J. 1996. A Security Policy Model for Clinical Information Systems. *IEEE Symposium on Security and Privacy*, Oakland, California, U.S.A.
- ANTÓN, A. I. 1996. Goal-Based Requirements Analysis. *Second IEEE International Conference on Requirements Engineering (ICRE '96)*, Colorado Springs, Colorado.
- ANTÓN, A. I., J. H. DEMPSTER, ET AL. 2000. Deriving Goals from a Use Case Based Requirements Specification for an Electronic Commerce System. *Sixth International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ)*, Stockholm, Sweden.
- APT, K. R., H. A. BLAIR, ET AL. 1988. Towards a Theory of Declarative Knowledge. *Foundations of Deductive Databases*. J. Minker. San Mateo, CA, Morgan Kaufmann: 89-148.
- BARKER, S. 2000. Security Policy Specification in Logic. *International Conference on Artificial Intelligence (ICAI00)*, Las Vegas, Nevada, USA.
- BARKER, S. 2001. Access Control Policies as Logic Programs. *London*, Imperial College of Science, Technology and Medicine.
- BARKER, S. AND A. ROSENTHAL 2001. Flexible Security Policies in SQL. *Fifteenth Annual IFIP WG 11.3 Working Conference on Database and Application Security*, Niagara on the Lake, Ontario, Canada.
- BARKLEY, J., R. KUHN, ET AL. 1998. Role-Based Access Control for the Web. *CALS Expo International & 21st Century Commerce 1998: Global Business Solutions for the New Millennium*, Long Beach, CA, USA.
- BARKLEY, J. F., K. BEZNOSOV, ET AL. 1999. Supporting Relationships in Access Control Using Role Based Access Control. *Fourth ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, USA.
- BELL, D. E. AND L. LAPADULA 1973. Secure Computer Systems: Mathematical Foundations and Model. *Bedford, MA*, MITRE Corporation.
- BERTINO, E., P. BONATTI, ET AL. 2000. TRBAC: A Temporal Role-Based Access Control Model. *5th ACM Workshop of Role-Based Access Control*, Berlin, Germany.
- BIBA, K. J. 1977. Integrity Constraints for Secure Computer Systems. *Bedford, MA*, USAF Electronic Systems Division.

- BLAKE, S., D. BLACK, ET AL. 1998. An Architecture for Differentiated Services. *Network Working Group - RFC2475*, <http://www.ietf.org/rfc/rfc2475.txt>.
- BLAZE, M., J. FEIGENBAUM, ET AL. 1999. The Role of Trust Management in Distributed Systems Security. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. New York, NY, USA, Springer-Verlag: 185 - 210.
- BLAZE, M., J. FEIGENBAUM, ET AL. 1998. Keynote: Trust Management for Public-Key Infrastructures. *Security Protocols International Workshop*, Cambridge, England, Springer-Verlag LNCS.
- BOS, H. 1999. Application-Specific Policies: Beyond the Domain Boundaries. *Sixth IFIP/IEEE International Symposium on Intergrated Network Management (IM'99)*, Boston, MA, USA.
- BOSWELL, A. 1995. Specification and Validation of a Security Policy Model. *IEEE Transactions on Software Engineering* **21**(2).
- BRANDOZZI, M. AND D. E. PERRY 2001. Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions. *First International Workshop From Software Requirements to Architectures (STRAW'01)*, Toronto, Canada.
- BREWER, D. F. C. AND M. J. NASH 1989. The Chinese Wall Security Policy. *IEEE Symposium on Research in Security and Privacy*, Oakland, California, USA, IEEE.
- BURGESS, M. 1995. A Site Configuration Engine. *USENIX Computing systems* **8**(3).
- BURGESS, M. 2001. Recent Developments in CfEngine. *Unix NL Conference*, The Hague.
- BURGESS, M. AND F. E. SANDNES 2001. Predictable Configuration Management in a Randomized scheduling Framework. *IEEE/IFIP Workshop on Distributed Systems Operations and Management (DSOM '2001)*, Nancy, France.
- CASASSA MONT, M., A. BALDWIN, ET AL. 1999. POWER Prototype: Towards Integrated Policy-Based Management. *Bristol, UK*, HP Laboratories Bristol.
