

Ponder: Realising Enterprise Viewpoint Concepts

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

Abstract

This paper introduces the Ponder language for specifying distributed object enterprise concepts. Ponder, is a declarative language, which permits the specification of policies in terms of obligations, permissions and prohibitions and provides the means for defining roles, relationships and their configurations in nested communities. Ponder provides a concrete representation of most of the concepts of the Enterprise Viewpoint. The design of the language incorporates lessons drawn from several years of research on policy for security and distributed systems management as well as policy conflict analysis. The various language constructs are presented through a scenario for the operation, administration and maintenance of a mobile telecommunication network.

1. Introduction

The ODP Reference Model Enterprise Viewpoint [1] defines concepts for specifying an abstraction of a system within a defined environment in terms of the system's purpose, its scope and the policies that apply to the system from its environment as well as those defined within the system. The current specification defines terms such as policy, role, domain, and community but there is no associated language by which a system can be clearly specified, analysed or implemented.

Ponder is a language for specifying Security and Management policy for distributed systems [2], which has evolved over a number of years from the Policy Based Management work at Imperial College [3],[4],[5]. The main motivation for this language is to specify policies that are interpreted by components in the system. The policies can then be easily modified in order to change the behaviour of the system without re-implementation of the components. We define *policy* as a *rule governing the choices in behaviour of the system*. Policies are derived from the high level goals (c.f. purpose) of an enterprise or from contracts defining Service Level Agreements. Although Ponder was not originally designed as an Enterprise Viewpoint Language, it does provide a concrete representation to "realise" most of the tangible concepts being defined for the Enterprise Viewpoint. However, there is not a one-to-one mapping between Enterprise

concepts and Ponder constructs, nor do we attempt to model some of the more abstract ones such as "purpose" and "objective". Ponder also defines additional concepts such as delegation and relationships.

Ponder is a declarative, object-oriented language for specifying different types of policies, for grouping policies into roles and relationships, and then defining configurations of roles and relationships as management structures. Ponder can be used to specify security policies with role-based access control, as well as general-purpose management policies. Although it is a typed language, Ponder offers a high degree of flexibility by supporting parameterisation of any parts of a specification. We have assumed a class-based object-oriented view of the underlying distributed system where interaction occurs through remote object invocations and asynchronous event notifications. Ponder's types and instances are implemented as objects in the system and therefore policies may also apply to these objects.

The next section outlines a simplified GSM mobile telecommunications enterprise that is used as a scenario to illustrate the concepts presented in this paper. Section 3 covers management structures that define the organisational units (communities) as configurations of roles and section 4 explains how to specify the main Ponder policies relating to authorisation and obligation. In sections 5 and 6 we elaborate on roles, relationships and additional types of policies. Related work is covered in section 7 followed by conclusions.

2. Scenario

In this scenario, we consider the enterprise components needed to distribute the management responsibilities of the cellular network of an imaginary operator RCom. We will limit our specifications to a simplified subset of the functions needed for the management of Base Station Sub-systems (BSS) and subscriber data. We assume that RCom's network is divided into regions with several branches.

Figure 1 is a schematic representation of the network components in our GSM mobile network scenario. We have assumed an object interface to network components and abstract events that do not correspond to the GSM specifications [7]. GSM uses TMN concepts for management and SS7 for network signalling.

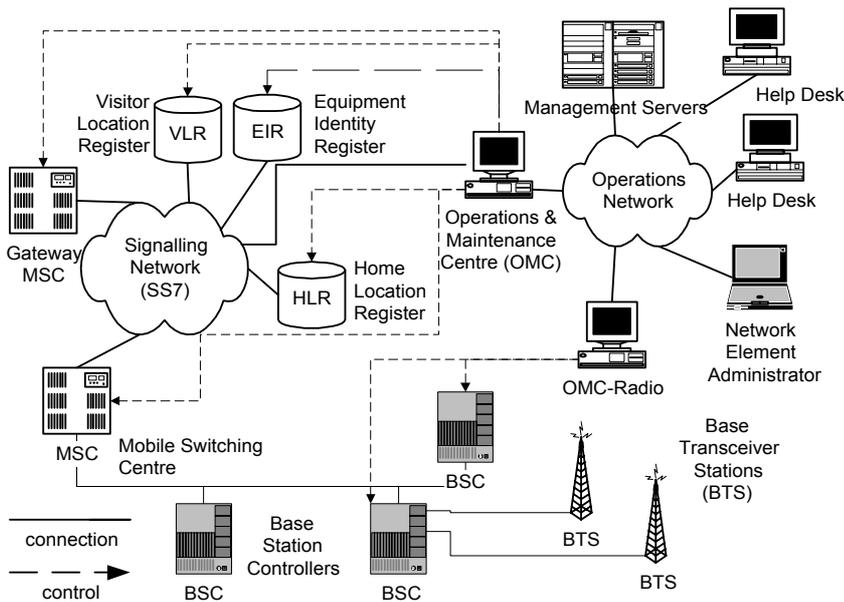


Figure 1. GSM system overview

In large-scale systems, there is a need to group objects in order to apply a common policy, to partition management responsibility and authority, to reflect geographical or organisational boundaries or for the convenience of human managers. For example, RCom's national network could be divided into 4 or 5 regions, corresponding to different local carriers that maintain their own Home Location Registers (HLRs). Each region could be further sub-divided into a number of branches, each corresponding to the coverage area of a Mobile Switching Service Centre (MSC) that covers approximately 800,000 inhabitants according to [8]. Regions and branches correspond to ODP communities.

