MANAGING SECURITY IN OBJECT-BASED DISTRIBUTED SYSTEMS USING PONDER

Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ
{n.damianou, nd, e.c.lupu, m.sloman}@doc.ic.ac.uk

ABSTRACT

Security management involves specification and deployment of access control policies as well as activities such as registration of users or logging and auditing events for dealing with access to critical resources or security violations. The management actions to be performed when an event occurs depend on the enterprise policy. Reusable composite policy specifications are important to cater for the complexity of large enterprise information systems. Analysing policies for conflicts is essential for the safe operation of the system. This paper describes the Ponder language for specifying policies for security management of Distributed Systems. Ponder is declarative, strongly-typed and object-oriented which makes the language flexible, scalable and adaptable to a wide range of security requirements.

1. INTRODUCTION

The advent of inter-organisational internet-based networking and services, which require integration of large enterprise information infrastructures, make the task of managing security in such systems very challenging. Many current approaches to security management focus only on access control and are not scalable or adaptable to large-scale distributed systems. The recent work on Policy Based Management of networks and distributed systems (see www-dse.doc.ic.ac.uk/policies) provides promising solutions to these problems of security management.

A policy is a rule that *defines a choice in behaviour of a system*. Separating the policy from the implementation of a system permits the policy to be modified in order to change the behaviour of a system, without changing its underlying implementation. The security community have developed a number of models relating to specification of mandatory and discretionary access control policy [3]. This has evolved into work on Role Based Access Control (RBAC) [14] and Role Based Management where a role may be considered a group of related policies pertaining to a position in an organisation [7, 9]. None of these approaches copes with all the aspects of the task of managing security in large enterprise distributed systems outlined below.

Distributed Systems are changing from the traditional client-server model to a more dynamic service-oriented paradigm. The development of end user applications and the widespread usage of data networks have created a great demand for network architectures that can rapidly adapt to new user requirements and provide customised services to the clients. Various techniques have emerged for programming network elements to support adaptable services, for example Active Networks, Mobile Agents, Management by Delegation and Policy-based quality of service management. While all these methods support programming new functionality into network elements and host devices, they increase the security concerns regarding the access to network resources and services.

We identify the following requirements for a security management policy language aimed at managing large enterprise information systems:

- Provision and support for the specification of *access control policies* relating to large systems with millions of objects. This includes support for information filtering and delegation to cater for temporary transfer of access rights.
- In very large systems, it must be possible to specify policies for *groups of objects*.
- Provision and support for monitoring, logging and auditing of events such as security violations. This includes the specification of what actions to perform in response to the events. These are the active aspects of security policy specification and take the form of *manager obligation policies*.
- *Grouping of policy specifications* is needed to form composite policies relating to roles, organisational units such as departments or to specific applications. This is essential to cater for the complexity of policy administration in large enterprise information systems.
- *Analysis of policies*: It must be possible to analyse policies for conflicts and inconsistencies, which may lead the system to insecure states. In addition it should be possible to determine which policies apply to an object or what objects a particular policy applies to. Declarative languages are easier to analyse.
- The policy specification language must be *extensible and scalable*. New security policies may arise in the future. It should be easy to add them to the language without major redesign. An object-oriented language provides a solution for this.

This paper describes Ponder [4], a declarative object-oriented policy language for security management of distributed systems. The language is flexible, expressive and extensible to cover the wide range of security requirements implied by the current distributed systems paradigms identified above. Ponder is the result of experience gained in policy-based management at Imperial College over the past 10 years [7, 10, 16, 17].

Section 2 of the paper describes domains as a means of grouping objects to which policies apply Section 3 and 4 explain the support for access control policies in Ponder and how it can be used to specify security management policies. The composite policy structures in Ponder are described in section 5. Constraints, a very important feature of the language, are described in section 6. Section 7 discusses features of the language that make it both flexible and scalable: scripts and object-orientation. In section 8 we briefly compare Ponder with related work and section 9 presents conclusions.

2. SUBJECT AND TARGET DOMAINS

We assume that all policies relate to objects with interfaces defined in terms of methods using an interface definition language. We use the term **subject** to refer to users, principles or manager agents which have management responsibility. A subject accesses **target** objects (resources or service providers) by invoking methods, so the granularity of protection for access control is an interface method. Manager agents manage target objects which provide a management interface.