- CASTANEDA, H. 1981. The Paradoxes of Deontic Logic. *New Studies in Deontic Logic: Norms, Actions and the Foundations of Ethics*. Hingham, MA, Reidel Publishing Company: 37-85.
- CHANDRA, A. AND D. HAREL 1985. Horn Clause Queries and Generalizations. *Journal of Logic Programming* **2**(1): 1-5.
- CHEN, F. AND R. S. SANDHU 1995. Constraints for Role-Based Access Control. *First ACM/NIST Role Based Access Control Workshop*, Gaithersburg, Maryland, USA, ACM Press.
- CHOLVY, L. AND F. CUPPENS 1997. Analyzing Consistency of Security Policies. *IEEE Symposium on Security and Privacy (S&P97)*, Oakland, CA, IEEE Press.
- CHOMICKI, J., J. LOBO, ET AL. 2000. A Logic Programming Approach to Conflict Resolution in Policy Management. *Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, Breckenridge, Colorado, USA, Morgan Kaufmann.
- CLARK, D. D. AND D. R. WILSON 1987. A Comparison of Commercial and Military Computer Security Policies. *IEEE Symposium on Security and Privacy*.
- CONOVER, J. 1999. Policy-Based Network Management. *Network Computing*. <http://www.networkcomputing.com/1024/1024f1.html>.
- CORRADI, A., R. MONTANARI, ET AL. 2000. A Flexible Access Control Service for Java Mobile Code. *Annual Computer Security Applications Conference (ACSAC 2000)*, New Orleans, Louisiana, USA, IEEE Press.
- CUPPENS, F. AND C. SAUREL 1996. Specifying a Security Policy: A Case Study. *Ninth IEEE Computer Security Foundations Workshop*, Co. Kerry, Ireland, IEEE Press.
- DAMIANOU, N., N. DULAY, ET AL. 2001. The Ponder Policy Specification Language. *Policy 2001: Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, Springer-Verlag.

- DANTSIN, E., T. EITER, ET AL. 1997. Complexity and Expressive Power of Logic Programming. *12th Annual IEEE Conference on Computational Complexity (CCC'97)*, Ulm, Germany, IEEE Press.
- DARIMONT, R. 1995. Process Support for Requirements Elaboration. Département d'Ingénierie Informatique. *Louvain-la-Neuve, Belgium*, Université catholique de Louvain.
- DARIMONT, R. AND A. VAN LAMSWEERDE 1996. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. *4th ACM Symposium on the Foundations of Software Engineering (FSE4)*: 179-190.
- DMTF 1999A. Common Information Model (CIM) Specification, version 2.2.
- DMTF 1999B. Specification for the Representation of CIM in XML, version 2.0.
- GLASGOW, J., G. MACEWEN, ET AL. 1992. A logic for reasoning about security. *ACM Transactions on Computer Systems (TOCS)* **10**(3): 226-264.
- GRANDISON, T. AND M. SLOMAN 2000. A Survey of Trust in Internet Applications. *IEEE Communications Surveys and Tutorials* **3**(4).
- HAYTON, R. J., J. M. BACON, ET AL. 1998. Access Control in an Open Distributed Environment. *IEEE Symposium on Security and Privacy*, Oakland, California, U.S.A.
- HERZBERG, A., Y. MASS, ET AL. 2000. Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers. *IEEE Symposium on Security and Privacy*, Oakland, California, USA.
- HOAGLAND, J. A., R. PANDEY, ET AL. 1998. Security Policy Specification Using a Graphical Approach, UC Davis Computer Science Department.
- ISO/IEC 1999. Information Technology - Open Distributed Processing Reference Model - Enterprise Viewpoint.
- JAGER, G. AND R. F. STARK 1993. The Defining Power of Stratified and Hierarchical Logic Programs. *Journal of Logic Programming* **15**(1 & 2): 55-77.
- JAJODIA, S., P. SAMARATI, ET AL. 2000. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems* **26**(2): 214-260.
- JAJODIA, S., P. SAMARATI, ET AL. 1997. A Logical Language for Expressing Authorisations. *IEEE Symposium on Security and Privacy*, Oakland, USA, IEEE.
- JONES, A. J. I. AND M. J. SERGOT 1995. A Formal Characterisation of Institutionalised Power. *Logic Journal of the IGPL* **4**(3): 429-445.
