Proc. Policy 2001: Workshop on Policies for Distributed Systems and Networks,
Bristol, UK, 29-31 Jan. 2001, Springer-Verlag LNCS 1995, pp. 18-39

The Ponder Policy Specification Language

Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ
{n.damianou, n.dulay, e.c.lupu, m.sloman}@doc.ic.ac.uk

Abstract. The Ponder language provides a common means of specifying security policies that map onto various access control implementation 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. Key concepts of the language include 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. These reusable composite policy specifications cater for the complexity of large enterprise information systems. Ponder is declarative, strongly-typed and object-oriented which makes the language flexible, extensible and adaptable to a wide range of management requirements.

1 Introduction

Large enterprise information infrastructures have to integrate inter-organisational networks and internet-based services, which makes the task of managing such systems very challenging. The development of mobile computing applications requires support from adaptive network architectures and customised services to the clients. Various techniques have emerged for programming network elements to support adaptive services, for example active networks, mobile agents and management by delegation. While all these approaches support the programming of new functionality into network elements and host devices, they increase the security concerns regarding access to network resources and services, and make the management task even more demanding.

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. In this work a *policy is a rule that defines a choice in the behaviour of a system*. Separating the policy from the implementation of a system permits the policy to be modified in order to dynamically change the strategy for managing the system and hence modify the behaviour of a system, without changing its underlying implementation [26].

There are a number of groups working on very different approaches to specifying policy. Network component manufacturers and the IETF/DMTF are concentrating on information models [6][20] and condition-action rules with the focus on the management of Quality of Service (QoS) in networks [7][9][11][17]. The security community have developed a number of models relating to specification of mandatory and discretionary access control policy [4]. This has evolved into work on **role based access control** (RBAC) [24] and role based management where a role may be considered as a group of related policies pertaining to a position in an organisation [15][16]. A lot of work within the greater scope of management has already resulted in architectures and technologies that provide the basic infrastructure required to implement policy-based management solutions [8][27].

Separate tools are emerging for policy-based management of systems and specifying security. What is lacking is a common language that will provide a unified approach to supporting the concepts of the policy models emerging from the various research communities. We identify the following requirements for a policy language:

- Support for security policies for access control, and delegation to cater for temporary transfer of access rights to agents acting on behalf of a client as well as policies to express management activity.
- Structuring techniques to facilitate the specification of policies relating to large systems with millions of objects. This implies the need for policies relating to collections of objects rather than individual ones.
- Composite policies which allow the basic security and management policies relating to roles, to organisational units and to specific applications to be grouped. Composite policies are essential to cater for the complexity of policy administration in large enterprise information systems.
- It must be possible to analyse policies for conflicts and inconsistencies in the specification. In addition it should be possible to determine which policies apply to an object or what objects a particular policy applies to. Declarative languages make such analysis easier.
- Extensibility is needed to cater for new types of policy that may arise in the future and this can be supported by inheritance in an object-oriented language.
- The language must be comprehensible and easy to use by policy users.

This paper describes Ponder [5], a declarative, object-oriented language for specifying security and management policy for distributed object systems. The language is flexible, expressive and extensible to cover the wide range of requirements implied by the current distributed systems paradigms identified above. Ponder is the result of experience gained in policy-based management at Imperial College over the past 10 years [15][16][14][18][26]. We present the language syntax through simple examples of its use; for the complete syntax of the language see [6].

Sections 2 and 3 present the basic policy types supported by Ponder. Constraints are described in section 4. The composite policy structures in Ponder are described in section 5. Section 6 discusses features that make the language both flexible and extensible. In section 7 we briefly compare Ponder with related work and section 8 presents conclusions and future work.

2 Access Control Policies

Access control is concerned with limiting the activity of legitimate users who have been successfully authenticated [1][23]. Our emphasis has been on non-discretionary access control (as defined in [1]), where administrators have the authority to specify security policies that are enforced by the access control system. Delegation and propagation of authority are permitted only within the scope defined by the security policy. However, this does not exclude the use of Ponder to specify discretionary or mandatory security policies. Ponder supports access control by providing **authorisation, delegation, information filtering** and **refrain** policies as described below.

