Imperial College

OF SCIENCE, TECHNOLOGY AND MEDICINE

Department of Computing

180 Queen's Gate, London SW7 2BZ, U.K.

Ponder: A Language for Specifying Security and Management Policies for Distributed Systems

The Language Specification

Version 1.11

Imperial College Research Report DoC 2000/1

18th January, 2000

Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman

Contact: policy-99@doc.ic.ac.uk

http://www-dse.doc.ic.ac.uk/policies

ABSTRACT

This document defines a declarative, object-oriented language for specifying policies for the security and management of distributed systems. The language includes constructs for specifying the following basic policy types: authorisation policies that define permitted actions; event-triggered obligation policies that define actions to be performed by manager agents; refrain policies that define actions that subjects must refrain from performing; and delegation policies that define what authorisations can be delegated and to whom. Filtered actions extend authorisations and allow the transformation of input or output parameters to be defined. Constraints specify limitations on the applicability of policies while meta-policies define semantic constraints on permitted policies. Policy groups define a scope for related policies to which a common set of constraints can apply. Roles define a group of policies relating to positions within an organisation. Relationships define a group of policies pertaining to the interactions between a set of roles. Management structures define a configuration of role instances as well as the relationships between them. This document defines the grammar for the various types of policies in EBNF and provides simple examples of the constructs.

Keywords: Management, security, policy, delegation, role, management configuration

Ponder: to give thorough or deep consideration (to); mediate (upon)

CONTENTS

1 INTRODUCTION

This document acts as an informal language reference for Ponder, a language for specifying security and management policies for distributed systems. Ponder is derived from earlier policy specification notations developed at Imperial College over a number of years. (Sloman 1994b; Marriott and Sloman 1996; Marriott 1997). 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. It is intended to be extensible to cater for future types of policies. This document describes the grammar of the language and demonstrates its features through small examples. Some rationale for the design decisions is also included. Background information on the various language constructs can be found in the references given in section 9, although the syntax of the policy language has changed significantly.

Ponder is a declarative language with an object-orient model. Ponder does not assume a particular implementation platform; rather Ponder can map to, and co-exist with, one or more existing underlying platforms. We envisage a variety of 'back-ends' will be available. For example, we plan to provide back-ends that generate filters and access control lists for implementing security policy on various security aware platforms, e.g. operating systems such as Windows NT and Linux, distributed programming environments such as CORBA and JAVA, and technologies such as firewalls. Ponder can be used to manage one or more of these platforms simultaneously. Ponder could also be used to generate IETF policy schema for quality of service related policies, XML for transport across the network and ease of viewing via XML aware browsers.

1.1 Policy Concepts Overview

In Ponder, a **policy** is a rule that can be used to change the behaviour of a system. Separating policies from the managers that interpret them allows the behaviour and strategy of the management system to be changed without re-coding the managers. The management system can then adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system.

Ponder supports an extensible range of policy types. **Authorisation** policies are essentially security policies related to access-control and specify what activities a subject is permitted or forbidden to do, to a set of target objects. They are designed to protect target objects so are interpreted by access control agents or the run-time systems at the target system. **Obligation** policies specify what activities a subject must do to a set of target objects and define the duties of the policy subject. Obligation policies are triggered by events and are normally interpreted by a manager agent at the subject. **Refrain** policies specify what a subject must refrain from doing and are similar to negative authorisation policies but are interpreted by the subject. **Delegation** policies specify which actions subjects are allowed to delegate to others. A delegation policy thus specifies an authorisation to delegate. **Composite policies** are used to group a set of related policy specifications within a syntactic scope with shared declarations in order to simplify the policy specification task for large distributed systems. Four types of composite policies are provided: groups, roles, relationships and management structures. **Constraints** can be specified to limit the applicability of policies based on time or values of the attributes of the objects to which the policy refers. **Meta-policies** are policies about which policies can coexist in the system or what are permitted attribute values for a valid policy. For example, a semantic conflict may arise if there are two policies which increase and decrease bandwidth allocation when the same event occurs, or a conflict of duty may arise if there is a policy permitting the same manager to both sign cheques and authorise payment.

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 (Sloman and Twidle 1994a; Sloman 1994b). Membership of a domain is explicit and not defined in terms of a predicate on object attributes. A domain does not encapsulate the objects it contains but merely holds references to object interfaces. A domain is thus very similar in concept to a file system directory but may hold references to any type of object, including a person. A domain, which is a member of another domain, is called a **sub-domain** of the parent domain. Objects can be members of multiple domains i.e. domains can overlap. Path names are used to identify domains. In figure 1, domain D can be referred to as `/A/B/D` or `/A/C/D` as an object may have different local names with multiple parent domains, where `/` is used as a delimiter for domain path names. Policies normally propagate to members of sub-domains, so a policy applying to domain C will also apply to members of domains D and E.

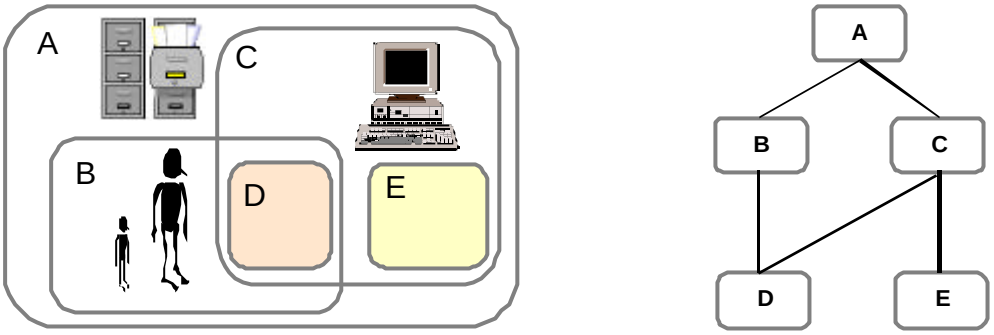

Figure 1. Domains

Organisational structure is often specified in terms of **organisational positions** such as regional, site or departmental network manager, service administrator, service operator, company vice-president. Specifying organisational policies for people in terms of role-positions rather than named persons permits the assignment of a new person to the position without re-specifying the policies referring to the duties and authorisations of that position. The tasks and responsibilities corresponding to the position are grouped into a role associated with the position (which is essentially a static concept in the organisation). The position could correspond to a manager or a user of a network or services. A **role** is thus the position, the set of authorisation policies defining the rights for that position and the set of obligation policies defining the duties of that position as defined in the Imperial College role-based management framework (Lupu 1998). All policies within a role have the same subject domain. A person or automated agent can then be assigned to or removed from the subject domain without changing the policies, as explained (Lupu and Sloman 1997b; Lupu and Sloman 1997c).

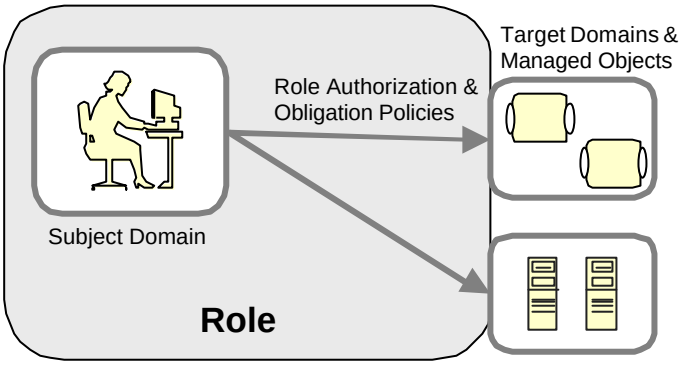

Figure 2. Management Roles

It is useful to group the policies, constraints and interaction protocols relating to common relationships between a number of roles. For example a supervision relationship between a head of department and group leader or a lecturer–student relationship. **Role relationships** specify policies about the interaction between roles, policies relating to shared objects and the protocols for interaction.

Organisations often have branches or departments with similar roles and relationships e.g. a branch of a bank or university department. **Management structures** are used to define configurations of role and relationship instances within an organisational unit. The management structure can then be instantiated for each branch.

2 PRELIMINARIES

2.1 Syntax

The syntax of Ponder is defined using the EBNF notation as specified in ISO/IEC 14977:1996(E). The most important features of EBNF used in this document are as follows:

- Terminal identifiers/symbols are quoted
- [and] indicate optional elements
- { and } indicate repetition. Zero or more elements
- (and) group items together
- | is the definition separator symbol. It separates alternatives in a grammar rule
- = is the defining symbol. On the left-hand side is the name of the grammar rule, and on the right-hand side is the definition of that name
- ; is the terminator symbol. Every rule is terminated by this symbol
- , is the concatenate symbol. Different terms in the same rule are separated by this symbol
- { and }- represents a sequence of one or more of the elements specified within the braces

The grammar syntax rules are indicated in `constant width` font type. Examples are presented in *`italic constant width`* font type with language keywords in ***`bold`***.

2.2 Lexical Conventions

2.2.1 Comments

The characters `/*` start a multi-line comment which terminates with the characters `*/`. The characters `//` start a single-line comment which terminates at the end of the line on which they occur. The characters `/*` and `//` have no special meaning within a multi-line comment, and the characters `//`, `/*` and `*/` have no special meaning within a single-line comment, so they are treated as part of the comment text.

2.2.2 Identifiers

An identifier in Ponder is an arbitrarily long sequence of letters and digits. The first character of an identifier must be a letter, other than the underscore _, which is also considered a letter. Upper and lower case letters are distinguished, and all characters are significant.

```
ident = letter, { letter | digit | '_' } ;
letter = l_case | u_case ;
u_case =  'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
          'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' |
          'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
l_case =  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
          'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' |
          'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
digit =   '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

Examples

```
Managers        x_coord     year_2000   SATURDAY
```

2.2.3 Paths

Paths in Ponder are used to indicate the location of an object, policy type definition or policy instance in the domain hierarchy (see section 2.5 Domain Scope Expressions). Paths are either absolute or relative and defined similarly to Unix file pathnames.