Domains [6] are used to group references to objects and sub-domains in a structure similar to that of a file system directory. However, domains may hold references to any type of objects including a person and objects can be members of multiple domains i.e., domains can overlap.

Domains have been implemented as directories in an extended LDAP Service. Ponder domains are merely a grouping construct, similar to the ODP group, so differ from ODP domains which have a single controller role that controls roles with respect to a specific aspect of their behaviour. In most of our case studies, we have found it more practical to separate manager agents into different domains from the objects they manage, as this allows different policies to be specified for

different managers regarding a common set of objects. A Ponder Domain could be used to model the ODP domain in that it could hold a controlling object plus controlled objects if there is a need to do so.

Figure 2 shows the partial domain structure within the Wales region with one branch (Cardiff) expanded. Sub-domains can be created in the Cardiff branch to group roles, policy objects and network elements. Different domains have been created for the operations network (opn) and the telecommunications network (tn) containing the Mobile Switching Centre (MSC), Visitors Location Register (VLR) and the Base Station Controllers (BSC). However, regional policies, implemented by managers in a regional operations centre, may also be defined for the uniform configuration, or logging strategy of all BSCs in the region. Hence, the BSC sub-domain of the cardiff branch may also be included in a sub-domain (cardiff) of the regional domain of Base Station Sub-systems (BSS). The Cardiff tn domain is then overlapping with the regional BSS domain. Help-desk staff in a branch may need to access the HLR and the Equipment Identity Register (EIR) in order to update subscriber information. It is therefore useful to also include the regional HLR and EIR in the local scope of the tn domain of a branch in order to simplify policy specification. The HLR and EIR can then be referred to as local objects at the branch level.

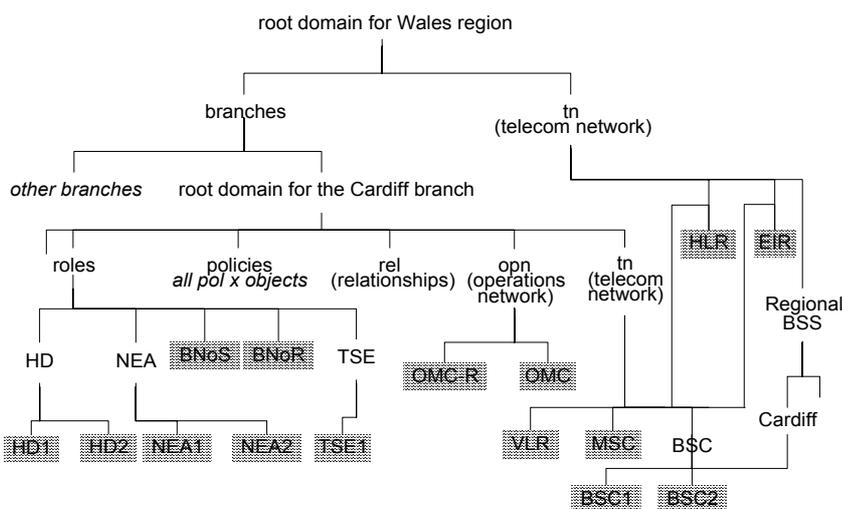


Figure 2 Regional domain structure (shaded boxes represent non-domain objects)

Path names can be used to identify domains and objects in the

domain structure e.g., `/wales/branches/cardiff/tn/VLR` refers to the VLR of the cardiff branch in region wales of the national network (root domain – ‘/’). Policies are specified in terms of set expressions formed from domains, so objects can be added and removed from domains without having to change the policy specifications. Policies applying to a domain normally propagate to members of sub-domains so a policy applying to `/wales/branches/cardiff/tn` will also apply to all BSCs managed within the branch.

3. Management Structures

A branch in the operation of a GSM network is an organisational unit which ensures the management and administration of the elements within the coverage area of an MSC and provides the customer service support within that area. In Ponder the representation of an organisational unit is called a management structure (mstruct) and groups the specifications of the various roles and their inter-relationships. Hence, a management structure realises the Enterprise Viewpoint concept of a community which defines a configuration of roles formed to meet an objective; in this case to manage part of the GSM network. Within a branch we have considered the following types of roles:

- **Help-Desk Staff (HD)** provide the interface between customers and RCom (not elaborated in this scenario).
- **Telephone Service Engineers (TSE)** investigate faults occurring on the radio interface between mobile stations and base transceiver stations and determine whether a base network operator should be alerted to deal with the fault.
- **Base Network Operators – Switches (BNoS)** are responsible for managing the Mobile Switching service Centre (MSC) and Visitors Location Register (VLR) situated in a branch and cooperate with the BNoRs for Radio related functions.
- **Base Network Operators – Radio (BNoR)** are responsible for Base Transceiver Systems (BTS) and cooperate with the TSEs.
- **Network Element Administrators (NEA)** perform all on-site management tasks requested by BNoS and BNoR. They may initiate tests and configuration procedures defined in the Operations Management Centre Workstations (OMC).