In large-scale systems it is not practical to specify policies for individual objects and so there is a need to be able to group objects to which a policy applies. For example, a log policy to check the security log files at 7:00am, may apply to all security managers within a particular region. An authorisation policy may specify that all members of a department have access to a particular service. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers [16, 17]. Membership of a domain is explicit and not defined in terms of a predicate on object attributes. A domain does not encapsulate the objects it contains but merely holds references to object interfaces. 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. Details of domains are described in [16, 17].

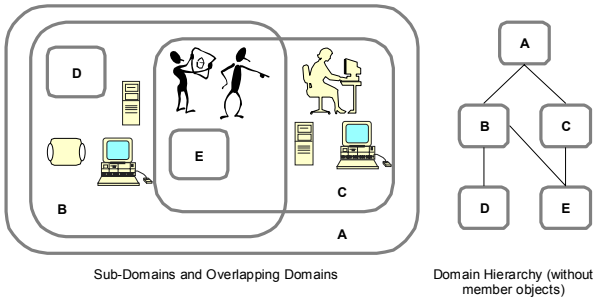

Figure 1. Domains

Path names are used to identify domains, e.g., domain E in figure 1 can be referred to as /A/B/E or /A/C/E as an object may have different local names with multiple parent domains. Policies normally propagate to members of sub-domains, so a policy applying to domain B will also apply to members of domains D and E. Domain scope expressions can be used to combine domains to form a set of objects for applying a policy, using union, intersection and difference operators.

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. A policy can select a subset of members of a domain and it's sub-domains, to which it applies, by means of a constraint in terms of object attributes.

3. ACCESS CONTROL POLICIES

Access control is concerned with limiting the activity of legitimate users who have been successfully authenticated. Discretionary Access Control (DAC) is a means of restricting access to objects based on the identity of the subjects and/or groups to which they belong [1, 15]. With DAC, access control is at the discretion of the user. The controls are discretionary in the sense that a subject with certain access permissions can pass those permissions on to any other subject. Delegation is an important part of any system supporting DAC, and is incorporated into the Ponder framework.

3.1 Authorisation

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. Constraints can be specified to limit the applicability of policies based on time or values of the attributes of the objects to which the policy refers. Constraints are discussed in section 6. A positive authorisation policy defines the actions that subjects are permitted to perform on target objects. A negative authorisation policy specifies the actions that subjects are forbidden to perform on target objects. Authorisation policies are implemented on the target host by an access control agent.

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

Members of the NetworkAdmin domain are authorised to load, remove, enable or disable objects of type Policy in the Nregion/switches domain. The domain could contain other types of objects as well.

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

Trainee test engineers are forbidden to perform performance tests on routers.

The specification of negative authorisation policies complicates the enforcement of authorisation in a system. However, there are reasons to support the provision for negative authorisation policies. Administrators often express high-level access control in terms of both positive and negative policies; retaining the natural way people express policies is important and provides greater flexibility. Negative authorisation policies can also be used to temporarily remove access rights from subjects if the need arises. In addition, many security platforms (e.g. Windows NT) include the specification of negative access rights.

3.2 Information Filtering

Positive authorisation policies may include filters to transform input or output parameters associated with their actions, based on attributes of the subject or target or on system parameters (e.g. time). For example people within a department can find out location information on a departmental member at the granularity of a room but external people can only determine whether the person is at work or out. If the same operation is used to get the information then a filter is required. Another example from the database world is when a payroll clerk is only permitted to read personnel records of employees below a particular grade. Although these are a form of authorisation policies they differ from the normal ones in that it is not possible for an external authorisation agent to make an access control decision, based on whether or not an operation, specified at the interface to the target object, is permitted. Essentially the operation has to be performed and then a decision made on whether to allow results to be returned to the subject or whether the results need to be transformed.

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

Members of *Agroup* plus *Bgroup* can set up a video conference with USA staff except the New York group. If the time is later than 5:00pm then the video-conference takes parameters: bandwidth = 2 Mb/s, priority = 1. Otherwise the first filter restricts the parameters to bandwidth = 2 Mb/s, priority = 3.

