An extended version of paper in Proc. IEEE/IFIP International Symposium on Integrated Network Management (IM'2001), Seattle, May 2001, IEEE Press.

A Policy Deployment Model for the Ponder Language

N. Dulay, E. Lupu, M. Sloman, N. Damianou
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
{nd, e.c.lupu, mss, ncd}@doc.ic.ac.uk

Abstract

Policies are rules that govern the choices in behaviour of a system. Security policies define what actions are permitted or not permitted, for what or for whom, and under what conditions. Management policies define what actions need to be carried out when specific events occur within a system or what resources must be allocated under specific conditions. There is considerable interest in the use of policies for the security and management of large-scale networks and distributed services. Existing policy work has focussed on specification, information models and application-specific policy enforcement. We address the important goal of providing a general-purpose deployment model for policies that is independent of the underlying policy enforcement mechanisms and can be employed in mixed policy environments. In this paper, we present a deployment model that is object-oriented and addresses the instantiation, distribution and enabling of policies as well as the disabling, unloading and deletion of policies. The model defines objects for policies, for domains, and for policy enforcement agent and outlines the interactions needed between them. The model also caters for changes in the memberships of domains since such changes also effect policy enforcement. The model forms part of the run-time support for Ponder; a new policy language that combines structuring ideas from object-oriented languages with a common set of policy basic types.

Keywords

Policy Based Management, Security Policy, Policy Specification Language, Policy Deployment, Policy Lifecycle, Management Domain, Policy Interpreter..

1 Introduction

There is considerable interest in **policy based management** of large enterprise data storage systems [20] and quality of service within network components in the Internet [11]. Policy based management may also provide a means of specifying the adaptive behaviour in networks to support future ubiquitous and mobile computing systems [17]. We define **policy** as a *rule governing the choices in behaviour of a managed system.* For such systems, policies provide the flexibility needed for dynamically modifying the behaviour of a system by changing policies without recoding the management components. In addition to policy based management, there has been work within different communities for specifying security policy for controlling access to resources, defining roles related to positions in organisations and for specifying trust policies related to Web access or for electronic commerce [8], [10].

Most of the current work on policy based management is focussed on support for policy specification, information models for the entities to which policies apply and policy implementation for specific application areas [7], [15]. However there has been very little work on how to disseminate policies to the entities that will interpret them and how to deal with dynamic large-scale environments where the set of objects to which policies apply change, and where the policies themselves need to be updated to cater for changing requirements. The model also serves as a reference model for building policy enforcement systems.

In this paper, we present a deployment model that supports the instantiation, distribution and enabling of policies as well as the disabling, unloading and deletion of policies. It forms part of the run-time support for Ponder [5] – a new language for specifying both management and security policies that has evolved over a number of years from the policy based management work at Imperial College [18]. We assume an object-oriented view of the underlying distributed system where interaction occurs through remote object invocations and asynchronous event notifications. Policies apply to domains of objects, and a policy applying to a domain will propagate to all objects of that domain including objects of nested subdomains. In our model, both domains and policies are mapped to objects. This allows policies to be written for the domain and policy objects in the policy management system leading to model that is capable of policy self-enforcement.

This paper outlines the deployment model that is needed to implement the Ponder concepts. We first provide an overview of the types of policies which can be specified using Ponder (section 2). Following this we present

the runtime model that describes how a policy is implemented by an object that is used to distribute, enable and otherwise coordinate management actions on the policy in a system (section 3). Section 4 outlines the runtime agents needed to enforce policy specifications while section 5 deals with how the model handles the inclusion and removal of managed objects in the system.

2 Ponder Overview

Ponder is a declarative, object-oriented language [5],[6] for specifying security policies with role-based access control, as well as general-purpose management policies for specifying what actions are carried out when specific events occur within the system or what resources to allocate under specific conditions. Unlike many other policy specification notations, Ponder supports typed policy specifications. Policies can be written as parameterised types, and the types instantiated multiple times with different parameters in order to create new policies. Furthermore, new policy types can be derived from existing policy types, supporting policy extension through inheritance.

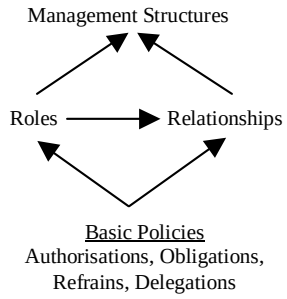

Figure 1: Policy Types

Ponder has four basic policy types: **authorisations**, **obligations**, **refrains** and **delegations** and three composite policy types: **roles**, **relationships** and **management structures** that are used to compose policies. The dependencies between the various types are shown in figure 1. Ponder also has a number of supporting abstractions that are used to define policies: **domains** for hierarchically grouping managed objects, **events** for triggering obligation policies, and **constraints** for controlling the enforcement of policies at runtime.

2.1 Basic Policies

Basic policies are defined over **sets** of objects formed by applying set operations, such as union, intersection and difference to the objects held within domains. Set operations can be restricted to apply only to the top-level members of a domain, or applied recursively to sub-domains, including all nested levels of a domain. In policy specifications, two sets have special significance, the **subject set** and the **target set**. These sets are used to represent the managed objects that the policy operates over. We use the terms *subject* and *target* to mean a single member of the subject set and target set respectively, and the terms *subjects* and *targets* to mean all members of the subject set and target set respectively.