We assume that all policies relate to objects with interfaces defined in terms of methods using an interface definition language. We use the term **subject** to refer to users, principals or automated manager components, which have management responsibility. A subject accesses **target** objects (resources or service providers), by invoking methods visible on the target's interface. The granularity of protection for access control in Ponder is thus an interface method. References to both subject and target objects are stored within domains maintained by a domain service. **Domains** provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers [25]. This facilitates policy specification for large-scale systems with millions of objects. Domains are similar to directories and have been implemented using an LDAP service.

2.1 Authorisation 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. Authorisation policies are implemented on the target host by an access control component.

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

Figure 1. Authorisation Policy Syntax

The syntax of an authorisation policy is shown in figure 1. Everything in **bold** is a language keyword in the figures presenting the syntax. Choices are enclosed in round brackets **( )** separated by |, optional elements are specified with square brackets **[ ]** and repetition is specified with braces **{ }**. Constraints are optional in all types of policies and can be specified to limit the applicability of policies based on time or values of the attributes of the objects to which the policy refers. Constraints are discussed in detail in section 4. Elements of a policy can be specified in any order. Note that the subject and target elements can optionally include the interface specification reference within the specified domain-scope-expression on which the policy applies. This can be used to check that the objects do support the specified operations or to locate the interface specification. The name of a policy can be specified as a path, thus identifying the domain into which the policy must be stored.

Example 1 Positive and negative authorisation policies

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

Members of the NetworkAdmin domain are authorised to load, remove, enable or disable objects of type PolicyT in the Nregion/switches domain. This indicates the use of an authorisation policy to control access to stored policies.

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

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

The above examples show direct declaration of policy instances using the keyword **inst**. The language provides reuse by supporting the definition of policy types to which any policy element can be passed as formal parameter. Multiple instances can then be created and tailored for the specific environment by passing actual parameters. Figure 2 shows the syntax for authorisation policy types and instantiations.

```
type ( auth+ | auth– ) policyType  “(” formalParameters “)”   “{”
   { authorisation-policy-parts } “}”

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

Figure 2. Authorisation Types and Instantiations

The authorisation policy *switchPolicyOps* (from example 1) can be specified as a type with the subject and target given as parameters as shown in example 2.

Example 2 Declaring instances from types

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

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

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

It can be argued that the specification of negative authorisation policies complicates the enforcement of authorisation in a system. However, there are reasons to support the provision for negative authorisation policies. Administrators often express high-level access control in terms of both positive and negative policies; retaining the natural way people express policies is important and provides greater flexibility. Negative authorisation policies can also be used to temporarily remove access rights from subjects if the need arises. In addition, many systems support negative access rights (e.g., Windows NT/2000). The existence of negative authorisation policies in a system may result in conflicts with positive authorisation policies. These conflicts are modality conflicts and can thus be always detected through static analysis of the policy specification. Although this adds the need to analyse policies for conflict detection, this kind of conflicts may however indicate potentially unforeseen problems with the specification. For a discussion on conflicts between policies see [14].

2.2 Information Filtering Policies

Filtering policies are needed to transform the information input or output parameters in an action. For example, a location service might only permit access to detailed location information, such as a person is in a specific room, to users within the department. External users can only determine whether a person is at work or not. Some databases support similar concepts of ‘views’ onto selective information within records – for example a payroll clerk is only permitted to read personnel records of employees below a particular grade. Positive authorisation policies may include filters to transform input or output parameters associated with their actions, based on attributes of the subject or target or on system parameters (e.g., time). In many cases it is not practical to provide different operations as a means of selecting the information. Although these are a form of authorisation policy they differ from the normal ones in that it is not possible for an external authorisation agent to make an access control decision based on whether or not an operation, specified at the interface to the target object, is permitted. Essentially the operation has to be performed and then a decision made on whether to allow results to be returned to the subject or whether the results need to be transformed. Filters can only be applied to positive authorisation actions.

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

Figure 3. Filters on Positive Authorisation Actions

Every action can be associated with a number of filter expressions (see figure 3). Each filter contains an optional condition under which the filter is applied. If the condition evaluates to true, then the transformations (the assignment statements in the body of the filter) are executed. The **in**/**out** keywords are used to indicate input and output parameters of the action on which the filter is specified; **result** is used to transform the return value of the action.

Example 3 Information filter policy