3.3 Delegation

Delegation is often used in access control systems to cater for the temporary transfer of access rights. However the ability of a user to delegate access rights to another must be tightly controlled by security policies. This requirement is critical in systems allowing cascaded delegation of access rights. A delegation policy permits subjects to grant privileges, *which they possess*, to grantees to perform an action on their behalf e.g. passing read rights to a printer spooler in order to print a file. A delegation policy is always associated with an authorisation policy which specifies the access rights that can be delegated. Negative delegation policies forbid delegation.

```
inst deleg+ (switchPolicyOps) {
    grantee    /DomainAdmin;
    action     enable(), disable() }
```

The above delegation policy accepts the *switchPolicyOps* auth+ policy from section 3.1 as a parameter. It states that the subject of that authorisation policy (Network-Admin), which is implicit in this policy, can delegate the enable and disable actions on policies from the domain */Nregion/switches* to grantees in the domain */DomainAdmin*

A Delegation policy specifies the authority to delegate, it does not control the actual delegation and revocation of access rights. It is implemented as an authorisation policy that authorises the subject (grantor) to execute the method *delegate* on the run-time system with the grantee as the parameter of the method. At run-time, when the subject executes the delegate method, a separate authorisation policy is created by trusted components of the access control system, with the grantee as the subject. Similarly the revoke method deletes or disables that second authorisation policy.

4. SECURITY MANAGEMENT POLICIES

Security management policies specify actions that must be performed when certain events occur and provide the ability to respond to changing circumstances if there is a need to do so to keep the system secure. Security management policies specify what actions must be specified when security violations occur and who must execute those actions; what auditing and logging activities must be performed, when and by whom. Security management policies can be used to handle cases of intrusion detection; policies can be set up to respond to the monitoring of security related activities, report suspicious activity and enact further surveillance or increase security measures in case of intrusions or attempted security violations. A security management policy might enable or disable access control policies accordingly to increase the degree of security provided by the system. Another example is denial of service; policies can be defined to respond to the access to resources or services and report to administrators or require managers to take certain actions when the number of simultaneous accesses has reach a specified limit. Similarly, policies may be written to alert system

administrators when disk spaces reach critical limits, thus preventing denial of service attacks.

4.1 Manager Obligation

Security management activities can be specified in Ponder with Obligation policies, which are event-triggered and define the activities subjects (human or automated manager agents) 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          3*loginfail(userid);
    subject     s = /NRegion/SecAdmin
    target      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 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.

```
inst oblig FirewallTraffic {
    on          high_reject_rate(fwHostId, eventId);
    subject     s = /firewalls/SecAdmin
    target      /externalFirewalls/fwHostId
    do          s.investigate(fwHostId) -> s.log(eventId) }
```

If the number of incoming packets that are dropped due to packet-filter restrictions at a firewall with hostId, fwhostId, is more than a predefined threshold within a small interval (say 30 seconds), a high_reject_rate event is generated by the monitoring service, since this may be cause for a security concern. This event then triggers the above obligation policy that causes the firewall security administrator to log the event and investigate it for further assessment.

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

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

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

inst oblig b3 {
    on (NoMembers/2+1)
                *votes(yes);
    subject trusted_agent;
    do enable();
    target policies/b1 }

inst oblig b4 {
    on (NoMembers/2+1)
                *votes(yes);
    subject trusted_agent;
    do ExtraMeeting();
    target chairman }
```

For the chairman example, we need an authorisation policy (b1) authorising the chairman to call an extraordinary meeting and an obligation (b2) triggered by an event generated after a majority of *yes* vote events have been received. The authorisation policy is enabled only when a trusted agent enables it (in b3). The trusted agent obligation policy is triggered by the same backing event. We acknowledge the fact that arbitrary backing policies probably require a separate scripting language to specify the backing condition.

4.2 Refrain Policies

Refrain Policies define the actions that subjects must refrain from performing (must not perform) on target objects and like obligations they are implemented by the subject. Refrain policies act as restraints on the actions that managers perform. They are in essence a form of subject-based access control. With refrain policies we can specify negative access control in much the same way we do that with negative authorisation policies, but have subjects enforce it. Refrain policies are used for situations where negative authorisation policies are inappropriate; we can not trust the targets to enforce the policies (they may not wish to be protected from the subject).

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

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

5. STRUCTURING POLICY SPECIFICATIONS

Security management in large systems with millions of objects is impossible without the ability to group security policies and structure them to reflect organisational structure, preserve the natural way system administrators operate or simply provide reusability of common definitions, easing the task of policy administrators. Ponder composite policies are used to that end.

5.1 Groups

This is a generic packaging construct to group related policies together for the purposes of policy organisation and reusability. A set of related policy specifications and related constraints are grouped together within a syntactic scope with shared declarations. This is a common concept in many programming environments. Reusability can be achieved by specifying groups as types, parameterised with any policy element and then instantiating them multiple times. The criteria for grouping policies

together may be application specific. The policies may reference the same targets, relate to the same department, concern a particular application, or have no obvious semantic relation apart from helping administrators organise and reuse specifications. For instance, the policies specified in section 4.1 to solve the simple backing example can be grouped together:

```
inst group backingGroup {
        //… the policies to handle backing go here }
```

5.2 Roles

Role is a very overloaded term within the security community. In the RBAC community a role is a collection of users and their permissions. In Ponder roles also include the duties of the managers. A Role provides a semantic grouping of policies with a common subject, generally pertaining to a position within an organisation such as department manager, project manager, analyst or ward-A nurse. 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.

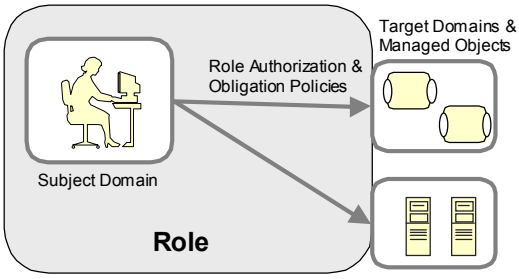

Figure 2. Roles – Subject domain

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 thus a special case of a group, in which all the policies have the same subject. A person or automated agent can then be assigned to or removed from the position domain without changing the policies as explained in [9]. See [8] for a discussion of the differences between RBAC and our Roles.

The following role example includes the specification of the subject domain after the '@'.

```
inst role SecurityManager {
        inst auth+      A1 { … }
        inst oblig      O1 { … }
        inst group      G  { … }
        …
} @ /roles/positionDomains/SM
```

5.3 Management Structures

Many large organisations are structured into units such as branch offices, departments, wards in a hospital etc., which have a similar configuration of roles and policies. Ponder supports the notion of management structures to define the configuration of policies, roles and nested management structures relating to organisational units. For example a management structure would be used to define a branch in a bank or a department in a university. The roles, groups and individual policies related with a branch/department can be grouped together in this management structure. The management structure can be specified as a type and then instantiated for different branches/departments which exhibit the same policy characteristics. Types are explained later in the paper.

6. CONSTRAINTS

An important element of each policy is the set of conditions under which the policy is valid. This information must be explicit in the specification of the policy. The validity of a policy however, may depend on other policies existing or running in the system within the same scope or context. Those conditions are usually impossible or impractical to specify as part of each policy. We need to specify those as part of a group of policies. It is thus useful to divide the constraints in two categories: constraints for single policies and constraints for groups of policies which we call meta policies. A subset of the Object Constraint Language [13] is used to specify constraints in Ponder as OCL is simple to understand and use and it is declarative – each OCL expression is conceptually atomic and so the state of the objects in the system cannot change during evaluation.

6.1 Basic-Policy Constraints

In general these limit the applicability of a policy. The constraint is expressed in terms of a predicate, which must evaluate to true for the policy to apply. Policy constraints can be considered as conjunctions of basic constraints, which can be either time-based constraints, or status-based constraints. The analysis of a set of policies can then be substantially improved since time-based constraints can be compared for possible overlap and state based constraints can be either simultaneously satisfied or mutually exclusive if they relate to states of the same system component. We separate the different types of constraints based on:

- Subject/target state – the constraint is used to select a subset of the objects in the subject or target domains based on the object state as reflected in terms of attributes at the object interface.
- Action/event parameters – constraints can be based on action or event parameter values.
- Time constraints specify the validity periods for the policy. A time library object is provided with Ponder to specify time constraints.

The policy compiler can resolve the different types of constraints at compile time and separate the constraints in order to aid in the analysability of policies.

The following is the same example policy specified in section 3.1, only this time we use a constraint on the subject status to specify it, instead of organising the trainee-test-engineers in a separate sub-domain.

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

inst auth+ filter1 { subject     /Agroup + /Bgroup;
                     target      USAStaff – NYgroup
                     action      VideoConf(BW, Priority);
                     when        time.between("1600", "1800") }
```