```
path = absolute_path | relative_path ;
absolute_path = '/' | {'/', path_seq}- ;
relative_path = path_seq, {'/', path_seq}- ;
path_seq = (non_digit, {digit | non_digit}) | '..' | '.' ;
ident_or_path = ident | path;
prefix_ident =  ident_or_path, {'.', ident};
```

Examples

```
/dept/sales/salesmen        ../secretaries          ./
```

2.2.4 Keywords

The following symbols (mostly identifiers) are reserved for use as keywords:

```
action      and         auth+       auth-       boolean
catch       char        constraint  deleg+      deleg-
do          domain      double      event       extends
false       grantee     group       import      in
inst        int         meta        mstruct     oblig
on          or          raises      refrain     rel
result      role        spec        string      subject
target      true        type        when        xor
```

The following identifiers are keywords adopted from the Object Constraint Language - OCL (Rational 1997):

```
bag         collect     collection  else        endif
enum        exists      forall      implies     iterate
not         reject      select      sequence    set
then
```

2.2.5 Operators

The following characters are used as operators. No white-space is permitted between two character operators.

```
@   !     ->    ||    &&    ^     %     =     <>    <     <=
>   >=    +     -     *     /
```

The following characters are used as operators and/or for punctuation:

```
|   ..    #     ::    (     )     {     }     [     ]     .
:   ,     ;
```

2.2.6 Literals

The following literals (often referred to as constants) are supported by the grammar. Their meaning is taken from similarly named types in the OMG IDL grammar (OMG 1999), Chapter 3.

Integer-constant – consists of a sequence of digits and is taken to be decimal (base ten).

Double-constant – consists of an integer part, a decimal point, a fraction part and an optional exponent part. The integer and fraction parts both consist of a sequence of decimal digits. The exponent part contains an `e` or `E`, an optional sign (`+` or `-`) and an integer number.

Character-constant – any character enclosed in single quotes, other than quote itself (') and non-printable characters.

String-constant – a sequence of characters surrounded by double quotes.

Boolean-constant – takes the values true or false, denoted by the reserved keywords `true` and `false`.

Examples

```
666      3.14159265385     10E+12      '%'    “administrator”    true
```

2.3 Pre-defined Types and Constants

The following types are predefined in Ponder:

```
int, double, char, string, boolean, domain, subject, target, event.
```

Two constants are also pre-defined: `true` and `false`.

2.4 Expressions

Expressions in Ponder follow the IDL constant expressions syntax extended to include the Object Constraint Language (OCL) syntax. The operators used for specifying constraints are taken from the OCL syntax. IDL constant expressions are also extended to include action calls. Constraint expressions are specified in OCL syntax, but OCL literals are a subset of those specified for general expressions in Ponder.

2.5 Domain Scope Expressions

Domain scope expressions are used to combine domains to form a set of objects for applying a policy to. The set of objects (i.e. the domain scope expression) to which a policy applies is evaluated each time that the policy is interpreted because domain membership can change dynamically. Note: in practice, implementation optimisations are used to minimise run-time evaluation.

The different domain scope expression operators are explained in table 1. Note: the set union, difference and intersection operators have equal precedence and are evaluated left to right.

Syntax	Explanation
d	Returns all non-domain members of the domain and all distinct non-domain members of all nested sub-domains recursively traversed all levels down the domain structure.
@*nd*	If *d* is a domain, returns a set that contains all non-domain members of the domain. The integer constant *n* specifies that the domain structure is to be traversed *n* levels down, e.g. n = 1 specifies only direct members, whereas n = 2 would include distinct members of the sub-domains of *d* also. If *d* is a non-domain object, returns a set that contains the non-domain object.
**d* **nd*	Returns a set that contains all non-domain and all domain members of the domain *d*, including the domain itself. The integer constant *n* specifies that the domain structure is to be traversed *n* levels down. If *n* is omitted, all nested sub-domains are recursively traversed.
{c}	Returns a set that contains the object *c*. This braces syntax is needed if *c* is an identifier in order to produce a set from it.
a+*b*	Returns a set that contains all distinct members of *a* and *b* (Set Union).
a^*b*	Returns a set that contains only members that are in both *a* and in *b* (Set Intersection)
a-*b*	Returns a set that contains members of *a* that are not also in *b* (Set difference)

Table 1. Domain Scope Expressions

```
domain_scope_expression =
    path                                                    |
    '{', object, '}'                                        |
    '*', [int_value], object                                |
    '@', int_value, object                                  |
    '(', domain_scope_expression, ')'                       |
    domain_scope_expression, '+', domain_scope_expression |
    domain_scope_expression, '-', domain_scope_expression |
    domain_scope_expression, '^', domain_scope_expression ;

object := ident | path ;
```

Examples

Given:

Domain	Direct Members
A	{B, C, a1, ab, ac, x}
B	{D, b1, ab, bc, bd, x}
C	{D, E, c1, ac, bc, cde, x}
D	{d1, bd, cde, x}
E	{e1, cde, x}

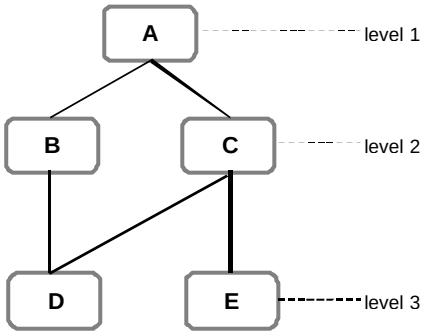

Figure 3. Domain structure

Domain Scope Expression	Resulting Set
/A	{a1, ab, ac, x, b1, bc, bd, c1, cde, d1, e1}
/A/B	{b1, ab, bc, bd, x, d1, cde}
/A/C	{c1, ac, bc, cde, x, d1, bd, e1}
/A/B + /A/C	{b1, ab, bc, bd, x, d1, cde, c1, ac, e1}
/A/B + /A/C - /A/B/D	{b1, ab, bc, x, c1, ac, e1}
*/A	{A, B, C, D, E, a1, ab, ac, x, b1, bc, bd, c1, cde, d1, e1}
*/A/B	{B, D, b1, ab, bc, bd, x, d1, cde}
*/A/C	{C, D, E, c1, ac, bc, cde, x, d1, bd, e1}
*/A/B ^ */A/C	{D, bc, bd, x, d1, cde}
@1/A	{a1, ab, ac, x}
*2/A	{A, B, C, D, E, a1, ab, ac, x, b1, bc, bd, c1, cde}

3 PONDER SPECIFICATIONS

A Ponder specification consists of type definitions, instance declarations, domain statements and import statements.

```
ponder_specification =
    {import_or_domain | type_or_instance} ;

import_or_domain = (import_statement | domain_statement), [';'] ;

type_or_instance =
    ('type', {type_definition}-) | ('inst', {inst_declaration}-) ;
```

3.1 Ponder Policies

Ponder supports the following kinds of policies:

Basic policies	**Keyword**
Positive Authorisation Policy	`auth+`
Negative Authorisation Policy	`auth-`
Obligation Policy	`oblig`
Refrain Policy	`refrain`
Positive Delegation Policy	`deleg+`
Negative Delegation Policy	`deleg-`
Composite policies	**Keyword**
Group	`group`
Role	`role`
Relationship	`rel`
Management Structure	`mstruct`
Other	**Keyword**
Meta-Policy	`meta`

Table 2. Ponder Policies

Ponder policies can be visualised as base classes forming an inheritance hierarchy. Classes in *italic font* in the following diagram (figure 4) are abstract classes. There is a concrete class for each of the Ponder policies specified in table 2. Users can create instances of concrete classes directly, or use type definitions to effectively create user-defined sub-classes of the corresponding base-class. Base-classes can be thought of as templates from which instances and types can be created in an object-oriented fashion.

Extending the Ponder language to cater for new kinds of policies is simplified using an underlying object-oriented implementation. Ponder can be extended by adding new base sub-classes to the existing ones, or by adding new attributes to existing base classes

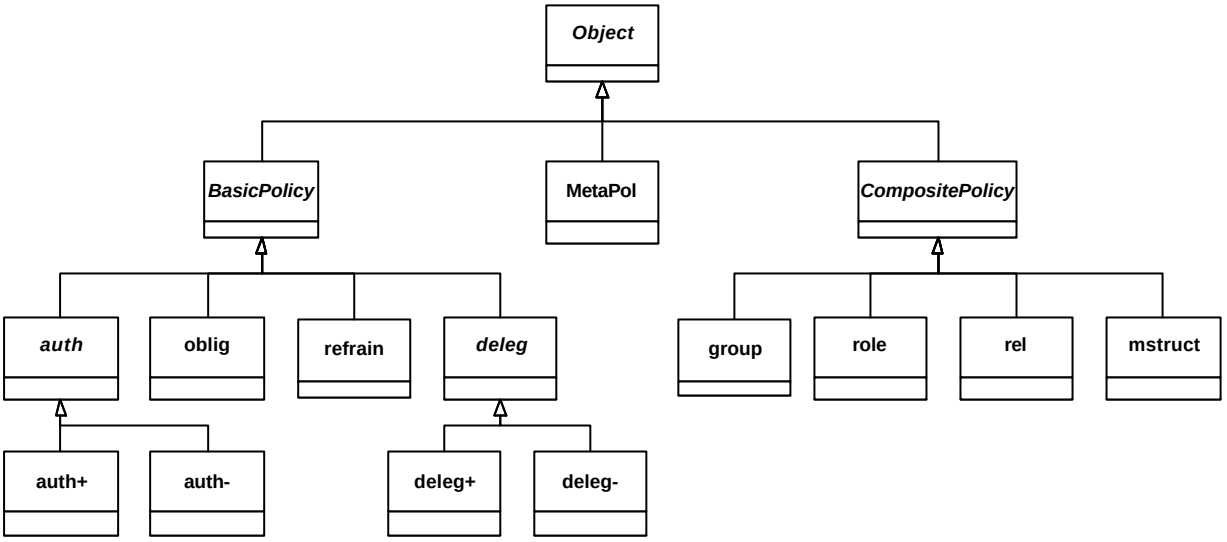

Figure 4. Ponder Base-Class Diagram

3.2 Scope

A name declared in a syntactic block (typically in a policy) is local to that block and can be used within it. Forward references to identifiers declared later in the same scope are allowed.

3.3 Policy Type Definitions

A type definition introduces a new user-defined policy type, from which one or more policy instances of that type can be created. The name of the policy type is specified as an identifier, or as a domain path to indicate the place within the domain structure where the type definition will be stored. In the case of an identifier or a relative path, the policy type is stored relative to the current working domain, which can be specified with the domain definition (see section 3.5). Each policy type definition can be optionally followed by a semicolon.