- KOHLI, M. AND J. LOBO 1999. Policy Based Management of Telecommunication Networks. *Policy Workshop 1999*, HP Labs, Bristol, UK.
- LEE, M. 2000. Event and Rule Services for Achieving a Web-based Knowledge Network. Computer and Information Science and Engineering, University of Florida.
- LOBO, J., R. BHATIA, ET AL. 1999. A Policy Description Language. *AAAI*, Orlando, Florida.
- LUPU, E. C. AND M. S. SLOMAN 1997B. Towards a Role Based Framework for Distributed Systems Management. *Journal of Network and Systems Management* **5**(1): 5-30.
- 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* **25**(6): 852-869.
- MANNA, Z. AND A. PNUELI 1992. The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag.
- Martinez, P., and Brunner M., et. al. Using the Script MIB for Policy-based Configuration Management, IEEE/IFIP Network Operations and Management (NOMS2002), Florence, April 2002.

- MINSKY, N. H. 1991. The Imposition of Protocols Over Open Distributed Systems. *IEEE Transactions on Software Engineering* **17**(2): 183-195.
- MINSKY, N. H. AND P. PAL 1997. Law-Governed Regularities in Object Systems - Part 2: A Concrete Implementation. *Theory and Practice of Object Systems (TAPOS)*, John Wiley. **2**.
- MOFFET, J. AND M. S. SLOMAN 1993. Policy Hierarchies for Distributed Systems Management. *IEEE JSAC* **11**(9 (Special Issue on Network Management)): 1404-14.
- MOFFETT, J. D. 1998. Control Principles and Role Hierarchies. *Third ACM/NIST Role Based Access Control Workshop*, Fairfax, Virginia, USA, ACM Press.
- MONT, M. C., A. BALDWIN, ET AL. 1999B. POWER Prototype: Towards Integrated Policy-Based Management. *Bristol, UK*, Extended Enterprise Laboratory, HP Laboratories.
- MOORE, B., E. ELLESSON, ET AL. 2001. Policy Core Information Model -- Version 1 Specification. *Network Working Group - RFC3060*, <http://www.ietf.org/rfc/rfc3060.txt>.
- OASIS 2001. XACML language proposal, version 0.8.
- OMG 1999B. Object Constraint Language Specification, version 1.3.
- ORTALO, R. 1998. A Flexible Method for Information System Security Policy Specification. *5th European Symposium on Research in Computer Security (ESORICS 98)*, Louvain-la-Neuve, Belgium, Springer-Verlag.
- PRAKKEN, H. AND M. J. SERGOT 1997. Dyadic Deontic Logic and Contrary-to-duty Obligations. *Defeasible Deontic Logic: Essays in Nonmonotonic Normative Reasoning*. D. Nute. Boston, Kluwer Academic Publishers. **Synthese Library: 263**: 223-262.
- RIBEIRO, C., A. ZUQUETE, ET AL. 2001A. SPL: An access control language for security policies with complex constraints. *Network and Distributed System Security Symposium (NDSS'01)*, San Diego, California.
- RIBEIRO, C., A. ZUQUETE, ET AL. 2001B. Enforcing Obligation with Security Monitors. *Third International Conference on Information and Communications Security (ICICS 2001)*, Xian, China.
- SAMARATI, P. AND S. VIMERCATI 2000. Access Control: Policies, Models, and Mechanisms. *Foundations of Security Analysis and Design (Tutorial Lectures)*. R. Focardi and R. Gorrieri, Springer -Verlag: 137-196.
- SANDHU, R., D. FERRAILOLO, ET AL. 2000. The NIST Model for Role-Based Access Control: Towards A Unified Standard. *5th ACM Workshop on Role-Based Access Control*, Berlin, Germany.
- SANDHU, R. S. 1998. Role Activation Hierarchies. *Third ACM/NIST Role Based Access Control Workshop*, Fairfax, Virginia, USA, ACM Press.
- SANDHU, R. S., E. J. COYNE, ET AL. 1996. Role-Based Access Control Models. *IEEE Computer* **29**(2): 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* **32**(9): 40-48.
- SERGOT, M. J., F. SADRI, ET AL. 1986. The British Nationality Act as a logic program. *Communications of the ACM*(29): 370-386.