```
inst auth+ filter1  {
   subject  /Agroup + /Bgroup ;
   target   USAStaff – NYgroup ;
   action  VideoConf(BW, Priority)
           { in BW=2 ; in Priority=3 ; }     // default filter
           if (time.after("1900")) {in BW=3; 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 7:00pm then the video conference takes parameters: bandwidth = 3 Mb/s, priority = 1. Otherwise the first filter restricts the parameters to bandwidth = 2 Mb/s, priority = 3.

2.3 Delegation Policies

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

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

Figure 4. Delegation Policy Syntax

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

Figure 4 shows the syntax of a positive delegation policy. Note that the only required part is the **grantee**. The rest of the parts (subject, target, action) must be subsets of those in the associated authorisation policy; if not specified they default to those of that policy. A positive delegation policy can specify delegation constraints to limit the validity of the delegated access rights, as part of the **valid**-clause. Such constraints

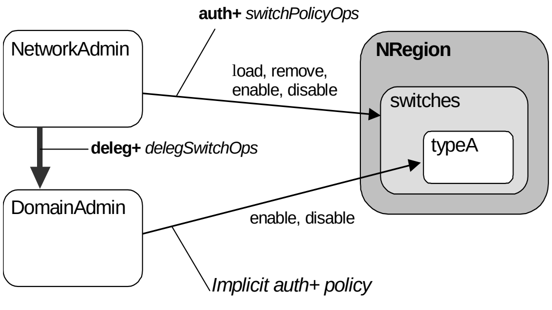

Figure 5. Delegation and Authorisation Policies

can be time restrictions (duration, validity period) to specify the duration or the period over which the delegation should be valid before it is revoked. Note that negative delegation policies do not contain delegation constraints.

Example 4 Delegation policy

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

The above delegation policy is associated with the switchPolicyOps auth+ policy from example 1. It states that the subject of that authorisation policy (NetworkAdmin), which is implicit in this policy, can delegate the enable and disable actions on policies from the domain /Nregion/switches/typeA to grantees in the domain /DomainAdmin. Note how the policy restricts the target to a subset of the switchPolicyOps policy target (See figure 5). The valid-clause, specifies that the delegation is only valid for 24 hours from the time of creation; after that it must be revoked.

A delegation policy specifies the authority to delegate, it does not control the actual delegation and revocation of access rights. 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.

2.4 Refrain Policies

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. Refrain policies have a similar syntax to negative authorisation policies, but are enforced by subjects rather than target access controllers. They are used for situations where negative authorisation policies are inappropriate because the targets are not trusted to enforce the policies (e.g., they may not wish to be protected from the subject). The syntax of refrain policies is the same as that of negative authorisation policies (figure 1).

Example 5 Refrain Policy

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

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

3 Obligation Policies

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

Obligation policies are event-triggered and define the activities subjects (human or automated manager components) must perform on objects in the target domain. Events can be simple, i.e. an internal timer event, or an external event notified by monitoring service components e.g. a temperature exceeding a threshold or a component failing. Composite events can be specified using event composition operators.

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

Figure 6. Obligation Policy Syntax

The syntax of obligation policies is shown in figure 6. Note the required event specification following the **on** keyword. The target element is optional as obligation actions may be internal to the subject, whereas authorisation actions always relate to a target object. If actions are to be invoked on a target, then they must be preceded by a prefix indicating the target set. Concurrency operators specifying whether actions should be executed sequentially or in parallel are used to separate the actions in an obligation policy. The optional **catch**-clause specifies an exception that is executed if the actions fail to execute for some reason.