Authorisations and Delegations

Authorisation policies are designed to protect target objects and are conceptually enforced by the target objects. In practice, authorisation policy enforcement is delegated to one or more **enforcement agents** that intercept actions and perform checks on whether the access is permitted. In our model, the enforcement agents for authorisation policies are termed **access controllers** and typically interface to lower-level access control mechanisms that really carry out the access control, for example a firewall protecting the services on its network, an operating system protecting its resources, or a database manager protecting its databases. An access controller will normally protect all the targets at its location and enforce all authorisation policies relating to them.

Ponder supports both **positive authorisations** that permit an action, and **negative authorisations** that forbid an action. Authorisations can be constrained by boolean expression and essentially handle the common functionality found in existing access control mechanisms. The treatment of policy conflicts is elaborated in [13]. The following example gives a flavour of the language. The policy states that all members of the secretaries domain are permitted to send documents for printing to spoolers in the colour printers domain, but only between 0900 and 1700 and only if the document to print is no longer than 10 pages.

```
type auth+  printing (subject S, target T, int validfrom, int validto, int maxPages) {
            action T.print (document);
            when time.between (validfrom, validto) && document.size () <= maxPages;
      }

      inst auth+  printingpolicy = printing ( /secretaries, /printers/colour, 0900, 1700, 10);
```

Positive authorisations can also specify an **authorisation filter** that allows the input and/or result parameters of an authorisation action to be modified. Filters are useful for where we wish to restrict what information an action is allowed to see or return, based on the values of the parameters and/or attributes of the subject and target of the policy, e.g. we can filter the results of a database query to ensure that no sensitive information is returned.

For delegation, Ponder takes a simple approach whereby a delegation policy can be written to permit the subjects of an authorisation policy (grantors) to delegate some or all of their access rights to a new set of

subjects (grantees). Effectively, when a grantor performs a delegation action, a new authorisation policy is created. Since this new authorisation policy is identical to the original authorisation policy except for a new subject set, the implementation and enforcement of delegated policies is the same as that for authorisation policies and is not considered further in this paper. Note that delegation does not transfer access rights from a grantor to the grantee set; grantors retain their access rights after a delegation is performed. Ponder also supports constraints on delegation policies, negative delegation policies and cascaded delegation.

Refrains and Obligations

While the subjects of an authorisation policy can be any objects that initiate invocations, the subjects of refrain and obligation policies are instances of special enforcement agents called **policy management agents** (PMAs) whose behaviour is defined by the refrain and obligation policies that apply to them (or the real-world entity that they represent). Policy management agents thus enforce all the refrain and obligation policies for a subject directly. Such agents will normally be generic; although multiple implementations are allowed since, any object that implements a PMA interface can be the subject of refrain and obligation policies.

Refrains define what actions a subject is not permitted to invoke. Refrains are similar to negative authorisations but are enforced at the subject by the policy management agent and apply to the actions that the subject invokes. Refrains are used where we do not trust the targets to enforce a policy.

In contrast to the other basic policy types, which are essentially access control policies, **obligations** are event-triggered policies that carry out management tasks on a set of target objects or on the subject itself. Obligation policies allow us to automate systems, for example, when security violations occur; when resources need to be reconfigured in response to quality-of-service degradation, etc. In the following example, when a print error event occurs, the printManager policy management agent will notify all operators of the error and log the event internally.

```
type oblig+  printManagement (subject S, target T) {
                on printError (printer, error);
                do T.notify (printer, error) ->  S.log (printer, error);
}

inst oblig  p2 = printManagement (/printManager, /operators);
```

2.2 Composite Policies

Composite policy types are used to group and inter-relate polices together in order to support **policy-specification-in-the-large** and to model the organisational structures within a system. Three types of composite policies are provided: *roles*, *relationships* and *management structures*.

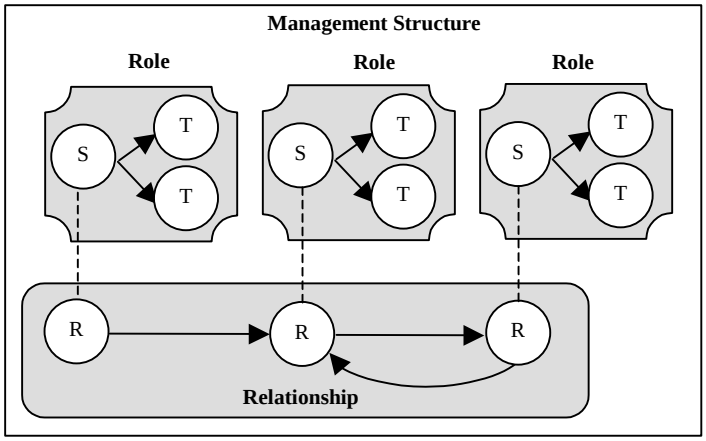

Figure 2: A management structure with 3 roles and 1 relationship

Roles allow us to group basic policies that have the same subject [12]. Since the policies within a role have the same subject, it is possible to dynamically assign or remove a subject to a role without changing the policies of that role. We formulate policies between roles with **relationships**, which are collections of basic policies that operate over the subject sets of the related roles or with respect to resources shared by the roles in the relationship. Roles and relationships can be further composed into **management structures**, and these can be grouped within other management structures, allowing a policy hierarchy to be formed that mirrors the organisational structures of a system. Figure 2 shows an example of a management structure with three roles and one relationship between the three roles with arrows representing policies.

This section has outlined the basic policy types in Ponder, authorisations, delegations, refrains and obligations, and the use of domains to define the subjects and targets of policies. Further details on the language and examples of composite policies can be found in [5].