The configuration of roles and their inter-relationships for the branch is shown in Figure 3 where inter-role relationships have been represented by arrows. Unlike the Enterprise Viewpoint, Ponder has an explicit construct for defining relationships between roles which group the obligations and permissions of the roles towards each other, e.g., right to assign a task to a role, and policies with regards to the use of shared resources.

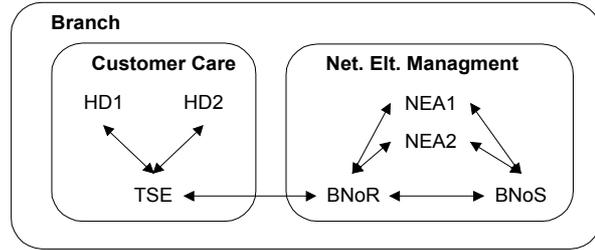


Figure 3 Branch Management Structure

The branch management structure is itself sub-divided in two management structures – one responsible for the customer care and the other responsible for the management of network elements. The customer care structure groups the help-desk roles and the telephone service engineer role that is responsible for investigating failures. Network element administrators and base network operators are grouped in the network element management structure. This configuration can be specified in Ponder as follows:

```

type mstruct branch (domain br, nw) {
    import /type/custcare; /type/netelements;

    inst mstruct cc = custcare (br, nw);
                ne = netelements (br, nw);

    type rel radiofail (role eng, radio_op) { ... }

    inst rel f = radiofail (cc.tse, ne.bnor) ;
}

domain c = ../wales/branches/cardiff/;
inst mstruct cardiff = branch (c, c/tn/);

```

The branch type, declared with the keyword mstruct takes two parameters, the domain br in which it resides and the domain nw in which the various network elements reside. Like any other composite type in Ponder, an mstruct can contain any number of type declarations, and any number of instances, declared as two blocks with the keywords type and inst above. Ponder does not impose limits on the number of these blocks or their ordering. A type specification may also contain constraints (not shown here, but see section 6.4) and may import type declarations from a type repository. In this example, type specifications for the custcare and netelements mstructs have been imported and one instance of each has been created within the branch. The branch mstruct type also contains the declaration of a relationship type and its instantiation. The relationship instance f specifies the policies governing the behaviour of the TSEs and of the BNoRs towards each other. An mstruct may contain roles, policies, groups of policies, relationships and other mstructs. Finally, an instance of the branch is created for the Cardiff area. Similarly we specify below the mstruct types for the custcare and netelements structures from Figure 3.

```

type mstruct custcare (domain br, nw) {
  import /type/helpdesk; /type/engineer; /type/complaint;

  domain r = br/roles/; h = br/roles/hd/; s = br/rel/;

  inst role h/hd1 = helpdesk (...);
             h/hd2 = helpdesk (...);
             r/tse = engineer (nw/vlr, nw/eir);

  inst rel s/hd1_tse = complaint (hd1, tse);
           s/hd2_tse = complaint (hd2, tse);
}

type mstruct netelements (domain br, nw) {
  import administrator; switch_op; radio_op; repair; base;

  domain r = br/roles/; a = r/nea/; s = br/rel/;

  inst role a/nea1 = administrator (...);
             a/nea2 = administrator (...);
             r/bnos = switch_op (nw/bsc/, nw/msc);
             r/bnor = radio_op (nw/bsc/);

  inst rel s/bnos_nea1 = switch_repair (bnos, nea1);
           s/bnos_nea2 = switch_repair (bnos, nea2);
           s/bnor_nea1 = radio_repair (bnor, nea1);
           s/bnor_nea2 = radio_repair (bnor, nea2);
           s/bnos_bnor = base (bnos, bnor);
}

```

Federation between existing management structures can be accomplished by importing mstruct instances from a domain and creating new policies or relationships with respect to existing roles.

4. Authorisation, Obligation, Refrain and Policy hierarchies

The main types of basic policies in Ponder are authorisations, obligations and refrain. Additional types of policies such as delegations and filtered authorisations are presented in Section 6. Authorisations can be either positive (permissions) or negative (prohibitions) and define the actions that a set of subjects (objects) are permitted or prohibited from performing on a set of target objects under given conditions. Subjects are any objects initiating an action such as a remote method invocation within the system. Targets are any objects on which actions are performed. Conditions are specified by constraints expressed in a subset of the Object Constraint Language (OCL) [11]. Obligations specify which actions subjects must perform on a given set of target objects when notified of the occurrence of a particular event and refrain policies specify which actions subjects must refrain from performing.

Note that there are some differences between Ponder policies and policies in the Enterprise Viewpoint specification. An Enterprise Viewpoint authorisation “is a prescription that a particular behaviour must not be prevented” and a permission defines a “prescription that a particular behaviour is allowed to occur. A permission is equivalent to there being no obligation for the behaviour

not to occur.” The latter assumes a semantic similar to that considered in Standard Deontic Logic [12] but the difference between authorisation and permission does not seem to be meaningful in a computing system. Ponder authorisations specify a prescription that the access control system associated with the target objects should permit the specified actions. Ponder authorisations do *not* imply obligations. While deontic logic may prove useful for reasoning about policies, it does not provide a clear mapping onto implementation mechanisms. Enterprise Viewpoint prohibitions can be realised by negative Ponder authorisations and Ponder refrain policies. A negative authorisation is a prescription on the behaviour of the access control system which protects the *target* objects while a refrain policy is a prescription on the behaviour of the *subject* of the policy (see Section 4.3 below).