Members of *Agroup* plus *Bgroup* can set up a video conference with USA staff except the New York group. If the time is later than 5:00pm then the video-conference takes parameters: bandwidth = 2 Mb/s, priority = 1. Otherwise the first filter restricts the parameters to bandwidth = 2 Mb/s, priority = 3. The time-based constraint added limits the policy only between 4:00pm and 6:00pm.

6.2 Meta Policies

Meta policies specify policies about the policies within a composite policy and are used to define application specific constraints. We specify meta policies for groups of policies, i.e. policies within a specific scope, to express constraints which limit the permitted policies in the system, or disallow the simultaneous execution of conflicting policies.

Following are some examples in which meta policies can be used to specify application dependent constraints on groups of policies.

6.2.1 Self-Management

"There should be no policy authorising a manager to retract policies for which he is the subject". From [7].

This happens within a single authorisation policy with overlapping subjects and targets. Here's how the example given could be specified as a meta policy in Ponder:

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

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

6.2.2 Separation of Duty

Dynamic separation of duty can be easily specified in Ponder as constraints in authorisation policies, by accessing attributes of the objects. In the following example, the same user from the accountants domain can not both issue and authorise the same cheque.

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

Static separation of duty is handled using meta-policies since it involves constraints on groups of objects. The following static separation of duty example states that the same subject cannot both submit and approve the budget for a project in a company.

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

7. FLEXIBILITY, EXTENSIBILITY, SCALABILITY

7.1 Scripts as Actions

An obligation action can be defined as a script using any suitable scripting language to specify a complex sequence of activities or procedures with conditional branching. Scripts are implemented as objects and stored in domains. Thus policies can be specified with scripts as targets. Access to scripts can thus be controlled if required.

Scripts sometimes make it simpler to specify policies. With a script we can specify an action to move a policy from a domain /D1 to a domain /D2. Specifying an authorisation to achieve this without considering this script is problematic: there are two targets to the policy, domain D1 and domain D2. But if we use the script then we have only one target, the script itself.

```
inst auth+ A { subject /domainAdmin; action execute;
    target domainMove(A,B) ; when A="/D1" and B="/D2"}
```

Domain administrators are allowed to execute the script object 'domainMove' when the parameters to it are /D1 and /D2.

7.2 Object-orientation and Inheritance

The object-oriented model of the language provides extensibility, scalability and flexibility. Extensibility is achieved using the object model of the language (figure 3). This allows new policy types that may be identified in the future to be defined as sub-classes of existing abstract policy classes. The language provides scalability by allowing users to define policy types, instantiate them and extend them with inheritance by specialisation at an infinite depth.

Any policy element can be passed as a parameter to

policy types. This makes the language flexible since it allows for greater reuse of policy specifications; multiple instances can be created for many different conditions and under a variety of circumstances.

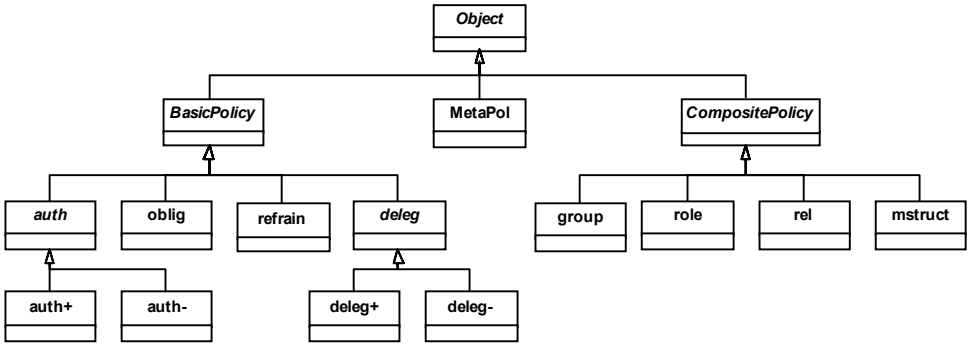

Figure 3. Ponder Object Model

The authorisation switchPolicyOps (section 3.1) can be specified as a type with the subject and target as parameters. The policy type then applies to any instance of a subject being authorised to manage policies of a domain.