3 Deployment Model

In our model, each policy type is compiled into a policy class by the Ponder compiler and represented by a policy object at runtime. Policy management operations are carried out by invoking methods on the policy object. The policy object maintains the state of the policy and co-ordinates all policy operations acting as single point for managing concurrent and possibly conflicting requests from multiple policy administrators and from domain objects to which the policy applies (see section 5). For most operations, the policy object will invoke corresponding operations on its underlying enforcement agents, access controllers for authorisation policies, policy management agents for refrain and obligation policies (see figure 3). If semantic constraints allow, policy objects can perform requests concurrently. The declarative nature of Ponder helps by allowing a policy to be enforced in parallel at many enforcement agents. Although in our initial implementation, policy objects are held in a central Policy Server, in large-scale systems, they can be distributed closer to the objects that they manage.

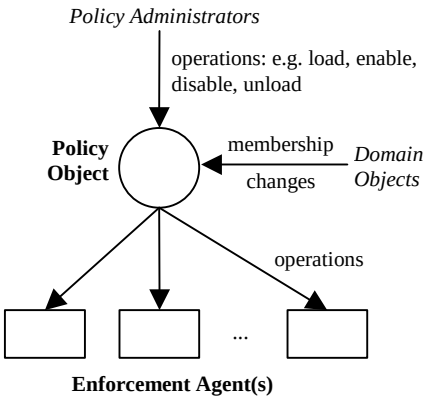

Figure 3: Policy operations

Instantiation of a basic policy creates and initialises a **policy object**; either an **authorisation policy object** (APO) or an **obligation policy object** (OPO) or a **refrain policy object** (RPO). Instantiation can be achieved in various ways:

- By running a policy administrator tool, which accesses the compiled policy classes from a policy server and instantiates them. The policy administrator tool has a graphical interface for interactive usage as well a textual command line interface for scripting.
- By executing a compiled Ponder specification. When such a specification is executed, all policy instantiation declarations within it are elaborated.
- By writing or generating a program, that calls the constructor for the policy class directly.

Policy objects are placed into one or more policy object domains. This allows them to be grouped and more importantly to have policies applied to them; for example, authorisation policies can be specified to control who has access to the actions on the policy objects.

Policy objects entrust the enforcement of policies to one or more enforcement agents: for authorisation policies to each target's **access controller** (AC), and for refrain and obligation policies to each subject's **policy management agent** (PMA). This devolution of enforcement allows for better scaling and performance. For authorisation policies, each target object has an access controller that enforces the policy while for refrain and obligation policies; each subject is an instance of a policy management agent:

Policy Object	Enforcement Agent	Number
Authorisation	Access Controller (AC)	1 AC for each target host
Obligation & Refrain	Policy Management Agent (PMA)	1 PMA for each subject object

An overview of the policy deployment model is shown in figure 4. It includes 3 supporting services: a domain service, a policy service, and an event service. The **Policy Service** acts as the interface to policy management, it stores compiled policy classes, creates and distributes new policy objects and otherwise supports policy management actions not provided elsewhere in the model. The **Domain Service** manages a distributed hierarchy of domain objects and supports the efficient evaluation of subject and target sets at run-time. Each domain object holds references to its managed objects but also references to the policy objects that currently apply to the domain [19],[22]. The Domain Service, is implemented using an LDAP server which generates events for changes to the membership of a directory and allows object to be members of more than one domain.

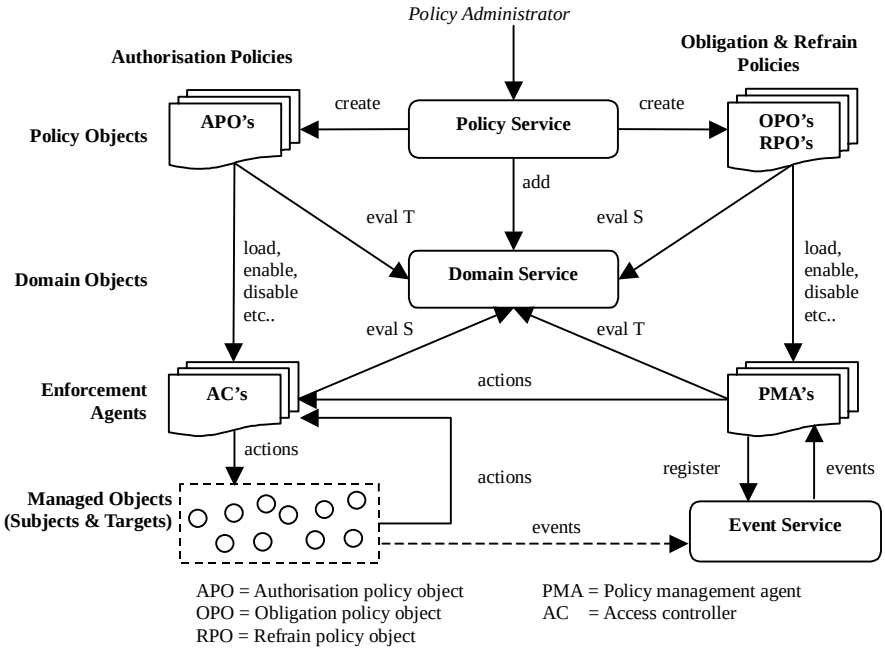

Figure 4: Overview of policy deployment model

The **Event Service** collects and composes events from the underlying systems and from the managed objects in the system, and forwards them to registered policy management agents triggering obligation policies. After a policy object is instantiated, it can be **load**ed into its enforcement agents, and once loaded, it can be **enable**d causing its enforcement agents to actively implement it. An enabled policy can be **disable**d and later re-enabled, or disabled and then **unload**ed, removing it from its enforcement agents. Unloaded (i.e. **dormant**) policies can either be re-loaded or deleted. Figure 5 shows all the states for a policy object.