Example 6 Obligation policy

```
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.

Types external to the policy specification can be specified assuming the corresponding specifications are accessible from a type repository.

Example 7 External types

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

The printFail obligation type accepts two parameters one of which is an external type called QueueMan. This is an interface specification of a printer queue manager object. The qman parameter is then used as a parameter in the call to putInQueue which is local to the printManager. The use of the || concurrency operator allows the actions to be performed in parallel.

4 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. These conditions are usually impossible or impractical to specify as part of each policy and therefore need to be specified as part of a group of policies. Thus, it is 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 (OCL) [22] is used to specify constraints in Ponder. OCL is simple to understand and use and it is declarative – each OCL expression is conceptually atomic and so the state of the objects in the system cannot change during evaluation.

4.1 Basic Policy Constraints

Basic policy constraints limit the applicability of a basic policy and are expressed in terms of a predicate, which must evaluate to true for the policy to apply. Policy constraints can be considered as conjunctions of basic constraints, which can be either time or state based 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 based on the object state as reflected in terms of attributes at the object interface.
- Action/event parameters – constraints can be based on event parameter values in obligations or action parameter values in authorisations or refrains.
- 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.

Example 8 Use of attribute and time constraints

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

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

```
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. The time-based constraint limits the policy to apply between 4:00pm and 6:00pm.

4.2 Meta-Policies

Meta-policies specify policies about the policies within a composite policy or some other scope, 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. A meta-policy is specified as a sequence of OCL expressions the last one of which must evaluate to true or false. The rest of the OCL expressions can be navigational expressions resulting in a collection. The **raises**-clause is followed by an action that is executed if the last OCL expression evaluates to true.

```
inst meta metaPolName raises exception [ “(” parameters  “)” ] “{”
   { OCL-expression}
   boolean-OCL-expression “}”
```

Figure 7. Meta-Policy Syntax

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

Self-Management: “*There should be no policy authorising a manager to retract policies for which he is the subject*”, from [12]. This happens within a single authorisation policy with overlapping subjects and targets. This can be specified in Ponder as follows:

Example 9 Self-management meta-policy

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

Example 10 Separation of duty

```
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 ;
}
```

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

The above policy implements a static separation of duties in that it prevents the same person being authorised to perform conflicting actions. Dynamic separation of duties is a slightly different, in that all members of a group are authorised to perform potentially conflicting actions but after performing one action they cannot perform a conflicting one. This is implemented as constraints relating to attributes of the subject and target object rather than as a meta-policy.

Example 11 Dynamic separation of duty

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

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

5 Composing Policy Specifications

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.

5.1 Groups

This is a packaging construct to group related policies together for the purposes of policy organisation and reusability and is a common concept in most programming languages. There are many different potential criteria for grouping policies together – they may reference the same targets, relate to the same department or apply to the same application. Figure 8 shows the syntax for a group instance. It can contain zero or more basic policies, nested groups and/or meta-policies in any order. A meta-policy specifies constraints on the policies within the scope of the group.

```
inst group groupName   "{"
   { basic-policy-definition }
   { group-definition }
   { meta-policy-definition } "}"
```

Figure 8. Group Syntax

Reusability can be achieved by specifying groups as types, parameterised with any policy element and then instantiating them multiple times. For instance, policies related to the login process can be grouped together since they must always be instantiated together (example 12).

Example 12 Group policy

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

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

   inst oblig loginFailure { … }  // see example 6
}
```

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

5.2 Roles

Roles provide a semantic grouping of policies with a common subject, generally pertaining to a position within an organisation such as department manager, project manager, analyst or ward-nurse. Specifying 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 [16]. A role can also specify the policies that apply to an automated component acting as a subject in the system.

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.