4.1. Authorisation

Authorisation policies define what actions a set of objects in a subject domain are permitted or prohibited to perform on a set of objects in a target domain. In the context of our case study, the following examples can be defined.

```

domain network = ../tn/; staff = ...; accounting = ...;

```

```

/* Help-desk staff are allowed to add, trace and lock
subscribers in the HLR during office hours */

```

```

inst auth+ helpdesk_access {

```

```

  subject staff/helpdesk/;
  target network/hlr;
  action add, trace, lock;
  when time.between (“0800”, “1700”);
}

```

```

/* Help-desk staff are not permitted to initiate traces on
foreign subscribers */

```

```

inst auth- helpdesk_deny {

```

```

  subject staff/helpdesk/;
  target network/vlr;
  action trace_foreign;
}

```

Authorisation policies specify subject and target domains, (optionally, the type of objects within the domains) and the actions that are permitted or prohibited. **Constraints** may be used to restrict applicability to a particular time interval as in the above examples (time is a library object), according to the state of the subjects, targets or other system objects, or according to the action parameters. For example, a variation of the policy `helpdesk_access` above can be written as:

```

inst auth+ alt_helpdesk_access {

```

```

  subject staff/helpdesk/;
  target network/hlr;
  action lock (sid);
  when accounting/bill_payments.unpaid_bills(sid); }

```

In this policy the parameter of the action lock is bound to the local variable sid (subscriber id) and help-desk staff are permitted to lock a subscriber, thereby preventing him from using the phone when the subscriber has failed to pay the bills. The presence of unpaid bills is checked for by calling the `unpaid_bills` function of the `bill_payments` object in the accounting domain. The domain structure presented in Section 2 is used to maintain references to all accessible system objects. Hence, policies may refer to these objects, for example in a constraint, by using their path-name within the domain structure. Policy constraints are specified using a subset of the OCL operators.

All policies can be defined as a policy type which is used to create policy instances with instantiation parameters to tailor the instance to its particular environment, i.e., to the particular objects to which it refers. For example, there will be similar policies in all the branches of RCom, but the policies will apply to particular people and equipment in each branch. Thus, the following policy type can be written for the policy `helpdesk_access` and instantiated with different subjects and targets. Note however that Ponder allows any elements of a policy to be parameterised including constraints and actions. This provides the flexibility needed to create multiple instances of a policy to perform actions in different circumstances such as different time intervals or different states of the subjects.

```
type auth+ a1_type (subject nea, target <bsc_type> bsc) {
  action shutdown, reset;
  when time.between("0800", "1700"); }
```

```
inst auth+ helpdesk_access = a1_type (roles/nea/, branch/tn) ;
```

Note: when the keywords **subject** or **target** are used as formal parameters, they act as a short form for a **set** parameter plus a subject/target declaration, e.g. the above is a short form for the following policy type declaration:

```
type auth+ a1_type (set nea, set <bsc_type> bsc) {
  subject nea;
  target bsc;
  action shutdown, reset;
  when time.between("0800", "1700"); }
```

In this example the actions are restricted to objects of type `bsc_type` within the `bsc` set passed as parameter.

4.2. Obligation

Obligation policies specify which actions subjects must perform on a domain of target objects in response to an event, with constraints similar to those for authorisation policies. An action could be a remote invocation on a target object, a local operation within the subject or a local script written in any suitable language. In the following example, the triggering event is specified with the keyword `on` and is followed by the actions that must be performed. Obligation policies can specify a sequence of actions using the `'->'` operator or parallel actions using the `'||'` operator.

```
/* On a TX circuit failure, replace the circuit with a backup, and in parallel reconfigure the BTS with logging the failure. */
```

```
type oblig fail_reconfigure (subject s, set b) {
  on failure (cir, bts, bsc);
  target f = b ^ {bsc};
  do f.disable(bts, cir) ->
    (f.enable(bts, "backup") || s.log (cir, bts, bsc)); }

inst oblig p = fail_reconfigure (.../roles/bnor/, .../tn/bsc) ;
```

This policy specifies that on the occurrence of a transmission failure event, BNoRs must disable the failed circuit and enable the backup circuit on the base transceiver system (BTS). Note, that the attributes of the event are bound to the local variables `cir`, `bts`, and `bsc` thereby identifying the failed transmission circuit (`cir`), the BTS to which the circuit belongs (`bts`), and the base station controller (`bsc`) to which the BTS is attached. The identifier for the failed circuit and for the BTS are used as parameters of the actions to be performed while the identifier of the managing BSC determines the target object on which the actions must be performed. The log operation is performed as a local operation within the subject. Ponder allows the use of variables defined later on in the same scope, (e.g. `bsc`) and does not impose an order on declarations within a policy (e.g. `subject`, `on`, `do`).