```
type_definition = (policy_type  | group_type | role_type | rel_type |
                   mstruct_type | meta_type), [';'] ;

policy_type =   pos_auth_type | oblig_type | neg_pol_type | deleg_type ;
```

Examples

```
type
    oblig allocBwT(subject m, target o) {
          on perfDegradation(bw,source)
          do bwReserve(bw+10)
    } // allocBwT
```

3.4 Policy Instance Declarations

A policy instance declaration creates an instance of a user-defined policy type. The name of the policy instance is specified either as an identifier, or as a domain path to indicate the place within the domain structure where the policy instance will be stored. In the case of an identifier or a relative path, the policy instance is stored relative to the current working domain. A policy instance in Ponder can also be specified inline without specifying a user-defined policy type. Each instance declaration can be optionally followed by a semicolon.

```
instantiation = ident_or_path , '=',  actual_call_decl ;

actual_call_decl = ident_or_path, '(', [actual_parameters], ')' ;

inst_declaration = (policy_inst | group_inst | role_inst | rel_inst |
                    mstruct_inst | meta_inst), [';'] ;

policy_inst = pos_auth_inst | oblig_inst | neg_pol_inst | deleg_inst ;
```

Examples

The following example shows the declaration of two instances of the user-defined obligation policy type *allocBwT* with different subjects and targets and a third policy instance declared in-line.

```
inst
    oblig site1/perf = allocBwT(site1/netOp, site1/edgeRtr)
    oblig site2/perf = allocBwT(site2/netOp, site2/edgeRtr)
    oblig allocBW {
          subject netOp; target edgeRtr
          on perfDegradation(bw,source)
          do bwReserve(bw+10)
    } // allocBW
```

3.5 Domain Statements

A domain statement has two distinct uses:

- To introduce a short local name for a longer domain path.
- To set the **current working domain,** which defines the domain where policy types and policy instances will be stored when no explicit domain path is given in their definition/declaration. The current working domain applies to subsequent type definitions and instance declarations within the current scope or until another domain statement is encountered in the current scope.

```
domain_statement = 'domain', (path | (ident, '=', domain_scope_expr)) ;
```

Examples

In the following example, the `serviceFailT` group policy instance is stored in `/region/branchA`, whereas the `auth+` policy instance `serviceConfig` is stored in `/region/branchB`.

```
domain a = /region/branchA        // a is a name for /region/branchA

inst
    group a/serviceFailT {
          import /typeRepository/serviceDefT
          oblig serviceReset { subject a/brManager; on e; … }
    } // a/serviceFailT

domain /region/branchB            // set current working domain

inst
    auth+ serviceConfig { … }
```

3.6 Import Statements

The import statement is used to bring into the current scope, policy type definitions, policy instances, constant definitions, event definitions and scripts (see below) stored in other domains. An import statement specifies a path to the domain, or to the particular definition/declaration that is to be imported. A domain path followed by a `/-` will cause all the definitions/declarations within the specified domain to be imported.

```
'import', (ident_or_path) ;
```

Examples

```
type
    group serviceDefT (subject s1, target t1) {
          import /myEvents/timeoutEvent // imports single event
          import /myTypes/-             // imports all definitions
          event e = 3*timeoutEvent(s)
          inst auth+ a = serviceReset (s1, t1)
    } // serviceDefT
```

3.6.1 Scripts

A script is an externally-defined code object that can be imported into a Ponder specification from a domain, and invoked as an action in an obligation policy or as a filter in a positive authorisation policy. Scripts are typically used when it is necessary to perform a more complex set of actions than is possible with Ponder. Any suitable programming/scripting language can be used for writing scripts. Since scripts are objects, Ponder policies can be applied to script objects.

3.7 Event Definitions

Events in Ponder are used to trigger obligation policies. It is convenient to be able to define events separately, and re-use them in multiple obligation policies. Event expressions can be used to combine basic events into more complex ones.

Table 3 specifies the event composition operators that can be specified in event expressions. All event operators have equal precedence and evaluated is strictly left to right.

Operator	Explanation
e1 && e2	Occurs when *both* e_1 and e_2 occur irrespective of their order
e + time-period	Occurs a specified period of time after the occurrence of event e
$\{e_1$; $e_2\}$! e_3	Occurs when e_1 occurs followed by e_2 with no interleaving e_3
e_1 | e_2	Occurs when either e_1 or e_2 occurs irrespective of their order
e_1 -> e_2	Occurs when e_1 occurs before e_2
n * e	Occurs when e occurs n times, where n is an integer value

Table 3. Event Composition Operators

```
event_def = ident, [event_params], '=', event_expr [';'];

event_expr =
    basic_event                                                 |
    (basic_event, next_event)                                   |
    (int_value, '*', event_expr)                                |
    ('{', event_expr, ';', event_expr, '}', '!', event_expr)    ;

next_event =
    (event_op, event_expr) | ('+' int_value) ;

event_op = '&&' | '|' | '->' ;

basic_event =   (ident, [event_params]) | (ident, '.', action_call) |
                '(', event_expr, ')' ;

event_params =  '(', [formal_parameters], ')' ;
```

Examples

In the following, a timer object for generating time-based events is assumed. The first event occurs at a particular date (15 Dec. 2001) and time (22:15:00), the second event occurs every 24 hours at 07:20. The third event `circuitFailure(h,x,y)` demonstrates the use of parameters in the definition of an event. The named event receives three parameters (`h,x,y`) that can be referenced in the obligation policy that uses this event. The first parameter corresponds to the parameter of the `envAlarm(h)` while the second and third to the two parameters of `rFailure(x,y)`. The two events that are used in the event expression are assigned to the new event. You can see how the first parameter is used in the specification of the target in the obligation policy `resetCircuit`.

```
event
    a = timer.at("2001:12:15", "22:17:00") ;
    b = timer.every(24, "07:20") ;
    circuitFailure(h,x,y) = (envAlarm(h) -> rFailure(x,y))

inst
    oblig resetCircuit {
          subject brEngineer
          on circuitFailure(h,x,y)
          do resetCircuit
          target brCircuits/h
    } // resetCircuit
```

3.8 Constraint Definitions

Constraints are used to limit the applicability of basic policies e.g. in the constraint part of these policies – the when-clause (see section 4.1). Constraint definitions allow constraints to be separately defined and multiply used. A constraint in Ponder is an OCL expression. In the specification of constraints, `time` is assumed to be a predefined object on which operations such as `between`, `before` or `after` can be invoked related to the current time. The distinction between time and other constraints is helpful for conflict analysis of policies.

```
constraint_def = ident, [constraint_params],'=', constraint_spec, [';'];

constraint_params = '(', [formal_parameters], ')';

constraint_spec = ocl_expression;
```

Examples

In the following example, two constraints are specified, which are both used in the specification of the constraint on the obligation policy `serviceReset`. The first constraint takes a parameter `s`, which is used in its specification. The second constraint `workHours`, is a time constraint, and is valid only between 8:00am and 4:00pm.

```
constraint
    active(s) = s.isActive() and s.isEnabled();
    workHours = time.between(0800, 1600);

type
    oblig serviceReset(subject s, target t) {
          on e
          do t.reset()
          when active(s) and workHours
    } // serviceReset
```

The second example demonstrates the use of a more complicated constraint limiting the applicability of the policy specified. The constraint is directly specified in the when clause of the policy.

```
type
    oblig perfIncreaseT (subject s, target t) {
          on perfDegradation(bw, source)
          do t.bwReserve(bw) -> s.log(bw, source)
          when ( s.a>5  and (t.b+7)<10 and time.between(1200,1400) )
            or ( s.a>15 and (t.b+7)<20 and time.between(0200, 0400) )
            or active(s) ;
    } // perfIncreaseT
```

3.9 Constant Definitions

Constants can be defined in Ponder. A type identifier can be used to indicate the user-defined type for which a constant is declared.

```
constant_def = constant_def_aux, [';'] ;
constant_def_aux =
    'int',      ident, '=',  int_value        |
    'double',   ident, '=',  double_value     |
    'char',     ident, '=',  char_value       |
    'string',   ident, '=',  string_value     |
    'boolean',  ident, '=',  boolean_value    |
    type_ident  ident, '=',  expr_or_path     |
                ident, '=',  expr_or_path     ;

expr_or_path = expression | path;
```

Examples

Any of the types shown in the syntax can be specified. Here are a few examples.

```
const
    int y = 5;
    x = managerX.getName();
    string str1 = "this is a string"
```

3.10 External Specifications

External specifications are used to embed non-Ponder text into a Ponder specification. Unlike comments which are un-named and ignored by the Ponder compiler, external specifications are named and preserved by the Ponder compiler and runtime system. Such specifications can be accessed by external tools either at compile-time and/or run-time. External specifications are typically used to develop Ponder variants/extensions or attach non-Ponder definitions, code, scripts, performance and protocol requirements, structured documentation etc. with a Ponder specification.

```
external_spec = ident, '<<' any-sequence-of-characters '>>' ;
```

Examples

In the following example, an external specification named `refs`, associated with an authorisation policy specifies references to related obligation policies for which it is required as well as a parent policy from which it is refined and child policies which are derived from it. An analysis tool can extract the specification, parse it and interprete it accordingly.