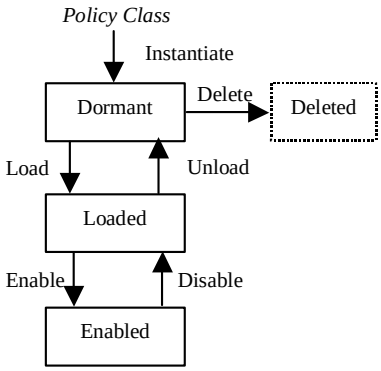

Figure 5: Policy object states

For obligation policies, PMAs register with an **event service** to receive relevant events generated from the managed objects of the system. On receiving an event, the PMA queries the domain service to determine the target objects used in the obligation method and performs the policy actions, provided no constraint or refrain policy prevents the action.

For simplicity the detailed treatment of composite policies and delegation polices is omitted in this paper. Briefly, composite policies map to objects that elaborate the instances within them while delegation policies map to authorisation policy objects that allow grantors to invoke the delegate operation on the policy service, with respect to a specific authorisation policy.

3.1 Policy Distribution

In addition to the policy class, the Ponder compiler also generates for each policy, an enforcement class that the policy object distributes to its enforcements agents. The enforcement class provides the specific implementation behaviour needed to enforce the policy at the enforcement agent. If needed, the Ponder compiler can generate multiple implementations of the enforcement classes for a single policy. This is useful where a single policy will be implemented by different enforcement agents, e.g. by an enforcement agent for Windows access control and another for Unix access control.

Authorisation Policy Objects (APOs)

For authorisation policies, each target object has a single access controller (AC), which enforces all the authorisation policies for the target object. Each AC (e.g. firewall, operating system) normally enforces many authorisation policies and protects many different target objects. We assume that the AC for a specific target object can be determined by some means, e.g. directly from the target object or from the Policy Service. The key to distributing policies is to determine what the enforcement agents are for a policy. For an authorisation policy, the policy object evaluates the target set and determines the AC for each target object in the target set. It then passes to each AC, the subset of target objects that the AC must protect (see example on the right, where t (α)

means target t at access controller α), as well as a copy of the enforcement object for the policy. On completion, the policy object retains references to each access controller.

Note that the full target set for an authorisation policy does not need to be saved by the policy object, only references to the underlying access controllers. Since policies apply to the objects in domains and objects can be dynamically added to, or removed from domains after the policy has been distributed, each domain needs to maintain references to the policies applying to it. When a domain membership changes occurs, the domain object can notify the change to its referenced policy objects (see section 5 for further details).

Policy Object	Target Set
A1	t1 (α), t2 (β), t3 (α), t4 (β)
A2	t1 (α), t2 (β), t5 (χ)

AC	Policies Enforced
α	A1 (t1, t3), A2 (t1)
β	A2 (t2, t4), A2 (t2)
χ	A2 (t5)

Obligation and Refrain Policy Objects (OPOs and RPOs)

Obligation and refrain policies are handled in a similar way to authorisation policies but the distribution is based on the subject set. The policy object evaluates the subject set and passes to each subject, a copy of the enforcement object for the policy (see example in tables). Recall that for obligation and refrain policies, that the members of the subject set are policy management agents that enforce the policies. The policy object also notifies all domains used in the subject set expression to add the policy to their current policy list, so that the domains can notify changes in their domain membership back to the policy object.

Policy Object	Subject Set
O1	s1, s2, s3
O2	s1, s2, s4
R1	s1, s2, s3, s4

PMA	Policies Enforced
s1	O1, O2, R1
s2	O1, O2, R1
s3	O1, R1
s4	O2, R1

3.2 Enabling and Disabling a Policy

Enabling a loaded policy activates the enforcement object for the policy within each of the policy's underlying enforcement agents. Once enabled, a policy is enforced until it is subsequently disabled. Disabling a policy deactivates the enforcement object within each underlying enforcement agent – however the enforcement object remains loaded in each enforcement agent and can be subsequently re-enabled. This is an overhead with loaded but not enabled policies; they take up memory and cause methods to be called when domain membership changes occur. If a policy is unlikely to be re-enabled quickly, then it may be better to unload it after disabling it. If domain membership changes are infrequent and memory abundant, or if a policy applies to many distributed objects then keeping a policy loaded but not enabled may be more efficient than unloading it and subsequently having to distribute it.

3.3 Unloading and Deleting a Policy

Unloading a policy removes the policy's enforcement object from each of the policy's enforcement agents and causes all references to the policy to be removed from the domains that it applies to. An unloaded policy object remains dormant until subsequently deleted or re-loaded. Deleting a dormant policy object removes the policy object from the Policy Service and from all domains that it is a member of.