4.3. Refrain

Refrain Policies define the actions that subjects must refrain from performing on target objects, and like obligations, they are implemented by the subject. Refrain policies are used for situations where negative authorisation policies are inappropriate, as the targets do not wish to be protected from the subject. A refrain can also be used when the subject is permitted to perform the action but is asked to refrain from doing so when particular constraints apply. Refrain policies are syntactically the same as negative authorisation policies. For example, staff should refrain from calling customers on Sunday, although they may be authorised to do so:

```
type refrain polite_behaviour (subject staff, target subscribers) {
  action call;
  when time.dayOfWeek( ) = "Sunday" ;
}
```

4.4. Policy Hierarchies

Policies applying to an enterprise system, such as those described in the Enterprise Viewpoint, may be abstract and need to be refined into concrete realisations in order to be implemented. The progressive refinement of an abstract policy or goal leads to a refinement hierarchy where lower levels realise the more abstract representation above. Ponder aims to specify the policies situated in the concrete levels of this hierarchy and that can be directly implemented by an automated process or a human

manager, possibly with the aid of an automated system. This explains some of the divergences from the Enterprise Viewpoint specification. For example, Ponder restricts obligation policies to those actions that must be performed in response to events. It requires the triggering event to be specified as part of the policy and does not provide the means of specifying non-functional descriptions such as “it is obligated that X occurs” which are encountered in deontic specifications.

We are currently working on tool support for refining high-level goals into implementable policies following the Requirements Engineering [13] approach for refining goals into specifications. We also hope to make use of their approach to consistency analysis to see whether policies correctly satisfy goals and to include conflict analysis, based on our previous work [10].

5. Roles and Relationships

5.1. Specification

In Ponder, *roles* are groups of policies governing the behaviour of the same subject while *relationships* group the policies defining the rights and duties of roles towards each other. Policies that are part of a role or of a relationship apply to the entity assigned to the role. It is therefore possible to dynamically assign or remove an entity to a role without changing the policies of that role. As shown in Section 3 roles and relationships are themselves grouped in management structures. Policies grouped in roles have the same subject that is therefore not specified as part of each individual policy. The subject may either be explicitly determined upon instantiation of a role type or implicitly created upon instantiation.

A Ponder role corresponds to the rights and duties relating to a position within an organisation, for example a telephone service engineer, radio network operator, a nurse or a doctor in a hospital. A role could also define the set of policies for which an automated device is the subject e.g. a BSC reconfiguration agent. An ODP role is an “identifier for a behaviour, which may appear as a parameter in a template for a composite object, and which is associated with one of the component objects of the composite object.” Ponder roles cannot contain other roles or be assigned to other roles. Furthermore, while an Enterprise community containing roles may be assigned to a role, management structures cannot be assigned to roles in Ponder.

Details of how Ponder roles can be implemented in terms of position domains and how users or automated agents are assigned to roles have been described elsewhere [10]. This caters for the Enterprise requirement of assigning different objects to roles at different times.

The rights and duties for telephone service engineers in the operations of a GSM network can be written as follows:

```

type role engineer (vlr, eir){
    /* On a handset failure the engineer must update the
       record in the Equipment Identity Register (eir) */

    inst oblig update { target eir;

        on    hs_failure (x);
        do    eir.update_record (x) ; }

    /* On use of grey-listed MS initiate a trace in the visitors
       location register (vlr) */

    inst oblig trace { target vlr;

        on    grey_list_use (x) ;
        do    vlr.trace_foreign (x) ; }

    /* other policies */
}

```

A role type may include multiple policy and constraint instances as well as further type definitions and can be extended by specialisation. For example, a common role type definition can be specified for base network operators grouping their common rights and duties:

```

type role op (set t) {

    /* Restart failed equipment in target domain */

    inst oblig restart { target f = t ^ {id};

        on    failure (id) ;
        do    f.restart () -> f.run_self_test () ; }

    /* other policies */
}

```

This specification can be extended to define the role types for base network operators for switches (*switch_op*) and for radio equipment (*radio_op*) with the specific rights and duties of each.

```

type role switch_op (set <bsc_type> b, <msc_type> m)
extends op (b) {

    /* On A_interface circuit failure the circuit
       must be blocked and reset. */

    inst oblig reset {
        on    A_failure (x, bsc) ;
        target f = b ^ {bsc} ;
        do    f.reset (x) ; }

    inst oblig block { target m ;
        on    A_failure (x, bsc) ;
        do    m.block_circuit (x, bsc) ; }

    inst auth+ bsc_auth { target b ;
        action reset, disable ; }

    inst auth+ msc_auth { target m ;
        action block_circuit, unblock_circuit ; }
}

```

```

type role radio_op (set <bsc_type> b) extends op (b) {
  inst oblig reset { ... }
  inst auth+ auth_station_controller { target b ;
    action force_handoff, bar_cell, shutdown ; }

  /* On cell-overload, force a hand-over of all
     connected mobiles */
  inst oblig clear_cell {
    on cell_overload (bts, bsc) ;
    target f = b ^ {bsc};
    do f.force_handoff (bts) ; }

  inst oblig fail_reconfigure {
    on failure (cir, bts, bsc) ;
    target f = b ^ {bsc};
    do f.disable (bts, cir) ->
        f.enable (bts, "backup") ; }

  /* On 3 occurrences of a failure event report,
     increase BTS transmission */
  inst oblig increase {
    on 3*radio_link_fail (bts, bsc) ;
    target f = b ^ {bsc};
    do f.increase_power(1) ; }
}

```

Specialisation of a role type is achieved by the keyword `extends` followed by the name of the role type which is extended and the parameter values for that role.