```
inst role roleName   "{"
   { basic-policy-definition }
   { group-definition }
   { meta-policy-definition }  "}" [ @ subject-domain ]
```

Figure 9. Role Syntax

A role (figure 9) can include any number of basic-policies, groups or meta-policies. The subject domain of the role can be optionally specified following the **@** sign. If it is not specified then a subject domain with the same name as the role is created by default.

Example 13 Role policy

```
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.

5.3 Type Specialisation and Role Hierarchies

Ponder allows specialisation of policy types, through the mechanism of inheritance. When a type extends another, it inherits all of its elements, add new elements and overrides elements with the same name.

```
type role roleTypeName “(” formalParameters “)”
     extends parentRoleType  “(” actualparameters “)”  “{”
   role-body  “}”
```

Figure 10. Inheritance Syntax

An example of the use of inheritance to extend a Role type is shown below. Similar syntax can be used to extend other types.

Example 14 Role inheritance

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

The MSServEngineer (MobileStation Service Engineer) role extends the ServiceEngineer role specified in example 13. It inherits the policies of the parent role and adds an obligation policy that updates the equipment’s record in the equipment identity registry (the target) when the mobile station signals a failure (the event).

Role and organisational hierarchies can be specified using specialisation. The role-hierarchy in figure 11 can be specified in Ponder by extending roles as shown in the following example.

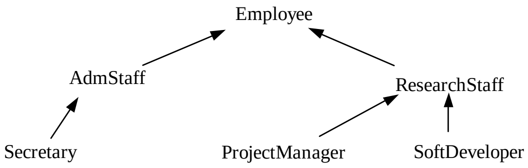

Figure 11. A Role Hierarchy

Example 15 A role hierarchy

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

5.4 Relationships

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.

Example 16 Relationship type

```
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.

5.5 Management Structures

Many large organisations are structured into units such as branch offices, departments, and hospital wards, which have a similar configuration of roles and policies. Ponder supports the notion of management structures to define a configuration in terms of instances of roles, relationships and nested management structures relating to organisational units. For example a management structure *type* would be used to define a branch in a bank or a department in a university and then *instantiated* for particular branches or departments. 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 12 shows a simple management structure for a software development company consisting of a project manager, software developers and a project contact secretary. Example 17 gives the definition of the structure.

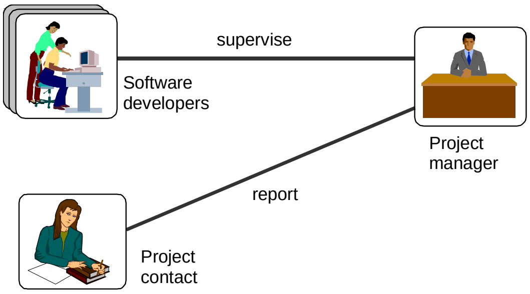

Figure 12. Simple Management Structure

Example 17 Software company management structure

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

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

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

6 Additional Features

6.1 Class Hierarchy

The class hierarchy of the language (figure 13), allows new policy classes that may be identified in the future to be defined as sub-classes of existing policy classes. The model also provides a convenient means of translating policies to structured representation languages such as XML. The XML representation can then be used for viewing policy information with standard browsers or as a means of exchanging policies between different managers or administrative domains.

Ponder is a strongly-typed language as every identifier and expression can be checked to be type consistent, and errors can be automatically detected if an operator or function is applied to the wrong type of data. Although most of the type checking occurs statically at compile time, Ponder also supports a form of dynamic type checking by accessing type definitions in a type repository. Statically strongly typed is an important characteristic for a policy language. Note that Ponder does not allow casting, and thus avoids the loopholes created because of casting in typed languages. Policies might be used in situations where their validity is important (e.g. security policies). Since program testing is usually not going to be an option in a policy-based system, static type checking is

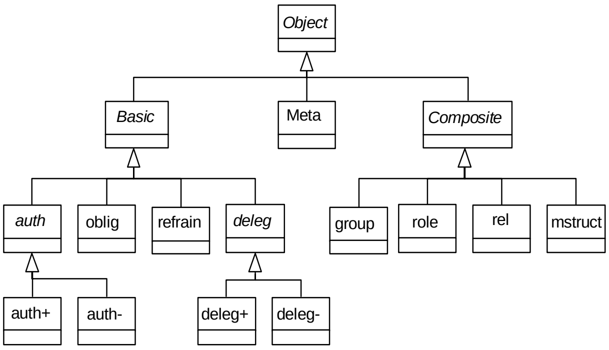

Figure 13. Ponder Object Meta Model

important to capture type inconsistencies that might cause failure of a policy at run-time.

6.2 Scripting

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 authorisation policies can be specified to control access to the scripts.

Scripts give the flexibility of including complex actions which cannot be expressed as single object method invocations and can contain conditional statements supported by the scripting language. For example a script could be defined to update software on all computers in a target domain as an atomic transaction which rolls back to the old version if any one of the updates fail.

If an interpreted language such as Java is used to program scripts, then the scripts could be updated using mobile code mechanisms to change the functionality of automated manager agents. Although this suffers from all the usual security vulnerabilities of mobile code [3], extending the functionality of manager agents is an operation on the agent's management interface and can be restricted by authorisation policies.

6.3 Imports

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

Example 18 Import statement