4 Enforcement Agents

Enforcement agents are objects that implement a policy enforcement interface. This interface supports the loading, enabling and disabling of enforcement objects disseminated from policy objects. In our model enforcement objects are created by policy objects and moved to enforcement agents when the policy loading is requested. Enforcement agents store the passed enforcement objects in a local policy table and forward most policy operations to corresponding methods implemented by the enforcement object. Compiled enforcement classes provide the implementation code for enforcing policies and can be target-dependent and subject-dependent. Thus a policy may have different implementations of the policy enforcement class. In order to handle dynamic changes in domain object membership, the policy enforcement interface also has methods for the addition and removal of objects. In the next subsections we look at some of the aspects of enforcement agent implementation.

4.1 Policy Management Agents (for Obligation and Refrain Policies)

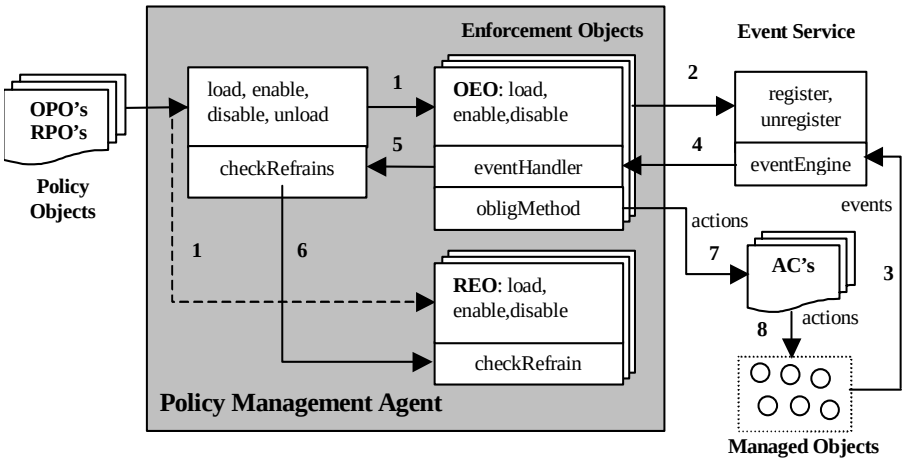

Figure 6: Overview of a Policy Management Agent

Policy management agents enforce all the enabled refrain and obligation methods for a subject. An overview of the operation of a policy management agent (PMA) is shown in figure 6. The enforcement objects for obligation policies and refrain policies (OEOs & REOs) are loaded from corresponding policy objects (OPOs & RPOs) and stored locally (1). When an obligation policy is enabled its obligation enforcement object registers the obligation event specification along with a reference to an event handler with the event service (2). The event service processes events (3) and disseminates them to handlers based on their event specifications (4). On receiving an event, handlers check both the constraints of the obligation policy and all enabled refrain enforcement objects (REOs) within the agent to check if any REO disallows actions within the obligation method (5 & 6). If constraints and refrains allow, the event handler then calls the obligation method, which performs actions on managed objects (7,8).

Two interactions are omitted from figure 6. Firstly, the event handler in the PMA queries the domain service in order to evaluate the target set on which actions are to be invoked i.e., the event handler effectively coordinates the execution of the obligation policy. Secondly, obligation policies are allowed to invoke actions internal to the PMA.

4.2 Access Controllers (for Authorisation Policies)

Access controllers enforce all the authorisation policies for one or more target objects. Access controllers are normally co-located with the targets that they protect. Unlike policy management agents, which can be generic, access controllers require close interaction with the underlying access control mechanism, for example, with the host operating system, or a firewall, or the method dispatch mechanism of a programming language. The means used to interact with each mechanism will vary. Access controllers follow the general approach for all enforcement agents. They implement the policy enforcement interface and provide methods to load, enable, disable and unload authorisation enforcement objects (AEOs) similarly to PMAs and OEOs, except that AEOs are not event-driven. Authorisation enforcement objects also provide methods for evaluating constraints and handling authorisation filters.

When an action is "intercepted" by an access controller, it calls a **checkAccess** method to check whether the access should be permitted. This method will check, for example, that the subject of the action is in the subject set of the policy, that the target of the action is in the target set of the policy and that the action is a valid action for the target. The method will also evaluate all policy constraints and enforce any global rules for the systems, for example that access to the target object must only allowed if there is a positive authorisation that allows the subject to perform the action on the target object, and no negative authorisation that forbids the subject from performing the action.

5 Domain Membership Changes

Policies in Ponder operate over domains of objects. When a policy is loaded, the set of objects that the policy applies to, is evaluated by the policy object – the subject-set for refrain and obligation policies, the target-set for authorisation policies. Domains however, are not static, and any policies that are in a loaded or enabled state need to be informed of changes to the memberships of domains to which the policy refers. In our model, domain objects that hold references to the managed objects in the domain implement domains. When a policy is loaded,

its policy object passes a reference to itself to all domain objects to which the policy applies. Domain objects can thus inform the policy objects of domain membership changes.

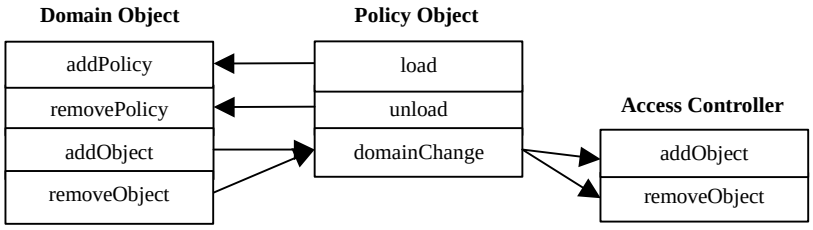

Figure 7: Domain updates

A policy that applies to a domain will also apply to subdomains (unless policy propagation is explicitly limited in the policy specification). Thus a change in the membership of a subdomain can also result in a change in the membership of parent domains. When an object is added to a domain or removed from a domain, the domain object for the domain notifies each policy object in its policy list of the change, but also requests that each parent domain do the same, with notifications passing upwards until either the root domain or a domain with no currently active policies is reached.