Note that policies may be specified within a role as well as outside the role, for example as part of a management structure or simply applying to a domain of objects.

5.2. Role Relationships

Relationships group the rights and duties, i.e., the permissions, prohibitions and obligations of the related roles towards each other or with regards to the use of shared resources. A management structure may include several roles with different relationships between them. For example, in the network element management structure (Figure 3) there are several relationship instances between base network operators and network element administrators and a relationship of a different type between the two base network operators. In addition to policies, relationships may also define the interaction protocols that govern communications between the roles in terms of message exchanges [5], however this has not yet been included in the current version of Ponder. A relationship, or a relationship type is defined in Ponder by specifying the roles that participate in the relationship and the policies associated with these role. Policies that define the rights and duties of the roles towards each other have the relevant roles as subjects and targets. For example, in the case of our GSM operations scenario, base network

operators collaborate with network element administrators and are authorised to assign them tasks. This `assign_rel` relationship type is specified below, and further specialised for the cases of a BNoS (`switch_repair`) and a BNoR (`radio_repair`) using the same inheritance mechanism as for roles.

```

type rel assign_rel (administrator admin, op operator) {
  inst auth+ assign_pol {subject operator ; target admin ;
    action assign; }
}

type rel switch_repair (administrator admin, switch_op operator)
extends assign_rel (admin, operator) {
  ...
}

type rel radio_repair (administrator admin, radio_op operator)
extends assign_rel (admin, operator) {
  ...
}

```

This `assign_rel` relationship comprises the `assign_pol` authorisation policy instance which gives base network operators the right to assign tasks to network element administrators. The subject and target of this policy are the participant roles of the relationship as defined by the role statement. Note that these roles are typed. Hence, an instance of this relationship can be created only between roles of types `administrator` and `op` respectively. This relationship type is further specialised and these extended versions require participant roles to be of types `switch_op` and `radio_op` which are sub-types of `op`. Hence, strong typing principles from classical object-oriented programming have also been applied in Ponder in order to maximise the compile time verification and ensure rigorous specifications.

6. Group policies and other policy types

6.1. Groups

Groups are constructs which permit policy makers to group related policies together in order to instantiate all the policies at the same time, to share common definitions between the policies or to apply constraints to the set of policies, e.g.:

```

group reset_group (set s) {
  set b = .../cardiff/tn/bsc/;
  extern msc_type m = .../cardiff/tn/msc ;
  //create a reference to an object

  constraint active(x,bscid) = bscid.isActive(x)
  and m.isActive(x);
}

```

```

inst oblig reset { subject s ; target s1 ;
  on A_failure(x, bscid);
  set s1 = b ^ {bscid};
  do s1.reset(x)
  when active (x, bscid)
}

inst oblig block { subject s ; target m ;
  on A_failure (x, bscid);
  do m.block_circuit (x) ;
  when active (x, bscid)
}

inst auth+ bsc_auth { subject s ; target b ;
  action reset, disable; }

inst auth+ msc_auth { subject s ; target m ;
  action block_circuit, unblock_circuit ; }
}

```

6.2. Delegation

Ponder permits the specification of delegation policies which specify the subset of the permissions grantors possess which they are permitted to delegate to others. Therefore, a delegation policy in Ponder must explicitly refer to the authorisation policy that gives the grantors the permissions, a subset of which can be delegated. For example, base network operators may temporarily delegate network element administrators the right to bar a GSM cell.

/ Branch network operators – radio are permitted to force handovers, shutdown or barr a cell controlled by a given BSC */*

```

domain r = .../cardiff/roles/ ; n = ...cardiff/tn/ ;

inst auth+ auth_bsc {

  subject r/bnor ;
  target n/bsc/ ;
  action force_handoff, bar_cell, shutdown ; }

```

/ Branch network operators – radio are permitted to delegate the barrCell action to network element administrators on the cell controlled by the BSC1 */*

```

inst deleg+ delegate_bar (auth_bsc) {

  target n/bsc/bsc1 ;
  grantee r/nea/ ;
  action bar_cell ; }

```

This second policy authorises the BNoR to delegate the bar_cell action to objects specified in the grantee scope. The delegated action can be performed on objects specified in the target of the delegation policy, which must be a subset of the target scope of the authorisation policy (here auth_bsc) that grants the subject the permissions the delegate_bar policy allows him to delegate. Delegation policies can be positive or negative, a negative policy specifying that the delegation is explicitly prohibited. Cascaded delegation is catered for by allowing delegation policies to serve as basis for further delegating the permissions.

6.3. Filters

Defining permissions and prohibitions in terms of the actions that subjects are authorised or forbidden to perform is not sufficient in some cases. Consider for example that both help-desk staff and telephone service engineers may be permitted to query a location service in order to find the location of a particular subscriber. While telephone service engineers investigate faults and must know exactly to which cell the mobile station is connected, help-desk staff need only know if the subscriber is currently connected to the home network or is roaming. Filtered authorisations are an extension to Ponder's authorisation policies, which can be used in order to specify transformations to the input parameters or result of an action which subjects are permitted to perform. The location service example can be specified in Ponder as follows:

```

type auth+ filter_location (subject s, target t) {

  action lookup ( ) if subset (s, .../cardiff/roles/hd/) {
    result = select ("Network", result)
  }
}

```

Queries from all other staff are left unmodified while queries from members of the HD domain (help-desk staff) are filtered by applying the select function to the result.