```
inst group /groups/groupA  {
   event e(userId) = 3*loginfail(userid) ;

   // other common specifications & basic-policies
}

inst group groupB {
   import /groups/groupA ;

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

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

6.4 Self-Management

The Ponder framework is self-managed in that policies and other constructs such as roles and relationships are implemented as objects stored within domains. Ponder authorisation policies can therefore be used to specify who is permitted to add, delete, edit policies or any of the other entities. Furthermore, Ponder obligation policies can be used to specify what actions must be performed on policy objects when certain events occur. For example, obligation policies can be specified to enable new policies or disable existing ones in order to adapt to new circumstances such as failures, emergency conditions, etc.

7 Related Work

Most of the other work on policy language specification relates to security. None includes the range of policies covered in Ponder and most lack the flexibility and extensibility features of Ponder.

Formal logic-based approaches are generally not intuitive and do not easily map onto implementation mechanisms. They assume a strong mathematical background, which can make them difficult to use and understand. The ASL [12], is an example of a formal logic 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.

Ortalo [21] describes a language to express security policies in information systems based on the logic of permissions and obligations, a type of modal logic called deontic logic. Standard deontic logic centres on impersonal statements instead of personal; we see the specification of policies as a relationship between explicitly stated subjects and targets instead. In his approach he accepts the axiom Pp = ¬O¬p ("permitted p is equivalent to not p being not obliged") as a suitable definition of permission. This axiom is not suitable for the modelling of obligation and authorisation policies; the two need to be separated. Miller [19] discusses several paradoxes that exist in deontic logic. Since [21] contains only syntactical extensions to deontic logic, it also suffers from the same problems.

LaSCO [10] 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.

The Policy Description Language (PDL) is an event-based language originating at the network computing research department of Bell-Labs [28][13]. Policies in PDL are similar to Ponder obligation policies. They use the event-condition-action rule paradigm of active databases to define a policy as a function that maps a series of events into a set of actions. The language has clearly defined semantics and an architecture has been specified for enforcing PDL policies. The language can be described as a real-time specialised production rule system to define policies. Events can be composite events similar to those of Ponder obligation policies. PDL does not support access control policies, nor does it support the composition of policy rules into roles, or other grouping structures.

8 Conclusion and Further Work

In this paper we have presented Ponder, a language for specifying policies for management and security of distributed systems. Ponder includes authorisation, filter, refrain and delegation policies for specifying access control and obligation policies to specify management actions. Ponder thus provides a uniform means of specifying policy relating to a wide range of management applications – network, storage, systems, application and service management. In addition, it supports a common means of specifying enterprise-wide security policy that can then be translated onto various security implementation mechanisms. We are currently implementing back-ends to the Ponder compiler for Firewall rules, Windows security templates and Java security policy.

