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. 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. 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.

2 Ponder Policy Specification Language

The Ponder language for specifying Management and Security policies 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. 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. It provides a common unified framework for specifying management policy for heterogeneous platforms. Key concepts of the language include domains to group the objects to which policies apply, roles to group policies relating to a position in an organisation, 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.

2.1 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.

2.2 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 1.

\[
\text{type auth+ PolicyOpsT (subject s, target PolicyT t) \{} \\
\quad \text{action load(), remove(), enable(), disable();} \\
\text{inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, Nregion/switches);} \\
\text{inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /Nregion/routers);}
\]

The two policy instances created from a \text{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 1: Example of Ponder authorisation policies**

Policies can also be declared directly without using a type as shown in the negative authorisation policy in Figure 2, which indicates the use of a time-based constraint to limit the applicability of the policy

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

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 2: Direct policy declaration**

Ponder also supports a number of other basic policies for specifying security policy: Information filtering policies 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.

Obligation policies are event-triggered condition-action rules which 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 3*loginfail(userid) ;
    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 3: Example Ponder obligation policy

2.3 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. 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 4.

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 4: 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 5: 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.

Figure 6 shows a simple management structure for a software development company consisting of a project manager, software developers and a project contact secretary. Figure 7 gives the definition of the 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 6. 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 7: 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 4 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.

3 PONDER TOOLKIT

3.1 Domain Browser

The PONDER domain browser provides a common user interface for all aspects of an integrated management environment. It can be used to group or select objects for applying policy, to monitor them or to perform management operations, although the current implementation only supports policy management.

The domain browser reads data from the domain service and provides a graphical tree-structured view of the directory structure. Usability is enhanced by customising pop-up menus according to the type of the object being selected. Furthermore, external tools can be invoked from within the domain browser for a specific managed resource or policy, depending on the current selected context. External tools also interface with the domain browser to allow for navigation or selection of objects from the domain service. For example, it is possible to specify a policy’s subject and target domains by selecting them from the domain browser. A domain structure is created using the domain browser as shown in Figure 8. Administrators can use the domain browser to manage the domain structure, group objects into domains to apply a common policy, modify or create new objects. Objects can represent users, roles, network components or manager agents.
3.2 Compiler Framework

The PONDER compiler maps policies to low-level representations suitable for the underlying system or into XML for transfer around the network.

Authorization policies can be mapped onto a variety of heterogeneous security platforms and mechanisms, such as firewalls, operating systems security, database security and Java authorizations. For example, if servers used to store data in the AI research group are Linux based while servers in other departments are Windows 2000 based, then appropriate code will be generated based on the type of server.

Dedicated code generators (compiler back-ends) must be implemented to translate the PONDER specification into the desired format. The compiler framework is designed for extensibility with custom code generators without recompiling the system. Preliminary implementations exist for translating PONDER policies onto various access control platforms. These include:

- A Java back-end which transforms PONDER authorization policies into access control policies for the Java platform. This has required several extensions to the Java security model in order to enable run-time PONDER policy evaluation, constraint checking and filtering.
- Experimentation with mapping PONDER authorization policies to Linux access controls. System level scripts have been specified to program the Linux security kernel. A code generator translates PONDER policies into calls on those scripts.

3.3 Policy Editor

The policy editor tool (Figure 10) is integrated with both the domain browser and the PONDER compiler and provides an easy to use development environment for specifying, reviewing and modifying policies.
Templates can be used to create policies easily. The domain browser can be invoked to select the subject and target domains for policies. Existing policies and policy types can be selected from the directory with the aid of the domain browser, loaded into the editor, modified, recompiled and stored back to the directory. Code generators added to the compiler framework, are accessible and can be enabled from within the editor to select the type of code to be generated.

Figure 10 Policy Editor

3.4 Management Console Tool

We have implemented a management console tool for dynamically managing policies. The steps involved in using the tool in relation to the policy life-cycle are demonstrated in Figure 11. The tool has two main views:

In the Policy Objects View, a policy instance can be selected from the directory (using the domain browser) and loaded into the management console. Similarly, if a domain is selected all policy instances under that domain will be loaded into the management console in an expandable tree-navigator. Policies can then be selected and loaded, unloaded, enabled or disabled as needed. Details about the selected policy are displayed including the policy-status. When a new backup policy for a specific user is specified, a policy administrator uses the management console to select the policy from the directory, load it and enable it. Multiple
management consoles could manage the same domain of policy objects, but LDAP does not support concurrency control.

In the **Enforcement Components View**, enforcement components can be selected and information about the policies loaded into them is displayed in a tabular format.

A **Command-line Window** can be used to type single-line commands to the PONDER compiler. This allows interactive instantiation of policy types.

![Figure 12 The Management Console Tool](image)

4 **ADDITIONAL INFORMATION**

The Ponder toolkit can be downloaded under a GNU Lesser General Public License from [http://www-dse.doc.ic.ac.uk/Research/policies/index.shtml](http://www-dse.doc.ic.ac.uk/Research/policies/index.shtml)

The following papers are available from [http://www-dse.doc.ic.ac.uk/~mss/MSSPubs.html](http://www-dse.doc.ic.ac.uk/~mss/MSSPubs.html)