Note that domains can overlap and both objects and subdomains can be members of multiple domains, so changes in the membership of a domain do not always result in a change to the set of objects to which a policy applies, e.g. a domain may be used in both a set union operation and a set difference operation. The treatment of domain membership changes offers scope for optimisations. As an example, if a change results in a bigger set, then we forward references to the new objects to the appropriate enforcement agents. Conversely, if a change results in a smaller set, then we remove references to the old objects from the appropriate enforcement agents. More complex domain operations such as subdomain inclusion and removal in the presence of concurrent updates are detailed in [22].

6 Related Work

Policy-based systems have recently been the subject of increasing research effort. The standardisation work within the IETF Policy Group [11] concentrates on quality of service management and configuration within networks. They assume policies are objects stored in a directory service. A policy consumer (policy decision point – PDP) retrieves policies from the policy repository (e.g. LDAP server). A policy execution point (PEP) such as a router requests policy decisions using the Common Open Policy Service Protocol (COPS). The PEP enforces the policy for example by permitting/forbidding requests or allocating packets from a connection to a particular queue. A PEP and PDP could be combined into a single component. The IETF are defining a policy framework that can be used for classifying packet flows as well as specifying authorisations for network resources and services [14]. They do not have a language for specifying policies but are using the X500 Directory schema [15]. IETF policies are of the form ***if*** *(a set of conditions)* ***then do*** *(a set of actions)*. Directories are used for storing policies but not for grouping subjects and targets. They do not have concepts of subject and target that can be used to determine to which components a policy applies, so the mapping of policies to components has to be done by other means. There has been some discussion on the use of roles as a means of selecting policies applying to a set of components and the possibility of introducing subjects and targets into their model, but none of this has been clearly specified yet.

A number of vendors are marketing policy toolkits for defining policies, related mostly to Quality of Service for network elements [4], [9], [16]. Most of these are similar to the IETF ideas but some also support specification of a security policy. None of them support a language but they do have graphical editors that allow administrators to define individual policies and then explicitly identify the enforcement components to which the policies must be loaded. None of these tools appear to have considered the automation of the policy lifecycle.

Researchers at Bell Labs have developed a Policy Definition Language (PDL) which can be used to define policies of the form *event* ***causes*** *action* ***if*** *condition* [1],[21]. Events can be compound event expressions and the actions can be simple local or remote method calls, complex workflows or trigger other events. One policy can trigger other policies to form a hierarchical policy chain. These policies are compiled into Java classes and stored in a Directory Server. Policy service nodes (policy enforcers) load their policies from the directory and policies can be dynamically loaded or unloaded from service nodes. An administrator uses a graphical interface to display current policies and drop and load policies at run time. An administrator thus manually disseminates policies although a policy service node can automatically retrieve the policies allocated to it from the directory server on restart. These policies have been used to program the SARAS distributed softswitch and other aspects of network operations and management, but the deployment of policies is somewhat ad-hoc. PDL does not support authorisation policies and has no support for composite policies.

The Trust Management work at AT&T is building a general-purpose system to process queries of the form "*does request r, supported by credential set C, comply with policy P?*" [8], [2]. This has been used for applications such as web based labeling, signed email and active networks. The credentials could be public key certificates with anonymous identity. Both policies and credentials are predicates specified as simple C-like expression and regular expressions. Credentials could be passed in messages over an untrusted network. It is assumed that an administrator loads specific policies into application servers. The IBM Trust Establishment framework provides similar functionality aimed at e-commerce applications [10]. They use XML for specifying **trust policies** and permit negative authorization as well as positive. The credentials result in a client being assigned to a role which specifies what the client is permitted to do. Both the AT&T and IBM work only cover specification and implementation of authorization policy. They do not appear to address the policy lifecycle management.

7 Conclusions and Future work

In this paper we have presented a runtime model for deploying and managing Ponder policies in a distributed system. The model is object-based and supports the enforcement of authorisation and obligation policies using multiple and heterogeneous access control mechanisms. It cleanly separates the dissemination and management of policies from their enforcement and acts a reference model for other implementations. The model is strongly influenced by the Ponder policy specification language, which advocates the use of domains to group objects and the application of policies to domains, in order to handle system changes and policy changes. However, we believe that other policy-based systems can benefit from adopting a deployment model that is similar to ours.

The model needs to be developed further. The most interesting requirement is to cater for consistent updates in the presence of concurrent operations on the domains and policies in the system. Although many operations can be performed in parallel, the detection of possible conflicts is difficult and potentially very slow. An interesting aspect that may help here, is that the policy system, is itself, subject to policy control and policies may be written and enforced to prevent undesirable actions on the domains that hold enabled policy objects and on the policy objects themselves. This paper does not cover issues relating to refinement of high level enterprise goals or service level agreements which can be considered a requirements engineering aspect to the policy lifecycle and is being addressed in a related project at Imperial College. Others are also addressing refinement [3].

The Ponder compiler and deployment model are written in Java and we have implemented access controllers for Windows 2000, Linux, firewalls and the Java security model.