```
inst
    auth+ net_config {
          subject netOp
          action setStrategy
          target qEdgeRtr

          spec refs <<
                related net_config2, net_config3;
                parent  config
                child   router_config
          >> // refs
    } // net_config
```

3.11 Parameters

This section defines the syntax of formal parameters and actual parameters.

3.11.1 Formal Parameters

All policy types can be parameterised. Parameters can be one of the predefined types (e.g. int, string, domain, subject, target, event) or of a user-defined type. If the type of a parameter is omitted then the type will be inferred either at compile-time or run-time. The possible types that can be specified or declared in Ponder are given by `type_decl` below:

```
type_decl =
    'int'       |     'double'    |     'char'      |     'string'    |
    'boolean'   |     'domain'    |     'subject'   |     'target'    |
    'grantee'   |     'event'     |     'action'    |     'auth+'     |
    'auth-'     |     'oblig'     |     'refrain'   |     'deleg+'    |
    'deleg-'    |     'role'      |     'rel'       |     'group'     |
    'mstruct'   |     'meta'      |     type_ident  ;

type_ident = ident_or_path;


formal_call_decl =
    ident_or_path, '(', [formal_parameters], ')', [extends_type];

formal_parameters = formal_param, {',', formal_param};

formal_param = [type_decl], ident ;
```

Examples

```
auth+ myAuthPolicy (subject a, int b, event e) { ... }
```

3.11.2 Actual Parameters

Actual parameters are used in instance declarations, action calls and exception-clauses. An actual parameter can be an expression or a domain-scope-expression. Actual parameters must correspond in number and type to the formal parameters of the corresponding formal parameter.

```
actual_parameters = actual_param, {',', actual_param} ;

actual_param  = expression | domain_scope_expr ;
```

4 BASIC POLICIES

Basic policies	Keyword
Positive Authorisation Policy	auth+
Negative Authorisation Policy	auth-
Obligation Policy	oblig
Refrain Policy	refrain
Positive Delegation Policy	deleg+
Negative Delegation Policy	deleg-

4.1 Policy Elements

The body of a basic policy consist of one or more policy elements. Several of these elements are common to all basic policy types: the subject, the target, the when-constraint, as well as import statements, event definitions, constant definitions and external specifications. Other policy elements are specific to a particular policy type. Policy elements can be specified in any order.

The **subject** and the **target** for a basic policy are specified using domain scope expressions or by formal identifier of type subject or target. Actual parameters for subjects and targets are domain scope expression.

Each basic policy can also optionally specify a **when**-constraint element that limits the applicability of the policy.

```
policy_elements =
    ('subject', subj_target       |
     'target',  subj_target       |
     'when',    constraint_spec   |
     import_statement ),    [';'] |
     common_element_spec          ;

subj_target = ident | domain_scope_expr ;

common_element_spec =
    'event',          {event_def}-      |
    'constraint',     {constraint_def}- |
    'const',          {constant_def}-   |
    'spec'            {external_spec}-  ;
```

4.2 Authorisation Policies

An authorisation policy specifies access control for security. A positive authorisation policy defines the actions that a subject is permitted to perform on a target. A negative authorisation policy specifies the actions that a subject is forbidden to perform on a target. Positive authorisation policies may also include filters to transform the parameters associated with their actions. Authorisation policies are implemented on the target host by an access control agent (ACA) utilising an access control decision facility associated with the target objects.

4.2.1 Positive Authorisation Policies

Positive Authorisation Policies define the actions subjects are permitted to perform on target objects.

```
pos_auth_type =
    'auth+', formal_call_decl, '{', {pos_auth_type_body}, '}' ;

pos_auth_inst =
    ('auth+', ident_or_path, '{', {pos_auth_type_body}, '}') |
    ('auth+', instantiation) ;

pos_auth_type_body =
    policy_elements | ('action', pos_auth_actions, [';']) ;
```

Authorisation Actions

Actions represent the operations defined in the interface of a target object. The permitted/forbidden actions are listed separated by commas. In an authorisation policy the actions can alternatively be specified using `'*'`. This means that the subject is authorised to perform all of the actions visible on the target object interface thus this feature should be treated with caution.

```
pos_auth_actions = (pos_auth_action_decl, {',', pos_auth_action_decl}) |
                   '*' ;

pos_auth_action_decl = auth_action, filter;

auth_action = [ident_or_path, '.'], ident, [auth_parameters_decl] ;

auth_parameters_decl = '(', ident_list, ')' ;

ident_list = ident, {',', ident} ;
```

For authorisation policies, parameters can be omitted from the action even though the action may actually have parameters. This indicates that we don't care about the parameters. In general, parameters for authorisation policies are specified as a list of identifiers. The identifier can then be used within the policy constraint clause to indicate a restriction on the parameter value. Authorisation action names can be optionally prefixed with the target object/domain of the policy.

Examples

The following is a simple example to demonstrate the syntax for specifying positive authorisation policies.

```
type
    auth+ serviceManT(subject s, target t) {
          action resetSchedule, enable, disable
    } // serviceManT

inst
    auth+ brService = serviceManT (brManager, brServices)
```

Authorisation Filters

Filters specify optional transformation of parameters related to an action only for positive authorisation policies as no transformation need take place if the action is forbidden. Filters may transform or select subsets of the information provided in the `in` and `out` parameters or the result of the invocation. A separate filter must be specified for every action in the authorisation policy. Multiple filters can be associated with a basic-policy. Filters consist of two parts:

- An optional condition based on subject/target state, action parameters or time specified using OCL for consistency with other types of constraints.
- The specification of a transformation expression or (external) function to be applied to the in, out or result parameters of the action call.

When the authorised action to which a filter is associated is invoked, the filter condition will be evaluated. If it evaluates to true, or it was omitted, then the filter will be executed.

```
filter =  ['if', ocl_expression], '{', {filter_body}-, '}' ;

filter_body =
    'in',       ident, '=', expression |
    'out',      ident, '=', expression |
    'result',          '=', expression ;
```

Examples

In the following example, the subject `s` is authorised to perform the operation `lookup(x,y)` on the target of the policy `t`. The `if`-clause of the filter associated with `lookup` checks whether the subject belongs in the group `extUsers`. It modifies the value of the second parameter `y`, which is both input and output to the action `lookup(x, y)`. It also transforms the result of the action, by calling an external function `selectBuilding(result)` for example to remove room details from the result.

```
type
    auth+ filterLocationT (subject s, target t) {
          action lookup(x,y) if belongs(s, extUsers) {
                in x = x-1
                out y = maths.abs(y)
                result = selectBuilding(result)     // external
          } // lookup
    } // filterLocationT
```

4.2.2 Negative Authorisation Policies

Negative Authorisation Policies define the actions subjects are forbidden (not permitted) to perform on target objects. They are are commonly used in many systems such as database and Web access control and in systems where the default policy permits access by anyone unless explicitly forbidden. Negative authorisation policies can also be used to temporarily restrict rights for a sub-domain or an individual object as an exception to the normal positive authorisation, which applies for a parent domain. For example suspension of access to the computer service for a week as a punishment for a student who has abused the system.

Note that allowing negative and positive policies can lead to conflicts and the need for precedence relationships between types of policies as discussed in (Lupu 1999). These issues are not part of the language although the policy precedence could be specified as a meta-policy.

Actions

The actions specify the operations that the subject is forbidden to perform on the target. The specification of negative authorisation actions is the same as for positive actions except there is no need for filters. The `'*'` character can be used to indicate all actions on the interface of target objects.

Negative authorisation policies have exactly the same syntax as refrain policies.

```
neg_pol_type =
    ('auth-' | 'refrain'), formal_call_decl, '{', {neg_type_body}, '}' ;

neg_pol_inst =
    (('auth-' | 'refrain'), ident_or_path, '{', {neg_type_body}, '}') |
    (('auth-' | 'refrain'), instantiation) ;

neg_type_body =
    policy_elements |('action', neg_pol_actions, [';']) ;

neg_pol_actions = (auth_action, {',', auth_action}) | '*';
```

Examples

```
type
    auth- serviceWithdrawT (subject s, target t) {
          action t.unload, t.remove
    } // serviceWithdrawT
inst
    auth- brWithdraw = serviceWithdrawT (brEngineer, brServices)

    auth- adminConfig {
          subject     configAgent
          action      setBW, reset
          target      links
    } // adminConfig
```

4.3 Obligation Policies

Obligation policies specify the action that a subject must perform on a set of target objects when an event occurs. Obligation policies are always triggered by events, since the subject must know when to perform the specified action. Unlike authorisation policies, obligation policies are interpreted by subjects. An exception can be used to specify an alternative action to cater for network or target object failures.

```
oblig_type = 'oblig', formal_call_decl , '{', {oblig_type_body}, '}';

oblig_inst =    ('oblig', ident_or_path , '{' {oblig_type_body}, '}') |
                ('oblig', instantiation) ;

oblig_type_body =
    policy_elements                     |
    ('subject', oblig_subj_target       |
     'target',  oblig_subj_target       |
     event_spec                         |
     'do',      oblig_actions           |
     'catch',   exception_spec),  [';'] ;
```

4.3.1 Obligation Actions

An obligation action consists of actions separated with concurrency operators indicating whether the actions are to be performed sequentially or in parallel. The action can be prefixed with the name of the object on which the action is called, as actions may be on the target, internal to the subject or part of the subject's interface. An object prefix is either one of the keywords `subject`/`target` or an identifier/path. An identifier/path indicates a specific object/domain on which the method is called. `subject.`*`actionName`*`(...)` means that the action is defined on the subject of the policy.

The concurrency operators for obligation policy actions are given in the following table.

Operator	Explanation
a_1 -> a_2	a_1 must follow a_2
a_1 || a_2	a_1 and a_2 may be performed concurrently. Execution continues when either has finished
a_1 && a_2	a_1 and a_2 may be performed concurrently. Execution continues when both have finished
a_1 | a_2	a_1 or a_2 can be performed (non-deterministic choice)

Table 4. Concurrency Operators

```
oblig_actions = basic_oblig_action  |
                basic_oblig_action, next_oblig_action ;

next_oblig_action = concurrency_op, oblig_actions ;

basic_oblig_action =  oblig_action_decl | '(', oblig_actions, ')' ;

oblig_action_decl =
    [oblig_subj_target, '.'], action_name,'(', [actual_parameters], ')' ;

concurrency_op = '->' | '|' | '||' | '&&' ;

action_name = [object_prefix], ident ;

object_prefix = (ident_or_path | 'subject' | 'target' |
                 oblig_action_decl), '.' ;
```

4.3.2 Events

The specification of events in the body of an obligation policy define the trigger for the action.