6.4. Constraints

Ponder uses a subset of Object Constraint Language (OCL) [11] for specifying constraints which limit the applicability of a policy, for example to a particular time interval or according to the state of the system. These are specified as part of the individual obligation, authorisation, refrain and delegation policies. Semantic constraints which restrict the set of permissible policies (also referred to as Meta-Policies) can be specified as part of any composite policy including groups, roles, relationships and management structures [10]. These are needed in order to determine application specific conflicts such as separation of duties or conflicts for resources. For example, a conflict of separation of duties can be specified in Ponder as follows:

```

type meta dutyconflict (act1, act2, targ_type) raises dcevent (z) {
  [z] = self.policies -> select ( pa,
    pb | pa.subject -> intersection(pb.subject) -> notEmpty
    and pa.action -> exists(act | act.name = act1)
    and pb.action -> exists(act | act.name = act2)
    and pb.target -> intersection(pa.target) ->
      oclIsKindOf (targ_type)
  );

  z -> notEmpty
}

```

This constraint can be instantiated in order to specify that the same users are not permitted to authorise a payment and initiate it:

```
inst meta dc = dutyconflict ("authorise", "initiate", "payment");
```

or that help-desk staff are not allowed to both update payment details and service subscriptions on customer records:

```
inst meta ss = dutyconflict ("serviceUpdate", "paymentUpdate",  
"customerRecords");
```

7. Related work

Policy-based systems have been the subject of increasing research efforts recently. In particular the main thrust for standardisation has been undertaken within the IETF Policy Group (www.ietf.org/html.charters/policy-charter.html). They use a policy model highly customised for quality of service management and configuration within networks. They assume policies are objects stored in a directory service [14]. A policy client (e.g., a router) makes policy requests on a server, which retrieves the policy objects, interprets them and responds with policy decisions to the client. The client enforces the policy by, for example, permitting/forbidding requests or allocating packets from a connection to a particular queue. 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 [15]. They do not explicitly differentiate authorisation and obligation policies. A simple policy rule defines a set of policy actions that are performed when a set of conditions becomes true. These conditions correspond to a combination of our events and constraints for obligation policies. Directories are used for storing policies but not for grouping subjects and targets. They use dynamic groups that can be specified by enumeration or by characterisation i.e., a predicate on object attributes. We can achieve this by means of a constraint on policies within the scope of a domain. Our notation, with explicit subjects and targets permits us to propagate policies to where they are required so we combine decision and enforcement at subjects for obligation policies and at targets for authorisation policies. Our policy service disseminates policies to the relevant distributed agents.

Sandhu [16] defines Role-Based Access Control (RBAC) structures using a direct inheritance of permissions between role instances. This permits reuse of permissions and follows an organisational structure in that more senior roles inherit the access rights of junior roles. However, this instance based inheritance model violates organisational control principles as shown by [17] and introduces the need to cater for a substantial number of exceptions as often some access rights should not be inherited by senior roles from junior roles. A more detailed comparison between our work and RBAC models is presented in [18], and a study on the different types of role hierarchies in RBAC was presented in [19].

There have been several attempts at defining Enterprise Viewpoint specifications by using the Unified Modelling

Language UML possibly with added extensions [20], [21], [22]. Proposals have alternated between modelling policies as notes attached to the design elements, OCL constraints or additional extensions to UML. [21] indicates that OCL is not directly suitable for expressing ODP prohibitions or obligations. Use cases have been presented as candidates for specifying community objectives and collaboration diagrams have been proposed for specifying communities. These approaches encounter difficulties related to the level of abstraction of the specification. Enterprise Viewpoint specifications are intended to be more general and more abstract than the system design elements of UML. Hence, Enterprise Viewpoint concepts are not defined in UML but can be realised (implemented) by a design that could be specified in UML. For example, an authorisation policy is implemented by a set of objects such as capabilities, certificates or access control list entries, and an obligation can be realised by the implementation of a particular activity diagram or collaboration. However, the enterprise level description requires a more abstract representation of the policy than the diagram of invocations, which implement it. It is therefore important to be able to identify a minimal and concrete representation for the Enterprise Viewpoint concepts, which represent functional aspects of the system rather than encumber a design notation with abstract concepts that need to be implemented by other representations in the same notation.

The various models proposed for defining Enterprise Concepts in UML are complex and differ substantially between them with little prospect of convergence or consensus. Ponder is a simple, easy to understand declarative language designed to be implementable.

[23] shows how policies expressed in structured English and Predicate Logic can be translated into Object Z. The benefits of this approach are in using Z for verification and analysis but there is no discussion in the paper of how the policies could be analysed. Z would still have to be translated into something suitable for implementation. See [10] for a discussion on conflict analysis for a previous version of our Policy Language.

8. Conclusions and Further Work

Ponder realises many enterprise viewpoint concepts using an intuitive, declarative language which can be directly mapped into an implementation. It is a typed language where individual policies, roles, relationships and management structures are themselves types. This provides a model that prescribes rigorous specification.