The Ponder composite policies (groups, roles, relationships and management structures) allow structured, reusable specifications, which cater for complex, large-scale organisations. Ponder's object-oriented features allow user-defined types of policies to be specified and then instantiated multiple times with different parameters. This provides for flexibility and extensibility while maintaining a structured specification that can be, in large part, checked at compile time. Meta-policies in Ponder provide a very powerful tool in specifying application specific constraints on sets of policies. Ponder is declarative which aids in the analysis of policies [14].

The language specification leaves room for future additions in many areas. Relationships need to be extended with interaction protocols to specify the interaction between roles. We are also investigating sub-types of meta policies to cover concurrency constraints and user-role assignment constraints. Giving the language formal semantics will also be part of future work. A policy specification toolkit is under development for defining, compiling and analysing policies. The design and implementation of a generic runtime object-model for enforcement of Ponder policies on any object-based platform is also under development.

Acknowledgement

We gratefully acknowledge their support of EPSRC for research grants GR/L96103 (SecPol), GR/M86109 (Ponds) and GR/L76709 (Slurp).

References

For additional references see http://www-dse.doc.ic.ac.uk/policies.

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. Chess, D.M., *Security Issues in Mobile Code Systems*, in Mobile Agents and Security, G. Vigna, Editor. 1998, Springer. p. 256.
4. 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
5. 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 2.2. Research Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, London, 3 April, 2000.
6. Distributed Management Task Force, Inc. (DMTF), *Common Information Model (CIM) Specification*, version 2.2, available from http://www.dmtf.org/spec/cims.html, June 14, 1999.
7. Goh, G. *Policy Management Requirements*, System Management Department, HP Laboratories Bristol, April, 1998.
8. Hegering, H.-G., S. Abeck, and B. Neumair, *Integrated Management of Network Systems: Concepts, Architectures and Their Operational Application*, 1999: Morgan Kaufmann Publishers.
9. Hewlett-Packard Company, *A Primer on Policy-based Network Management*, OpenView Network Management Division, Hewlett-Packard Company, September 14, 1999.
10. 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.
11. Internet Engineering Task Force, *Policy Working Group* http://www.ietf.org/html.charters/policy-charter.html
12. 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
13. Lobo, J., R. Bhatia, and S. Naqvi. *A Policy Description Language*. In Proc. of AAAI, July 1999. Orlando, Florida, USA
14. Lupu, E.C., and M. Sloman. *Conflicts in Policy-Based Distributed Systems Management*. IEEE Trans. on Software Engineering, 25(6): 852-869 Nov.1999.
15. Lupu, E.C. *A Role-Based Framework for Distributed Systems Management*. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K.
16. 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.

17. Mahon, H. *Requirements for a Policy Management System*. IETF Internet draft work in progress, Available from http://www.ietf.org, 22 October 1999.
18. Marriott, D.A. *Policy Service for Distributed Systems*. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K.
19. Miller, J., HELP! *How to specify policies*?, Unpublished paper, available electronically from http://enterprise.shl.com/policy/help.pdf
20. Moore, B., J. Strassner, and E. Ellesson, *Policy Core Information Model VI*, IETF Internet draft, Available from http://www.ietf.org, May 2000.
21. 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.
22. Rational Software Corporation, *Object Constraint Language Specification,* Version 1.1, Available at http://www.rational.com/uml/, September 1997.
23. 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.
24. 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.
25. 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.
26. Sloman, M.S., *Policy Driven Management for Distributed Systems*. Journal of Network and Systems Management, 1994b. 2(4): p. 333-360.
27. Sun Microsystems, Inc., *Java Management Extensions Instrumentation and Agent Specification,* v1.0, December 1999.
28. Virmani A., 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.
29. Weis, R. Policy Definition and Classification: Aspects, Criteria and Examples. In Proceedings of IFIP/IEEE International Workshop on Distributed Systems: Operations & Management. 1994a. Toulouse, France.