```
event_spec ='on', event_expr, {',', event_expr} ;
```

Details of `event_expr` are specified in section 4.8.1.

4.3.3 Exceptions

An exception specifies an optional single action (which can be a script) to be performed in case of failure of the normal obligation actions. An exception "parameter" from the runtime exception system is passed as an argument to the exception action.

```
exception_spec = ident, '(', [actual_parameters] ,')';
```

Examples

In the first example the actions are specified in the obligation policy type `perfIncreaseT`. The policy is triggered by a performance degradation event `perfDegradation(bw, source)` and the event parameters `(bw, source)` and reused in the specification of the actions. The subject of the policy invokes the action `bwReserve(bw)` on the target object followed by the action `log(bw, source)`, which is implemented on the interface of the subject.

```
type
    oblig perfIncreaseT (subject s, target t) {
          on perfDegradation(bw, source)
          do t.bwReserve(bw) -> s.log(bw, source)
    } // perfIncreaseT

inst
    oblig pl = perfIncreaseT(brEngineer, coreRouter+edgeRouter)

    oblig perfIncrease {
          subject brEngineer
          target {coreRouter} + {edgeRouter}
          on perfDegradation(bw, source)
          do t.bwReserve(bw) -> s.log(bw, source)
    } // perfIncrease
```

Consider the following obligation policy type instantiated as `da1` which indicates that when the patient's temperature exceeds 37 degrees, a nurse should administer analgesics to that patient. In this case only one of the nurses in `wardA` must administer the drug as if all the nurses performed the action the patient would probably die.

```
type
    oblig drugsAdminT1 (subject s, target t) {
          on t.temperature > 37
          do administer(analgesics)
    } // drugsAdminT1

inst
    oblig da1 = drugsAdminT1(/wardA/nurse, /sectionD/patient/stevens)
```

4.4 Refrain Policies

Refrain Policies define the actions that subjects must refrain from performing (must not perform) on target objects and like obligations they are implemented by the subject. Refrain policies 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. See section 4.10.2 for the grammar. Subjects and targets in refrain policies are specified as domain scope expressions.

Examples

In this example the `HQStaff` are assumed to be permitted to set up video conferences but a refrain policy states they must not do so to any destination on Fridays.

```
inst
    refrain politeBehaviour {
          subject HQStaff
          target /-                // Any target
          action videoconference
          when time.day(Friday)
    } // politeBehaviour
```

4.5 Delegation Policies

A Delegation policy specifies which actions subjects are allowed to delegate to others. A delegation policy is thus specifying an authorisation to delegate. Subjects must already possess the access rights to be delegated. Delegation policies are aimed at subjects delegating rights to servers or third-parties to perform actions on their behalf and are not meant to be the means by which security administrators would assign rights to subjects. A negative delegation policy identifies what delegations are forbidden. With a delegation policy, we need to specify the following information:

- The authorisation policy from which delegated rights are derived,
- **Grantors** – the subjects who can delegate these access rights
- **Grantees** – the objects to whom the access rights can be delegated

There are two types of delegation policy, positive and negative.

```
deleg_type = ('deleg+' | 'deleg-'),
             deleg_formal_call_decl,'{',{deleg_type_body}, '}' ;

deleg_formal_call_decl =
    ident_or_path, '(', ['auth+'], ident_or_path, ')',
                   '(', [formal_parameters], ')', [extends_type] ;

deleg_inst = deleg_inst_def | deleg_instantiation ;

deleg_inst_def = ('deleg+' | 'deleg-'), ident_or_path,
    '(', ['auth+'], ident_or_path, ')', '{', {deleg_type_body} ,'}';

deleg_instantiation = ('deleg+' | 'deleg-'), deleg_actual_call_decl;

deleg_actual_call_decl = ident_or_path, '=', ident_or_path,
    '(', ident_or_path, ')', '(', [actual_parameters], ')';

deleg_type_body =
    policy_elements                          |
    ('grantee', domain_scope_expr            |
     'action',  deleg_access_rights), [';']   ;

deleg_access_rights = neg_auth_actions;
```

4.5.1 Associated Authorisation

The syntax of a delegation policy type declaration has a different format from that of the other policy types. The authorisation and/or delegation policies involved in the delegation are specified separately before the list of other formal parameters to the policy preceded by the optional keyword `auth+`. The policies specified are those from which the access rights of the subject are derived.

4.5.2 Subjects, Targets and Grantees

The grantee entry allows the specification of the subject to which the policies are delegated. The subject of a delegation policy is what we call the grantor. It specifies who is authorised to delegate. The delegation policy allows the specification of a separate target to override the target of the authorisation policies. If a target is specified, then this should be a subset of the target in the associated authorisation policies. This allows the subject delegating access rights to restrict the targets to which the grantee can execute those access rights.

4.5.3 Delegated Access Rights

The delegated access rights must be a subset of those defined in the associated authorisation policy. An action being delegated may have an a filter which will be executed when the action is invoked by the grantee on the target.

4.5.4 Cascaded Delegation

Cascaded delegation is allowed provided that both the grantor and the grantee are in the grantee scope of the delegation policy. There is one other kind of "cascading" which can be specified by passing a delegation policy as a parameter to a delegation policy. In that case the grantee of a "cascaded" delegation might not be a subset of the grantee of the original delegation.

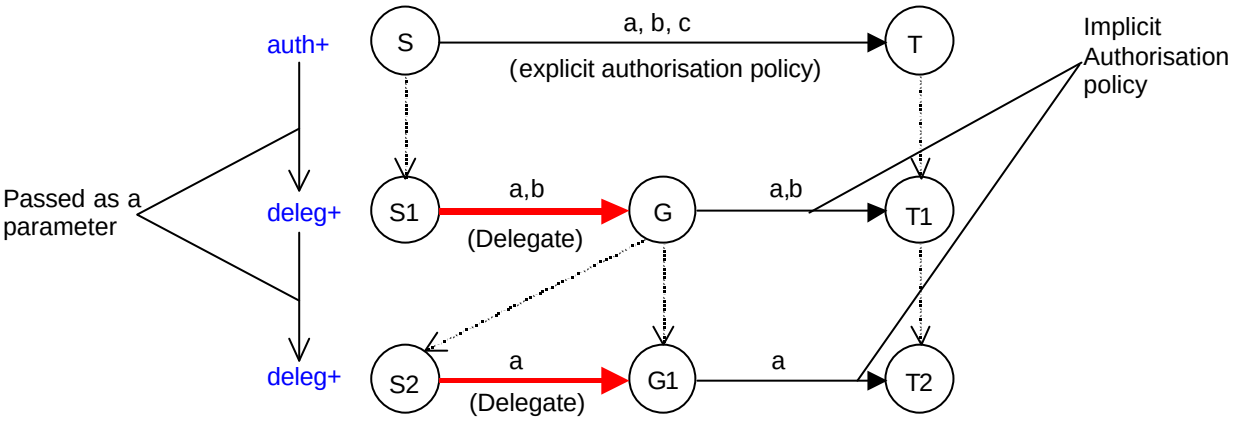

Figure 5. auth+ and deleg+ policies as parameters to Delegation Policies

In the above figure, the following relations are true: S1 ⊆ S, T1 ⊆ T, S2 ⊆ G, T2 ⊆ T1

Examples

In the following example, the `subject` (the grantor in the delegation policy), delegates only a subset of his/her access rights to the `grantee`. The delegation rights are derived from the associated authorisation policy.

```
type
    auth+ serviceManT (subject s, target t) {
          action t.resetSchedule, t.enable, t.disable
    } // serviceManT

    deleg+ sDelegT (serviceManT a)(subject grantor, grantee granteeD){
          action resetSchedule
    } // sDelegT

inst
    auth+ serviceMan = serviceManT(brManager, brServices)
    deleg+ sDeleg = sDelegT(serviceMan)(brEngineer+brSys, brServices)
    deleg+ sDeleg2 = sDelegT(sDeleg)(brEngineer, resetAgent)
```

5 COMPOSITE POLICIES

Composite policies	Keyword
Group	`group`
Role	`role`
Relationship	`rel`
Management Structure	`mstruct`

There is a need to group a set of related policy specifications within a syntactic scope with shared declarations in order to simplify the policy specification task for large distributed systems. This is a common concept in many programming environments and is the main motivation behind composite policy types in Ponder. At run-time, the set of policies defined in a composite policy, together with any constraints applying to the composite policy would be stored within a domain.

All composite-policies can include types and instance definitions as well as nested groups. However roles cannot include nested roles, relationships or management structures, and relationships cannot contain nested relationships or management structures. All composite-policies can be specified as types from which multiple instances can be created.

```
comp_pol_body = common_element_spec | (import_statement, [';']) ;

comp_nested_elem =    ('type', {comp_type_nested_elem}-) |
                      ('inst', {comp_inst_nested_elem}-) ;

comp_type_nested_elem = (group_type | policy_type | meta_type), [';'] ;

comp_inst_nested_elem = (group_inst | policy_inst | meta_inst), [';'] ;
```

5.1 Groups

This is a syntactic scope used to declare a set of policies and constraints which are grouped together as they have some semantic relationship and should be instantiated together. For example they may reference the same targets, relate to the same department or relate to a particular application. A group can contain any basic-policy or nested group specifications.