8 Acknowledgements

We gratefully acknowledge the support of EPSRC for research grants GR/L96103 (SecPol), GR/M86109 (Ponds) and GR/L76709 (Slurp) and British Telecom as part of the Alpine Project. We also acknowledge the contribution of our colleague Silvana Zappacosta.

9 References

Note: Papers and Web links are available via www-dse.doc.ic.ac.uk/policies

[1] Bhatia R., M. Kohli, J. Lobo, and A. Virmani. "A policy-based network management system". *Proc. of the International Conference on Parallel and Distributed Techniques and Applications/International Conference on Artificial Intelligence*, June 1999.

[2] M.Blaze, , J. Ioannidis, and A.D. Keromytis. "Trust Management and Network Layer Security Protocols", *Cambridge Protocols Workshop*, Springer-Verlag LNCS *1796 1999*. http://www.crypto.com/papers/networksec.pdf.

[3] M. Casassa Mont, A. Baldwin, C. Goh, "POWER Prototype: Towards Integrated Policy-Based Management", *IEEE/IFIP Network Operations and Management Symposium, (NOMS2000)*, ed. J. Hong, R., Weihmayer, Hawaii, May 2000, pp. 789-802.

[4] Cisco Assure QoS Policy Manager http://www.cisco.com/warp/public/cc/pd/nemnsw/cap/index.shtml

[5] N. Damianou, N. Dulay, E. Lupu, M.Sloman, "Ponder: A Language for specifying Security and Management Policies for Distributed Systems, V 2.3", *Imperial College Research Report DoC 2000/1* Oct. 2000.

[6] N. Damiano, Dulay N, Lupu, E, Sloman M, "The Ponder Policy Specification Language", *Proc. Policy 2001: Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, Jan. 2001, Springer-Verlag LNCS 1995

[7] Distributed Management Task Force, Inc. (DMTF), "Common Information Model (CIM) Specification", version 2.2, June 14, 1999, http://www.dmtf.org/spec/cims.html.

[8] J. Feigenbaum, "Overview of the AT&T Labs Trust Management Project: Position Paper", *Proceedings of the 1998 Cambridge University Workshop on Trust and Delegation*, Springer-Verlag LNCS.

[9] HP PolicyXpert http://www.openview.hp.com:80/products/policy/

[10] IBM. "Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers". in IEEE Symposium on Security and Privacy. 2000. http://www.hrl.il.ibm.com/TrustEstablishment/paper.asp.

[11] IETF Policy Framework workgroup http://www.ietf.org/html.charters/policy-charter.html
[12] E. Lupu, M. Sloman, "A Policy Based Role Object Model", *Proc. 1st. Enterprise Distributed Object Computing Conf. (EDOC'97)*, Australia, IEEE Press, 1997, pp36-47.
[13] E. Lupu, , M. Sloman. "Conflicts in Policy-Based Distributed Systems Management", *IEEE Trans. on Software Engineering*, 25(6): 852-869 Nov.1999.
[14] H. Mahon, Y.Bernet, S. Herzog, "Requirements for a Policy Management System", Oct. 1999, Available from http://www.ietf.org/draft-ietf-policy-req-01.txt
[15] B. Moore, J. Strassner, E. Elleson, "Policy Core Information Model", Oct 2000, Available from http://www.ietf.org/draft-ietf-policy-core-info-model-08.txt
[16] Orchestream Distributed Policy Engine http://www.orchestream.com
[17] M. Sloman, E. Lupu "Policy Specification for Programmable Networks", *Proceedings of First International Working Conference on Active Networks (IWAN'99)*, Berlin, June 1999, Springer-Verlag LNCS No. 1653, pp73-84.
[18] M. Sloman, "Policy Driven Management for Distributed Systems", *Journal of Network and Systems Management*, 2(4):333-360, Plenum Press, 1994.
[19] M. Sloman, K. Twidle, "Domains: A Framework for Structuring Management Policy". *Chap. 16 in Network and Distributed Systems Management,* M. Sloman ed., Addison-Wesley, 1994, pp. 433-453.
[20] Storage Network Industry Association (SNIA) Policy Work Group http://www.snia.org/groups/policy/index.html
[21] A. Virmani, J. Lobo, M. Kohli, "Netmon: Network Management for the SARAS Softswitch", *IEEE/IFIP Network Operations and Management Symposium, (NOMS2000)*, ed. J. Hong, R., Weihmayer, Hawaii, May 2000, pp803-816.
[22] N. Yialelis, "Domain-based Security for Distributed Operating Systems", Ph.D. Thesis, Dept of Computing, Imperial College, London, August 1996.

Appendix: Deployment Model Objects

This appendix provides a summary of the main objects described in the paper. The following conventions are used. We show object instance variables in **bold** and local variables in plaintext. Indentation is used to nest code and calls can be to remote object methods. Object parameters are passed by reference except if prefixed by the keyword **remote**. Hash tables are used throughout and can be indexed by any suitable object.