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

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

The first instance allows /NetworkAdmins to execute the actions on policies within the /Nregion/switches domain. The second instance allows /QoSAdmins to execute the actions on policies within the /Nregion/routers domain.

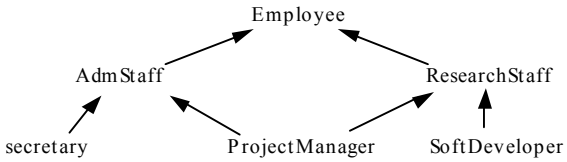

Figure 4. A role hierarchy

Ponder allows specialisation of policy types through the mechanism of inheritance. Any type can inherit from another. When a type extends another, it inherits all of its attributes, overrides attributes with the same name and can add new attributes. We gain more from the use of types and inheritance if we apply them to composite policies for maximum reusability. The role-hierarchy in figure 4 can be specified in Ponder by extending roles:

```
type role Employee(…) { … }
type role AdmStaff(…)          extends Employee { … }
type role ResearchStaff(…)     extends Employee { … }
type role Secretary(…)         extends AdmStaff { … }
type role SoftDeveloper(…)     extends ResearchStaff { … }
type role ProjectManager(…)    extends AdmStaff,
                                       ResearchStaff { … }
```

The hierarchical object model for the policy language provides a convenient means of translating policies to structured representation languages such as XML. The XML representation can then be used for viewing policy information with standard browsers or as a means of exchanging policies between different security managers or security domains.

8. RELATED WORK

There are a number of other approaches to designing a language for specifying access control policy but none include the range of policies (e.g. obligations) that we cover and most lack the scalability and extensibility features of Ponder.

Formal logic-based approaches are generally not intuitive and do not directly map onto implementation mechanisms. They assume a strong mathematical background which can make them difficult to use and understand. The ASL [6], is an example of a formal logical language for specifying access control policies. The language includes a form of meta-policies called integrity rules to specify application-dependent rules that limit the range of acceptable access control policies. Although it provides support for role-based access control, the language does not scale well to large systems because there is no way of grouping rules into structures for reusability. A separate rule must be specified for each action, there is no explicit specification of delegation and no way of specifying authorisation rules for groups of objects that are not related by type.

Another logic-based approach is that by Ortalo [12] who describes a logic language to express security policies in information systems. His approach is based on the logic of permissions and obligations, a type of modal logic called deontic logic. Standard deontic logic centres on impersonal statements instead of personal; we see the specification of policies as a relationship between explicitly stated subjects and targets instead. In his approach he accepts the axiom $Pp = \neg O \neg p$ ("permitted p is equivalent to not p being not obliged") as a suitable definition of permission. This axiom is not suitable for the modelling of obligation and authorisation policies; the two need to be separated. Miller [11] discusses several paradoxes that exist in deontic logic. Since [12] contains only syntactical extensions to deontic logic, it also suffers from the same problems.

LaSCO [5] 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 access control statements. The scope of this approach is very limited to satisfy the requirements of security management.

In [2], Chen and Sandhu introduce a language for specifying constraints in RBAC systems. It can be shown that their language is a subset of OCL and we can thus specify all of their constraints as meta-policies. Space limitations prevent further discussion of this issue.

9. CONCLUSION AND FURTHER WORK

In this paper we have presented Ponder, a language for specifying policies for security management of distributed systems. There is a serious lack of a complete approach to manage security in large enterprise information systems. Most work focuses only on access control. Ponder includes authorisation and delegation policies for specifying access control, obligation and refrain policies to specify management activity and grouping structures to structure policy specification at different levels. Its object-oriented features allow for user-defined types of policies to be specified and then instantiated multiple times with different parameters. This provides for flexibility and scalability while maintaining a structured specification that can be, in large part, checked at compile time. Meta-policies in Ponder provide a very powerful tool in specifying application specific security policies and constraints on sets of policies. Ponder is a declarative language to make analysis of policies feasible.