```
group_type = 'group', formal_call_decl, '(', {group_body}, ')' ;

group_inst =    ('group', ident_or_path, '(', {group_body}, ')') |
                ('group', instantiation) ;

group_body = comp_pol_body | comp_nested_elem ;
```

Examples

```
type
    group serviceFailT (subject s1, target t1, target t2, event e) {
          inst
                auth+ sReset {
                      subject s1; action resetSchedule; target t2;
                } // sReset
                oblig failReset {
                      subject s1
                      on e do resetSechedule()
                      target t1
                } // failReset
    } // serviceFailT

inst
    group brS_A = serviceFailT(brManager, brServices, failure)
    group brS_B = serviceFailT(opManager, deliveries, lateDelivery)
```

5.2 Roles

A role groups the policies specifying the duties and rights relating to a **position** within an organisation. A role is thus a particular type of group in which all policies have the same subject domain. A role can contain basic policies and groups of basic policies but not nested role, relationship or management structures.

The role instantiation declaration may specify an optional path name, which is to be used as the subject domain for the role. This assumes the subject domain has already been created in the domain hierarchy. If the subject domain is not specified then a domain with the name of the role instance is implicitly created and used as the subject domain i.e. the subject for policies within the role.

```
role_type = 'role', formal_call_decl, '(', {role_body}, ')' ;

role_inst =     (('role', ident_or_path, '(', {role_body}, ')') |
                 ('role', instantiation) ), [subject_domain]  ;

role_body = comp_pol_body  | comp_nested_elem ;

subject_domain = '@', ident_or_path ;
```

Examples

In the following example role `brManagerT`, extends the previously defined role `ManagerT` to provide specialisation. The `brManagerT` inherits all the definitions from `ManagerT`. The `@` following the instantiation of the role `branchManager`, indicates that the subject domain of the role is located at `/sd/brManagers`.

```
type
    role brManagerT (target brServices) extends ManagerT  {
          inst
            oblig review {
                on failure(service) do brServices.resetSchedule()
            } // review
    } // brManagerT

inst
    role branchManager = brManagerT(branchA/position/backupServices)
          @ /sd/brManagers
```

5.3 Relationships

Relationships specify policies pertaining to the relationship rather than the individual participating roles. Relationships can define roles, but cannot contain other relationships or management structures.

```
rel_type = 'rel', formal_call_decl, '(', {rel_body}, ')' ;

rel_inst =      ('rel', ident_or_path, '(', {rel_body}, ')') |
                ('rel', instantiation) ;

rel_body = comp_pol_body | rel_nested_elem | 'role', prefix_ident,[';'];

rel_nested_elem =     ('type', rel_type_nested_elem) |
                      ('inst', rel_inst_nested_elem) ;

rel_type_nested_elem = comp_type_nested_elem  | (role_type, [';']) ;

rel_inst_nested_elem = comp_inst_nested_elem  | (role_inst, [';']) ;
```

Examples

The following is an instance of a relationship between two roles that are "hard-coded" into the definition of the relationship. This relationship can only be used between the two declared roles. The two roles are already defined outside the relationship and are thus just referenced in the relationship using their full path name in the domain structure.

```
inst
    rel qSupervision {
          role /net/oam/netOperator
          role /net/edge/qConfig

          inst
            oblig report {
                subject /net/edge/qConfig.subject
                on time.at(1800); do report(q_info)
                target netOp
            } // report

            auth+ config {
                subject /net/oam/netOperator.subject
                action setStrategy; target qEdgeRtr
            } // config
    } // qSupervision
```

5.4 Management Structures

A management structure defines the configuration of roles and relationships in organisational units in terms of the required instances of the roles. For example it would be used to define a management structure (type) for creating branches in a bank or departments in a university. Management structures can include any nested composite-policy.

```
mstruct_type = 'mstruct', formal_call_decl, '(', {mstruct_body}, ')' ;

mstruct_inst =  ('mstruct', ident_or_path, '(', {mstruct_body}, ')') |
                ('mstruct', instantiation) ;

mstruct_body =  comp_pol_body | mstruct_nested_elem ;

mstruct_nested_elem = ('type', {mstruct_type_nested_elem}-) |
                      ('inst', {mstruct_inst_nested_elem}-) ;

mstruct_type_nested_elem =  comp_type_nested_elem  |
                          ((role_type | rel_type | mstruct_inst), [';']);

mstruct_inst_nested_elem =  comp_inst_nested_elem  |
                          ((role_inst | rel_inst | mstruct_inst), [';']);
```

Examples

In the following example a management structure instance is defined `oam/traffic`, which contains another management structure inside it (`qos`). The `oam/traffic` also contains the specification of two roles and two relationships. The second relationship `configAdmission` (which is specified in full), relates the `netOp` role instance, created within the outer management structure `oam/traffic`, with the `admControl` role, created within the `qos` management structure.

```
inst
    mstruct oam/traffic {
          inst
                role netOp {…}
                role qEdgeRtr {…}
                rel  qSupervision {…}
                mstruct qos {
```

```
                inst
                        role admControl {…}
                        role trShaping {…}
                        rel selectTraffic {…}
                } // qos

                rel configAdmission {
                        role netOp
                        role qos.admControl
                        inst
                                auth+ setClass {
                                        subject netOp; target admission
                                        action set(trClass, qos.admControl)
                                } // setClass
                } // configAdmission
        } // oam/traffic
```

5.5 Policy Type Specialisation

Ponder allows inheritance by specialisation for types; types can extend other types. When a type extends another type, it inherits all the attributes (policy elements) of the base type, and can add new ones.

The specification of the formal parameters in a type definition can be followed by an extends-clause to provide inheritance by specialisation. The type to be extended can be specified as a path indicating it's position in the domain structure. The syntax of the extends-clause can be the same as that of the `actual_call_declaration` (see section 3.4). The type that extends some other base type, can pass parameters to the base type with the extends clause in order to parameterise the base type.

```
extends_type = 'extends', (actual_call_decl | ident_or_path) ;
```

Examples

In the following example, the `specialised_nurseT` role type, extends (specialises) the specification of the role type `nurseT`.

```
type
    role nurseT (target t) {
        type
                oblig adminT(target t1) {
                        on t.temperature > 37
                        do administer(analgesics)
                        target t1
                } // adminT

        inst
                oblig admin1 = adminT(t)
                oblig drugsAdmin {
                        on administer_drugs
                        do update()
                        target /drugs_db
                } // drugsAdmin
    } // nurseT

    role specialised_nurseT (target t) extends nurseT(t) {
        inst
                oblig cat1_drugsAdmin {
                        on administer_Cat1_drugs
                        do update() -> check_availability()
                        target /drugs_db
                } // cat1_drugsAdmin
    } // specialised_nurseT
```

6 META-POLICIES

Meta-policies specify constraints, over a set of policies, on the permitted types of policies or their policy elements. Meta-policies can be defined within a composite-policy to apply to all policies within the scope of the composite policies. Meta-policies may also apply to all policies within a domain subtree. The Object Constraint Language (OCL) is used to specify meta-policies. The body of the meta policy (`meta_body`) specifies the constraint as a series of OCL constraints separated by semicolons. The last one of which must be a boolean constraint, that if `true` causes the action following the `raises`-clause to be executed. Note that the OCL expression can be named so that it can be passed to the constraint action as a parameter (see example).

```
meta_type =
    'meta', formal_call_decl, 'raises', action_call, '{', meta_body, '}';

meta_body = meta_expression, {';', meta_expression};

meta_expression = [ '[', ident, ']', '=' ], ocl_expression;

meta_inst =
    ('meta', ident_or_path, '{', meta_body, '}') |
    ('meta', instantiation) ;
```

Examples

The example meta-policy shown here, specifies an instance of the separation of duties principle. Two actions and a target type are passed as parameters to the meta-policy. Within its body, the meta-policy checks all pairs of policies in its scope, for possible conflicts. If there exists a pair of policies with common subjects, who have actions `act1` and `act2` respectively in their action entry, and whose target intersection is of the given `tarType`, then there is a conflict and the conflict action `conflictSepD(z)` is called. This action takes the set of pairs of policies resulting in conflict (the result of the OCL expression) as a parameter, so that it can act on them. In order to check the type of the target intersection we use the `oclIsKindOf` method defined in OCL.

```
type
    meta dutyConflictT(act1, act2, tarType) raises conflictSepD(z) {

          [z] = self.policies->select(pa, pb |
                pa.subject->interection(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(tarType)) ;