Authorisation Policy Object (APO)	Obligation Policy Object (OPO)

method *load* ( )
  TargetSet = **TargetSetExpression**.*evaluateSet* ( )
  TargetSetSize = TargetSet.*size* ( )
 	
  Foreach Object T in TargetSet Do
  AC = T.*getAccessController* ( )
  AccessControllerTable [AC].TargetSubset += {T}
 	
  Foreach AccessController AC in **AccessControllerTable** Do
  AC.*load* (**self**, **new remote** *EnforcementClass* (
  AccessControllerTable [AC].TargetSubset ...))
  AccessControllerTable [AC].TargetSubset = { }
 	
  Foreach Domain in **TargetSetExpression**.*domains* ( ) Do
  Domain.*addPolicy* (**self**)
 	
  State = LOADED
	method *load* ( ) // code is similar for RPO.load ( )
	  SubjectSet = SubjectSetExpression.*evaluateSet* ( )
	
	  Foreach PolicyManagerAgent PMA in **SubjectSet** Do
	  PMA.*load* (**self**, **new remote** *EnforcementClass* (..))
	
	  Foreach Domain in **SubjectSetExpression**.*domains* ( ) Do
	  Domain.*addPolicy* (**self**)
	
	  State = LOADED

method *enable* ( )
  Foreach AC in **AccessControllerTable** Do
  AC.*enable* (**self**)
  State = ENABLED
	method *enable* ( ) // similar for RPO's
	  Foreach PMA in **SubjectSet** Do
	  PMA.*enable* (**self**)
	  State = ENABLED

method *disable* ( )
  Foreach AC in **AccessControllerTable** Do
  AC.*disable* (**self**)
  State = LOADED
	method *disable* ( ) // similar for RPO's
	  Foreach PMA in **SubjectSet** Do
	  PMA.*disable* (**self**)
	  State = LOADED

method *unload* ( )
  Foreach AC in **AccessControllerTable** Do
  AC.*unload* (**self**)
 	
  AccessControllerTable = null
 	
  Foreach D in **TargetSetExpression**.*domains* ( ) Do
  D.*removePolicy* (**self**)
 	
  State = DORMANT
	method *unload* ( ) // similar for RPO's
	  Foreach PMA in **SubjectSet** Do
	  PMA.*unload* (**self**)
	
	  SubjectSet = { }
	
	  Foreach D in **SubjectSetExpression**.*domains* ( ) Do
	  D.*removePolicy* (**self**)
	
	  State = DORMANT

method *delete* ( )
  DomainService.*removeObject* (**self**)
  Delete **self**
	method *delete* ( )
	  DomainService.*removeObject* (**self**)
	  Delete **self**

method *domainChange* (O)
  NewSize = **TargetSetExpression**.*evaluateSet* ( ).*size* ( )
  Difference = NewSize – **TargetSetSize**
  TargetSetSize = NewSize
 	
  Switch (Difference)
  +1: AC = O.*getAccessController* ( )
  if AC notin **AccessControllerTable** then
  AC.*load* (**self**, **new** *EnforcementClass...*)
  AC.*addObject* (**self**, O)
  AccessControllerTable [AC] = { }
 	
  –1: AC = O.*getAccessController* ( )
  AC.*removeObject* (**self**, O)
	method *domainChange* (PMA)
	  NewSubjectSet = **SubjectSetExpression**.*evaluateSet* ( )
	  Difference = NewSubjectSet.*size* ( ) – **SubjectSet**.*size* ( )
	  SubjectSet = NewSubjectSet
	
	  Switch (Difference) {
	  +1: PMA.*load* (**self**, **new** *EnforcementClass* ..)
	  If **State** == ENABLED then PMA.*enable* (**self**)
	
	  –1: If **State** == ENABLED then PMA.*disable* (**self**)
	  PMA.*unload* (**self**)

Enforcement Agent (EA) : superclass of ACs & PMAs	AccessController (AC) : subclass of EA
method *load* (PO, EnforcementObject EO) **PolicyTable** [PO] = EO **PolicyTable** [PO].*load* ( ) method *enable* (PO) **PolicyTable** [PO].*enable* ( ) method *disable* (PO) **PolicyTable** [PO].*disable* ( ) method *unload* (PO) Delete entry **PolicyTable** [PO]	method *checkAccess* (Subject, Action, Target) Allows = Disallows = 0 Foreach EnforcementObject AEO in **PolicyTable** do If AEO.**State** == ENABLED && Subject in AEO.**SubjectSet**.*evaluateSet* ( ) && Target in AEO.**Targetsubset**.*currentSet* ( ) && Action in Target.**ActionList** && AEO.**Constraint**.*eval* (Subject, Action, Target) Then If AEO.type == AUTH+ then Allows ++ Elif AEO.type == AUTH– then Disallows ++ If Disallows > 0 then return DISALLOW Elsif Allows > 0 then return ALLOW Else return DISALLOW

Obligation Enforcement Object (OEO)	Domain Object (DO)
method *load* () **State** = LOADED method *enable* ( ) **EventId** = EventService.*register* (**EventSpec,** eventHandler) **State** = ENABLED method *disable* ( ) EventService.*unregister* (**EventId**) **State** = LOADED method *eventHandler* (Event) If **State** == ENABLED && **Constraint**.*evaluate* (Event) == ALLOW && **PMA**.*checkRefrains* (**ActionList**) == ALLOW Then *obligationMethod* (Event) method *obligationMethod* (Event) // execute obligation method actions	method *addPolicy* (PolicyObject PO) **PolicyObjectList**.*add* (PO) // add policy Object to list method *removePolicy* (PO) **PolicyObjectList**.*remove* (PO) method *addObject* (O) notifyPolicyObject (O) method *removeObject* (O) notifyPolicyObject (O) method *notifyPolicyObject* (O) Foreach PO in **PolicyObjectList** Do PO.*domainChange* (O) Foreach Domain PD in **ParentList** Do PD.*notifyPolicyObject* (O)