A policy specification toolkit is under development for defining, analysing and interpreting policies. The toolkit consists of a management console, which provides the ability to manage policies stored in a distributed policy service. The management console includes: a policy editor, an analysis tool, a domain browser and a policy-structuring tool. The back-end of the toolkit consists of: a Policy Compiler with multiple back-ends and a static policy analyser. The front-end of the compiler is complete and work continues on the back end.

The design and implementation of a generic runtime object-model for enforcement of Ponder policies on any object-based platform is under development. Some initial work has also been done on mapping Ponder policies onto different security mechanisms: access control policies on various security aware platforms such as Java Security, Windows NT Security and Firewall filters.

The language specification leaves room for future additions in many areas. For delegation policies we are currently working on extending the notation to specify constraints on the delegation (e.g. maximum delegation period, maximum number of delegation hops etc). We are also investigating sub-types of meta policies to cover concurrency constraints and user-role assignment constraints.

REFERENCES

[1] Abrams, M.D. Renewed Understanding of Access Control Policies. In Proceedings of 16th National Computer Security Conference. 1993. Baltimore, Maryland, U.S.A.

[2] Chen, F. and R.S. Sandhu. Constraints for Role-Based Access Control. In Proceedings of First ACM/NIST Role Based Access Control Workshop. 1995. Gaithersburg, Maryland, USA, ACM Press.

[3] Clark, D.D. and D.R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In Proceedings of IEEE Symposium on Security and Privacy. 1987

[4] Damianou, N., N. Dulay, E. Lupu, and M. Sloman. Ponder: A Language for Specifying Security and Management Policies for Distributed Systems. The Language Specification - Version 2.1. Research Report DoC 2000/1, Imperial College, Department of Computing, London, 3 April, 2000. Available from http://www-dse.doc.ic.ac.uk/policies/ponder.html

[5] Hoagland, J.A., R. Pandey, and K.N. Levitt. Security Policy Specificaton Using a Graphical Approach. Technical report CSE-98-3, UC Davis Computer Science Department, July 22, 1998.

[6] Jajodia, S., P. Samarati, and V.S. Subrahmanian. A Logical Language for Expressing Authorisations. In Proceedings of IEEE Symposium on Security and Privacy. 1997, pp. 31-42

[7] Lupu, E.C. A Role-Based Framework for Distributed Systems Management. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K.

[8] Lupu, E.C. and M.S. Sloman. Reconciling Role Based Management and Role Based Access Control. In Proceedings of Second ACM/NIST Role Based Access Control Workshop. 1997a. Fairfax, Virginia, USA, ACM Press.

[9] Lupu, E.C. and M.S. Sloman, Towards a Role Based Framework for Distributed Systems Management. Journal of Network and Systems Management, 1997b. 5(1): p. 5-30.

[10] Marriott, D.A. Policy Service for Distributed Systems. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K.

[11] Miller, J., HELP! How to specify policies?, Unpublished paper, available electronically from http://enterprise.shl.com/policy/help.pdf

[12] Ortalo, R. A Flexible Method for Information System Security Policy Specification. In Proceedings of 5th European Symposium on Research in Computer Security (ESORICS 98). 1998. Louvain-la-Neuve, Belgium, Springer-Verlag.

[13] Rational Rational Software Corporation, Object Constraint Language Specification, Version 1.1, Available at http://www.rational.com/uml/, September 1997.

[14] Sandhu, R.S., E.J. Coyne, H.L. Feinstein, and C.E. Youman, Role-Based Access Control Models. IEEE Computer, 1996. 29(2): p. 38-47.

[15] Sandhu, R.S. and P. Samarati, Authentication, Access Control, and Intrusion Detection. Part of the paper appeared under the title "Access Control: Principles and Practice" in IEEE Communications, 1994. 32(9): p.40-48.

[16] Sloman, M. and K. Twidle, Domains: A Framework for Structuring Management Policy. Chapter 16 in Network and Distributed Systems Management (Sloman, 1994ed), 1994a: p. 433-453.

[17] Sloman, M.S., Policy Driven Management for Distributed Systems. Journal of Network and Systems Management, 1994b. 2(4): p. 333-360.