          z -> notEmpty
    } // dutyConflictT

inst
    meta dc = dutyConflict('execute', 'authorise', 'payment')
    meta bwDc = dutyConflict('addBandwidth', 'use', 'service')
    oblig notifyConflict {
          subject policyService
          on dutyConflict(z)
          do policyService.notify(manager)
    } // notifyConflict
```

7 CONSISTENCY RULES

The following are rules that must be true for a specification to be complete.

7.1 Basic Policies

Basic policies cannot contain other policies. Although they usually need an explicit subject an exception is when a basic policy is specified as part of a Role, in which case the subject domain of the Role is the implicit subject.

Authorisation policies

For both positive and negative authorisation policies, the specification of the following policy elements is required. An authorisation policy must contain the following policy elements:

- subject (except in roles)
- target
- action

Obligation policies

An obligation policy must contain the following policy elements:

- subject (except in roles)
- action
- event

Refrain policies

A refrain policy must contain the following policy elements:

- subject (except in roles)
- action

Delegation policies

One or more positive authorisation and/or delegation policies must always be associated with a delegation policy (both positive and negative).

The only required policy element for a delegation policy is the specification of a grantee. Subjects and targets, if not specified, default to the aggregated subjects and targets of the associated authorisation/delegation policies. If actions to be granted are not specified they default to those of the associated authorisation/delegation policies.

7.2 Composite Policies

Roles

When authorisation, obligation and refrain policies are specified within a role, their subject is the position domain of the role. In this case the subject is implicit.

A role must not contain other roles, relationships or management structures.

Relationships

A Relationship should not contain other relationships or management structures.

Groups

A Group should not contain roles, relationships or management structures.

8 FUTURE WORK

Future versions of Ponder will include improvements in the following areas:

Delegation Policies. We need to be able to specify delegation constraints in order to support restrictions on delegation, such as time constraints (e.g. maximum delegation period), maximum number of delegation hops, etc. The current version of the language does not include delegation constraints. This needs further study into the area of access control delegation.

Relationships. Interaction protocols are not included in the current version of Ponder. This will be an important addition to the language.

Meta-Policies. Meta policies are a very powerful feature. Experimentation with various application-specific constraints specified as meta policies is needed to reach a more definite specification. Meta policies may include a `when`-clause to restrict their applicability; an event to trigger them or possibly other policy elements. Concurrency constraints for groups of policies might be specified as meta-policies.

Inheritance. The inheritance mechanism for policy types currently does not allow overriding of policy elements. We are currently investigating a suitable inheritance model to support this feature in a future version of Ponder.

Concurrency Constraints. There is a need to specify concurrency constraints for groups of policies. Concurrency can only be specified for the individual actions of an obligation policy in the current version. In future versions we will consider the addition of concurrency constraints for policies specified within the various composite policies.

Selector Object. There may be a need for an application specific selector object which selects the objects in the subject or target domains to which an obligation policy applies. For example only one, possibly the least loaded, of the potential objects in the subject domain should perform the action specified in an obligation policy. The action may need to be applied to all or none objects in the target domain as an 'atomic action'. The implementation issues related to this selector have still to be fully worked out.

Library objects for various utility functions (e.g. Maths, String, Time, Timers) will be provided.

Policy Refinement. We are actively working on providing tool support for policy refinement from goals or service level agreements to implementable policies and on analysis of policies for conflicts etc (SecPol)

9 REFERENCES

Note: Imperial College papers are available from http://www-dse.doc.ic.ac.uk/policies

Ponder SableCC grammar is available from http://www-dse.doc.ic.ac.uk/policies/ponder.html

Lupu, E. C. and M. S. Sloman (1999b). "Conflicts in Policy Based Management Systems" IEEE Transactions of Software Engineering, Nov 1999

Lupu, E. C. (1998). A Role-Based Framework for Distributed Systems Management. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K., July 1998.

Lupu, E. C. and M. S. Sloman (1997b). "Towards a Role Based Framework for Distributed Systems Management." Journal of Network and Systems Management **5**(1): 5-30, Plenum Press Publishing, 1997.

Lupu, E. C. and M. S. Sloman (1997c). A Policy Based Role Object Model. In Proceedings of the 1st IEEE International Enterprise Distributed Object Computing Workshop (EDOC'97), Gold Coast, Queensland, Australia, pp. 36-47, October 1997.

Marriott, D. A. (1997). Policy Service for Distributed Systems. Ph.D. Thesis, Department of Computing, Imperial College, London, U. K., October 1997.

Marriott, D. A. and M. S. Sloman (1996). Implementation of a Management Agent for Interpreting Obligation Policy. In Proceedings of the 7th IFIP/IEEE International Workshop on Distributed Systems Operations Management (DSOM'96), L' Aquila, Italy, October 1996.

OMG Object Management Group (1999). "The Common Object Request Broker: Architecture and Specification, Revision 2.3.1.", October 1999.

Rational Rational Software Corporation (1997). "Object Constraint Language Specification, Version 1.1.", Available at http://www.rational.com/uml/, September 1997.

SecPol: Specification and Analysis of Security Policy for Distributed Systems, http://www-dse.doc.ic.ac.uk/projects/secpol/SecPol-overview.html

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

Sloman, M. S. (1994b). "Policy Driven Management for Distributed Systems." Journal of Network and Systems Management **2**(4): 333-360, Plenum Press Publishing, 1994.

10 FURTHER EXAMPLES

This section provides more complete examples.

A Ponder Specification

The example below demonstrates the structure of a Ponder specification. Note that type and instance definitions can be nested. Import and domain statements can be placed anywhere within the specification.

In this example a role type `helpDeskT` is defined for a cellular GSM network company. Suppose that the network is divided into regions and each region is further subdivided into branches. Each region has a database called EIR (Equipment Identity Database) for the equipment of the region. Each branch has a database called HLR (Home Location Register) for the subscribers to the network.

The `helpDeskT` role includes an obligation policy (`customer_complaints`) to handle customer complaints; a group `hlr_managementT` specifying policies that relate to the management of an HLR database for a branch; a group `billing_and_abnormal` that contains policies related to cases of unpaid bills, stolen equipment etc. The first group is created as a type and then instantiated for the various HLR databases corresponding to each branch.

The authorisation policies that authorise the access to the HLR and EIR databases are not specified directly within the role. They are instead specified as a group `HD_authorisationsT` outside the role. This could be the case if there is a need to reuse those authorisations in other roles or anywhere else within the policy specification. The role `helpDeskT` then imports the `HD_authorisationsT` group, and instantiates it for the different HLR and EIR databases to which it needs access.

```
domain /policies/groups/types

type
    group HD_authorisationsT (subject hd, HLR_type hlr, EIR_type eir) {
          inst
                auth+ HD_auth_HLR {
                      subject hd
                      target  hlr
                      action  add_new_customer(), update_record(),
                              traceHomeSubscriberInHLR();
                } // HD_auth_HLR

                auth+ HD_auth_EIR {
                      subject hd
                      target  eir
                      action  blacklistEquipment()
                } // HD_auth_EIR
    } // HD_authorisationsT

domain /tr/rr/rc/HD

type
    role helpDeskT(EIR_type eir) {

          import /policies/groups/types/HD_authorisationsT

          inst
                oblig customer_complaints {
                      on customer_complaint(complaint)
                      do /* import complaint */
                       subject.investigate_complaint(complaint)
                } // customer_complaints

          type
                group hlr_managementT(HLR_type hlr) {
                      inst
                            oblig record_update {
```

```
                              on new_service_subscription(x)
                              do updateRecord(x.customer, x.service)
                              target hlr
                        } // record_update

                        oblig consistency_loss {
                              on unrecognised_customer_in_HLR(imsi)
                              do subject.checkRecord(imsi)
                        } // consistency_loss
            } // hlr_managementT

      inst
            group hlr_managementBrA = hlr_managementT(hlr_branchA)
            group hlr_managementBrB = hlr_managementT(hlr_branchB)

            group billing_and_abnormal {
                  inst
                        oblig notify_subscriber {
                              on unpaid_bills(imsi)
                              do notifySubscriber(imsi)
                              target emailServer
                        } // notify_subscriber

                        oblig stolen_equipment {
                              on reported_stolen(imei)
                              do blackListEquipment(imei)
                              target eir
                        } // stolen_equipment
                  } // billing_and_abnormal

            group hlr_auth1 = HD_authorisationsT(this.pd,
                              hlr_branchA, eir)
            group hlr_auth2 = HD_authorisationsT(this.pd,
                               hlr_branchB, eir)
    }

domain roles/HelpDesk

inst
    role helpDeskRegionA = helpDeskT(eir_regionA) @ pd/HD/HD1
    role helpDeskRegionB = helpDeskT(eir_regionB) @ pd/HD/HD2
```

Filters

The following is a hypothetical class-diagram of the information stored in a departmental server.

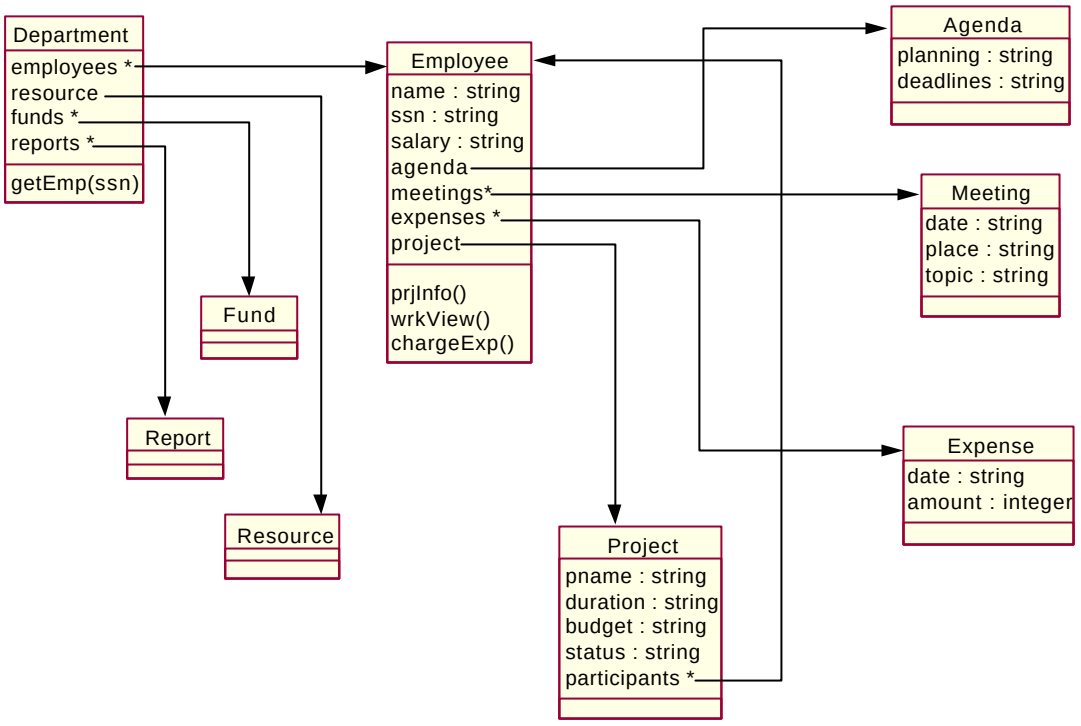

Figure 6. Departmental Information Class Diagram

The `getEmp(ssn)` method returns an Employee object given its `ssn`-number. Assume there is an authorisation policy authorising subjects to execute the method `getEmp(ssn)` on objects of type `Department` on the departmental file server. Depending on the subject of the authorisation, there is a filter that allows the subject to see only part of the information returned:

- The General Manager can see all of the information.
- The Departmental manager cannot see the agenda of the employee.
- Another fellow Employee cannot see the salary, his agenda and the budget of the projects to which the employee is assigned.
- A person outside the organisation can see only the name, project names and meeting topics of the employee.

Here are the authorisation policies to specify this.