- SLOMAN, M. S. 1994B. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management* **2**(4): 333-360.
- SLOMAN, M. S. 2001. Proceedings of Policy 2001: Workshop on Policies for Distributed Systems and Networks. *Policy 2001: Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, Springer-Verlag.
- SNIR, Y., Y. RAMBERG, ET AL. 2001. Policy QoS Information Model.
- SPIVEY, J. M. 1989. An Introduction to Z and Formal Specifications. *IEE/BCS Software Engineering Journal* **4**(1): 40-50.

- STEEN, M. W. A. AND J. DERRICK 1999. Formalising ODP Enterprise Policies. *3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, University of Mannheim, Germany, IEEE Publishing.
- STEEN, M. W. A. AND J. DERRICK 2000. ODP Enterprise Viewpoint Specification. *Computer Standards and Interfaces* **22**: 65-189.
- STONE, G. N., B. LUNDY, ET AL. 2001. Network Policy Languages: A Survey and a New Approach. *IEEE Network*: 10-20.
- STRASSNER, J. AND E. ELLESSON 1998. Terminology for describing network policy and services (version 00).
- STRASSNER, J., E. ELLESSON, ET AL. 2002. Policy Core LDAP Schema.
- SU, S. Y. W., H. LAM, ET AL. 2001. An Information Infrastructure and E-services for Supporting Internet-based Scalable E-business Enterprises. *5th IEEE Annual Enterprise Distributed Object Conference (EDOC2001)*, Seattle, WA, IEEE Computer Society.
- TENNENHOUSE, D. L., J. M. SMITH, ET AL. 1997. A Survey of Active Network Research. *IEEE Communications Magazine* **35**(1): 80-86.
- THOMAS, R. K. 1997. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. *Second ACM/NIST Role Based Access Control Workshop*, Fairfax, Virginia, USA, ACM Press.
- THOMSEN, D., D. O'BRIEN, ET AL. 1998. Role Based Access Control Framework for Network Enterprises. *14th Annual Computer Security Applications Conference*.
- UNGUREANU, V. AND N. H. MINSKY 1998. Unified Support for Heterogeneous Security Policies in Distributed Systems. *7th USENIX Security Symposium*, San Antonio, Texas.
- UNGUREANU, V. AND N. H. MINSKY 2000. Establishing Business Rules for Inter-Enterprise Electronic Commerce. *14th International Symposium on Distributed Computing (DISC 2000)*, Toledo, Spain, Springer-Verlag.
- VAN GELDER, A. 1988. Negation as Failure Using Tight Derivations for General Logic Programs. *Foundations of Deductive Databases*. J. Minker. San Mateo, CA, Morgan Kaufmann: 149-176.
- VAN LAMSWEEERDE, A. 1996. Divergent Views in Goal-Driven Requirements Engineering. *ACM SIGSOFT Workshop on Viewpoints in Software Development*, San Francisco, ACM.
- VAN LAMSWEEERDE, A. 1999. Goal-Oriented Requirements Analysis with KAOS (Presentation). *Policy Workshop 1999*, HP-Laboratories, Bristol, UK.
- VAN LAMSWEEERDE, A., R. DARIMONT, ET AL. 1995. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. *2nd IEEE Symposium on Requirements Engineering (RE '95)*, York, UK, IEEE Computer Society Press.
- VERMA, D., M. BEIGI, ET AL. 2001. Policy Based SLA Management in Enterprise Networks. *Policy Workshop 2001*, HP Labs, Bristol, UK, Springer-Verlag.
- VERMA, D. C. 2001. Policy-Based Networking: Architecture and Algorithms, New Riders Publishing.
- VIRMANI, A., J. LOBO, ET AL. 2000. Netmon: network management for the SARAS softswitch. *2000 IEEE/IFIP Network Operations and Management Seminar (NOMS 2000)*, Hawaii.
- VON WRIGHT, G. H. 1951. Deontic Logic. *Mind* **60**: 1-15.
- WIERINGA, R. J. AND J.-J. C. MEYER 1998. Applications of Deontic Logic in Computer Science: A Concise Overview. *Practical Reasoning and Rationality (PRR 98)*, Brighton, UK, John Wiley & Sons.