Getting the right domain structure which reflects the organisational requirement for object groupings and what objects need to be shared is an important starting point. Identifying roles in terms of rights and duties associated with positions is the next stage. Providing a representation for roles within the enterprise system permits the distribution of responsibilities reflecting organisational

structure as well as analysis for auditing and review purposes. Role classes permit the reuse of policy specifications and reflect the categorisation of classes of employees found in many organisations. Managers assigned to roles do not work in isolation but collaborate in a complex network of various types of relationships. Our role framework supports specification of the rights and duties imposed upon the related parties and the interaction protocols needed for collaboration.

We have a working editor and syntax analyser for Ponder and are experimenting with translation of Ponder into a variety of different representations – XML for ease of transfer and viewing by distributed administrators, Java for interpretation by automated agents. We are also mapping Ponder authorisation policies onto different security mechanisms – operating system mechanisms such as provided in Windows NT, Firewall rules or Java security policy. We are also developing graphical tools to support policy specification within a domain framework and policy analysis.

Acknowledgements

The work presented in this paper was funded by the EPSRC under Research Grants GR/L96103 (SecPol), GR/M86109 (Ponds) and GR/L76709 (Slurp), and by Fujitsu Network Systems Laboratories (Pro-Active Role Based Management for Distributed Services Project). We gratefully acknowledge their support.

References

See <http://www-dse.doc.ic.ac.uk/policies> for additional references.

- [1] Enterprise-Viewpoint Reference Model, CD 15414, ISO/IEC JTC1/SC7 N2187, 1999.
- [2] Damianou N, Dulay N, Lupu, E, Sloman M, Ponder: A Language for specifying Security and Management Policies for Distributed Systems, V 2.0 Imperial College Research Report DoC 2000/1 March 2000.
- [3] Sloman, M. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4):333-360, Plenum Press, 1994.
- [4] Marriott, D. A. and M. S. Sloman. Implementation of a Management Agent for Interpreting Obligation Policy. *IFIP/IEEE Distributed Systems Operations and Management (DSOM'96)*, L'Aquila, Italy, October 1996.
- [5] Lupu, E. A Role-Based Framework for Distributed Systems Management. Ph.D. Dissertation, Imperial College, Dept. of Computing, London, U.K., 1998.
- [6] Sloman, M. S. and K. P. 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.
- [7] ETSI. Digital cellular telecommunications systems (Phase 2); (GSM 12.00-12.04, 12.08,12.20). ETS 300 612-(1-5, 22, 27), 1996.
- [8] Mouly, M. and M.-B. Pautet. *The GSM System for Mobile Communications*. Published by the authors ISBN 2-9507190-0-7, 1992.
- [9] Lupu, E. C., and M. Sloman, A Policy Based Role Object Model, *Proc. 1st. Enterprise Distributed Object Computing Conf.*, Australia, 1997, pp36-47.
- [10] Lupu, E. C., and M. Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Trans. on Software Engineering*, 25(6): 852-869 Nov.1999.
- [11] Rational Software Corporation (1997). Object Constraint Language Specification, Version 1.1., Available at <http://www.rational.com/uml/>, Sep. 1997.
- [12] Jones, A. and Sergot M. On the Characterization of Law and Computer Systems: The Normative Systems Perspective. In *Deontic Logic in Computer Science*, J.-J.Ch. Meyer and R.J. Wieringa (eds.), John Wiley and Sons, 1993
- [13] Darimont, R. et al. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Proc. 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998, IEEE Computer Society Press, Vol. 2, pp. 58-62.
- [14] Mahon H., Bernet Y, Herzog S, Requirements for a Policy Management System, Oct. 1999, Available from <http://www.ietf.org/draft-ietf-policy-req-01.txt>
- [15] Moore, B. Strassner J. and Elleson, E., Policy Core Information Model V1, March 2000, Available from <http://www.ietf.org/draft-ietf-policy-core-info-model-05.txt>
- [16] Sandhu, R. et al. Role-Based Access Control Models. *IEEE Computer*, 29(2):38-47, 1996.
- [17] Moffett J, Control Principles and Role Hierarchies, 3rd. *ACM Role Based Access Control Workshop*, Fairfax, Virginia, 1998.
- [18] Lupu, E. C., and M. S. Sloman. Reconciling Role Based Management and Role Based Access Control. 2nd *ACM Role Based Access Control Workshop*, Fairfax, VA, 1997 pp. 135-142.
- [19] Moffett, J. and E. C. Lupu. The Uses of Hierarchies in Access Control, 4th *ACM Role Based Access Control Workshop*, Fairfax, Virginia, Oct., 1999
- [20] Blanc X., Gervais M., Le-Delliou R., Using the UML Language to Express the ODP Concepts, *Proc. 3rd. Enterprise Distributed Object Computing Conf.*, Germany, Sep.1999, pp.50-59.
- [21] Øyvind Aagedal J, Milošević Z, ODP Enterprise Language: UML Perspective, *Proc. 3rd. Enterprise Distributed Object Computing Conf.*, Germany, Sep.1999, pp.60-71
- [22] Linington P, Options for Expressing ODP Enterprise Communities and Their Policies by Using UML, *Proc. 3rd. Enterprise Distributed Object Computing Conf.*, Germany, Sep.1999, pp.72-82.
- [23] Steen M, Derrick J, Formalising ODP Enterprise Policies, *Proc. 3rd. Enterprise Distributed Object Computing Conf.*, Germany, Sep.1999, pp.84-93