```
inst
    auth+ GMgetEmployeeAuth {
          subject General_Manager
          target DeptFile_Server
          action getEmp(ssn)
    } // GMgetEmployeeAuth

    auth+ DMEmployeeAuth {
          subject Dept_Manager
          target DeptFile_Server
          action getEmp(ssn) {result = reject(result, agenda)}
    } // DMEmployeeAuth

    auth+ employeeAuth {
          const
                e = /employees
                other = /external

          subject {e} + {other}
          target DeptFile_Server

          action getEmp(ssn) if (subject = e) {
```

```
            result = reject(result, salary, agenda, projects.budget)
        } // getEmp

        action getEmp(ssn) if (subject <> e) {
            result = ext_select(result, name, project.pname,
                                meeting.topic)
        } // getEmp
    } // employeeAuth
```

Delegation

Consider the following hypothetical domain structure.

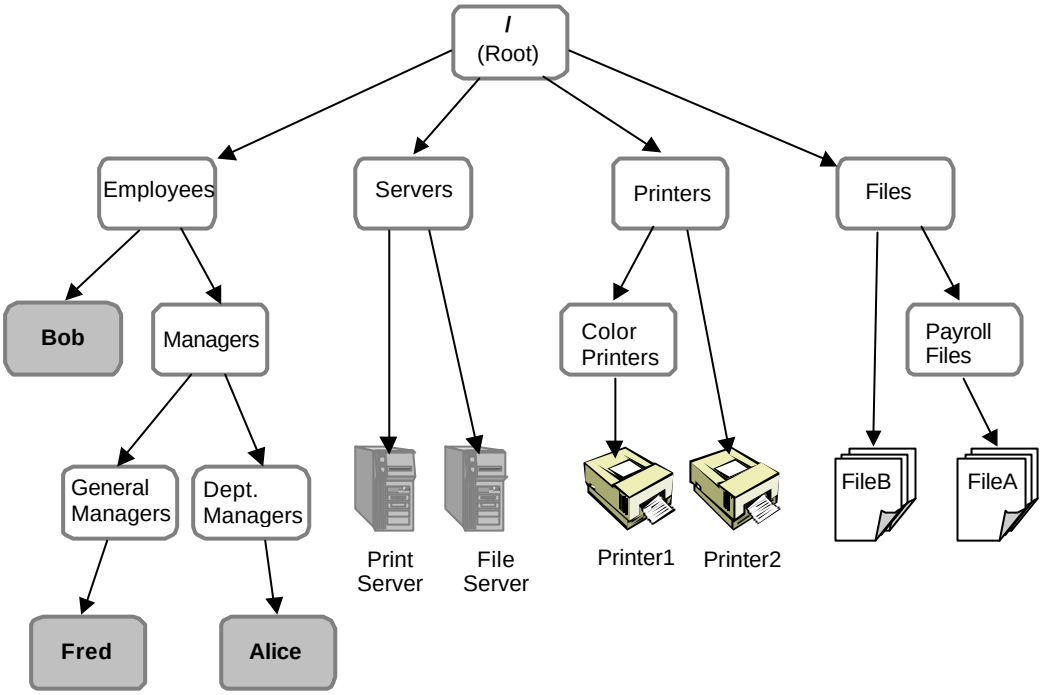

Figure 7. A hypothetical domain

Suppose that the following authorisation policies are in place:

```
type
    auth+ fileAccess (subject S, target files) {
        action read, write
    } // fileAccess

inst
    auth+ managerFileAccess =
        fileAccess(Employees/Managers, Files/PayrollFiles)

    auth+ employeeFileAccess = fileAccess(/Employees-Employees/Managers,
                    /Files-Files/PayrollFiles)

type
    auth+ printAccess (subject S, target printer) {
        action print
    } // printAccess

domain man = /Employees/Managers

inst
    auth+ GMprintAccess =
            printAccess(man/GeneralManagers, Printers/ColorPrinters)
```

```
auth+ employeePrintAccess =
            printAccess(/Employees, /Printers-Printers/ColorPrinters)

auth+ fileServerAccess {
      subject Employees
      target Servers/FileServer
      action all
} // fileServerAccess

auth+ printServerAccess {
      subject Employees
      target Servers/PrintServer
      action all
} // printServerAccess
```

The following delegation policy specifies that departmental managers are not allowed to delegate the access rights specified by the `managerFileAccess` policy to employees that are not managers.

```
inst
    deleg- invalidDeleg1 (managerFileAccess) {
          subject /Employees/Managers/DeptManagers
          grantee /Employees - /Employees/Managers
    } // invalidDeleg1
```

The following delegation policy specifies that general managers are not authorised to delegate the `write` access right specified by the `managerFileAccess` policy.

```
inst
    deleg- invalidDeleg2 (managerFileAccess) {
          subject /Employees/Managers/GeneralManagers
          grantee /Employees - /Employees/Managers
          action write
   } // invalidDeleg2
```

Finally, the last delegation policy specifies that general managers are authorised to delegate the `print` access right specified by the `GMprintAccess`, to departmental managers. Note that there is a constraint limiting the applicability of the delegation policy itself just like in any other type of policy. The example places a time constraint on the policy, but also another constraint which is a form of run-time delegation constraint. An external method `notCascading()` is called which returns `false` if the delegation action called at run-time is a cascaded delegation. This disallows cascaded delegation for this policy.

```
inst
    deleg+ colorPrintDeleg (GMprintAccess) {
          subject /Employees/Managers/GeneralManagers
          grantee /Employees/Managers/DeptManagers
          action print
          when time.between(1800, 0700) and notCascading()
    } // colorPrintDeleg
```

The following scenario (see figure 7) is based on the hypothetical domain structure of figure 6. The scenario is deliberately made more complicated than could have been in real situations just to demonstrate different aspects of the delegation policy. In order for the `FileServer` to be able to access the requested file, it must be delegated the access rights from the subject that requires the access to the file. The same is true for the `PrintServer`. In order for it to be able to print to a particular printer, it must be delegated the access right by the user requesting the print.

Now consider the following scenario. A general manager (`Fred`) wants to print a payroll file (`fileA`) on a color printer (`Printer1`). Fred first needs to delegate the access right to the `PrintServer` to print on `ColorPrinters`, the right to access the `FileServer` and request a read on payroll `FileA`, and the right to access payroll files. The `PrintServer` then needs to further delegate the right to read `PayrollFiles` to the `FileServer` in order for the file server to be able to read `FileA`.

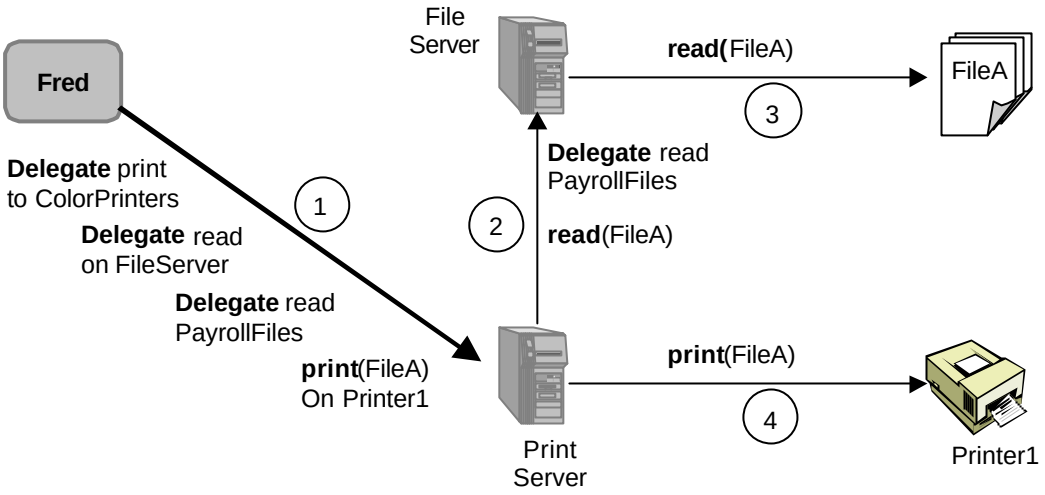

Figure 8. Delegation: Actions involved in printing a payroll file on a colour printer

The following delegation policies must then be in place in order for Fred to be able to print FileA on Printer1.

```
type
    deleg+ GMtoPrintServerT(auth+ authPol)(action actionToDelegate) {
          subject /Employees/Managers/GeneralManagers
          grantee /Servers/PrintServer
          action actionToDelegate
    } // GMtoPrintServerT

inst
    deleg+ GMtoPrint1 = GMtoPrintServerT(GMprintAccess)(print);

    deleg+ GMtoPrint2 = GMtoPrintServerT(fileServerAccess)(read);

    deleg+ GMtoPrint3 = GMtoPrintServerT(managerFileAccess)(read);

    deleg+ printStoFileS(GMtoPrint3) {
          subject /Servers/PrintServer
          grantee /Servers/FileServer
          action read
    } // printStoFileS
```

The first delegation policy (`GMtoPrint1`) states that a general manager can delegate the right to print to colour printers coming from the `GMprintAccess` authorisation policy. The second (`GMtoPrint2`), that it can call the action read on the file server, and the third (`GMtoPrint3`) that it can read payroll files.

The last delegation policy (`printStoFileS`) states that the print server can delegate the right to `read` payroll files to the file server. On the attempt to do so, the access control system would check that the print server has already been delegated this access right. The `GMtoPrint3` delegation policy only states that a general manager is authorised to delegate to the print server the referenced access right; it does not automatically mean that the print server has that right.

11 ANNOTATED BASE-CLASS DIAGRAM