Management Policy Service for Distributed Systems

Damian A. Marriott
dam@doc.ic.ac.uk

Morris S. Sloman
mss@doc.ic.ac.uk

Nicholas Yialelis
ny@doc.ic.ac.uk

Imperial College
Department of Computing
180 Queen's Gate, London SW7 2BZ, UK

Imperial College Research Report DoC 95/10

14 September 1995

Abstract

Interpreting policy in automated managers facilitates the dynamic change of behaviour of a distributed management system by simply changing policies. This paper describes a management policy notation which can be used to define both *authorisation policies* (what activities a manager is permitted to do) and *obligation policies* (the activities a manager must perform). Some example policy specifications are given to demonstrate the notation and the concepts involved. A graphical policy editor is described which permits high level abstract policies to be refined into lower level, implementable policies and maintains derivation and dependency relationships between the different policies. A policy service which stores policies is outlined and its integration within a domain service for grouping policies is explained. Outlines are given of implementations of automated managers for interpreting obligation policies and of an access control mechanism for enforcing authorisation policies.

Keywords: distributed systems management, management policy, security policy, policy notation, policy interpreters, event specification.

1 Introduction

Distributed systems management involves monitoring the activity of a system, making management decisions and performing control actions to modify the behaviour of the system (Sloman, 1994). Policies are one aspect of information which influences the behaviour of objects within the system. *Authorisation policies* define what a manager is permitted or not permitted to do in terms of operations on managed objects. *Obligation policies* define what a manager must or must not do and hence guide the decision making process; the manager has to interpret policies in order to achieve the overall objectives of the organisation.

Human managers are adept at interpreting both formal and informal policy specifications and, if necessary, resolving conflicts when making decisions. However the size and complexity of large distributed systems has resulted in a trend towards automating many aspects of management into distributed components. If the policies are coded into these components they become inflexible and their behaviour can only be altered by recoding. There is thus a

need to specify, represent and manipulate policy information independent from management components to enable dynamic change of policies and reuse of these components with different policies.

The specification of policies by human managers often begins at an high level of abstraction and it is through a process of *policy refinement* that these result, perhaps after several iterations, in policies that can be interpreted by the computer system. For example a policy 'the controller must maintain the reactor in a safe state' cannot be directly interpreted by an automated controller. It would have to be refined into actions to be performed by the controller when safety limits are reached, e.g. 'on temperature > 150 the controller must close valve 3'. There may be some policies that cannot be refined to such an extent, either because they fall beyond the scope of the computer system or technology for supporting them does not exist, so they can only be interpreted by humans. When a policy is refined, the resulting policies are said to be *children* of the original (*parent*) policy.

In this work policies are represented as objects which specify a relationship between subjects (managers) and targets (managed objects). Domains are used to group objects to which a policy applies. In general, a policy is specified in terms of a subject domain and target domain and the policy applies to all objects in the domains, so it is not necessary to specify separate policies for each object in a large system. Policy objects have attributes specifying the action to be performed or permitted, and constraints limiting the applicability of the policy. Policies themselves can be members of domains so that authorisation policies can be used to control access to the policy objects in a domain, e.g. to permit only authorised managers to define and modify policies.

A policy service provides a distributed database for storing policies and the refinement hierarchy, and a tool is provided to assist with the editing and manipulation of policies. If a high level policy is altered, it is then easy to trace the derived policies which may also need to be changed.

After refinement, *leaf-level* policies, i.e. policies which have no children, are ready to be distributed and enabled. Implementable obligation policies are interpreted by their subjects, automated managers, whereas high-level obligation policies may be sent to human managers for action. Leaf-level authorisation policies are translated into entries for access control lists associated with the target objects to which they apply. These access control entries are distributed to the security components which enforce access control in the systems on which their target objects reside. Authorisation policies that are still high-level are not implemented.

The policy service and tool have been implemented using ANSAware (APM, 1993) as the distributed platform and Tcl/Tk (Ousterhout, 1994) to provide the interpreted and graphical environments. The distributed aspects are being migrated to the CORBA-compliant Orbix platform (IONA, 1995), on which the obligation policy managers and access control mechanisms are being implemented.

2 Life of a policy

During its life a policy undergoes various changes of status, as shown in Figure 1. At its conception it acquires its *Dormant* status. It keeps this status, undergoing *editing* changes until its authors are satisfied that it is ready to be enacted, or until its death by *deletion*. A deleted policy does not actually exist anymore, hence the dotted box in the figure.

There are two steps involved in getting a policy up and running. The first is to *distribute*

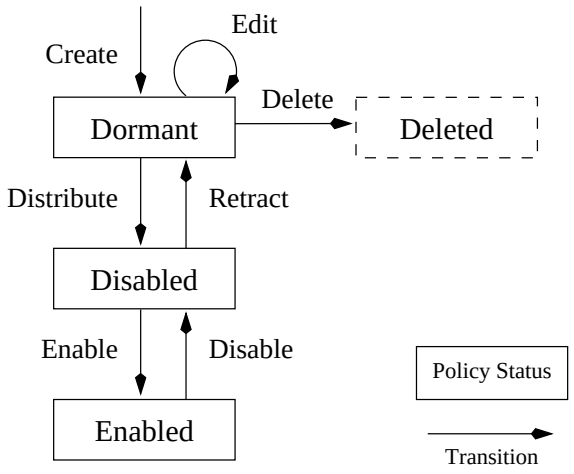

Figure 1: Status transitions in the life of a policy.

the policy to the object that will be interpreting it. Once this is done, the policy may be *Enabled*, i.e. take effect. This process is broken into these two steps to provide the added flexibility of being able to relatively quickly *enable* and *disable* policies at their implementors without needing to redistribute the policies, which could be both time-consuming and costly. Once *Disabled*, a policy may be removed from its interpreter by the *retract* operation. Note that a policy may not by deleted or altered until it has been disabled and retracted.

The policy service imposes an additional restriction on the transitions of policy status to prevent transient inconsistencies. A non-leaf-level policy can only be edited when all its descendants are in the dormant state. The policy service keeps track of this information for each policy. The policy editor provides a convenient means of editing policies that are currently distributed. The acts of distributing, enabling, disabling, retracting and deleting non-leaf-level policies cause the action to be propagated down the hierarchy.

3 Attributes of a policy

The general format of a policy is given below with optional attributes within brackets (the braces and semicolon are the main syntactic separators).

```
[description] identifier mode [trigger] subject '{' action '}' target
[constraint] [parent] [child] [xref] ';'
```

A more detailed description of the syntax is given in Appendix A. The attributes are now described.

3.1 Description

The description attribute of a policy provides an opportunity to express any general comments about a particular policy, such as its history, author and derivation. Other attributes of a policy (specifically, trigger, subject, action, target and constraint) may also be comments,

however any one of these being a comment destines a policy to be considered high-level and not able to be directly interpreted. Comments are C-style, as in '*/* this is a comment */*'.

3.2 Identifier

The policy service (see Section 6) provides a unique identifier as part of the object identification descriptor (*Oid*) for each new policy object that is created. When a policy is created a reference to it is placed in given a domain. The domain path name of this reference can be used by human managers to identify policies and is included when the editor writes policies into files (see Section 5). Note that the use of files here is simply a means for users, who are familiar with the notation syntax, to use a text editor for editing policies, however actual policy objects are stored by the policy service, which only uses Oids to identify policies, and does not store the path names.

3.3 Mode

The mode of a policy is given by either an '**O**', denoting an obligation policy, or an '**A**', denoting an authorisation policy, with either a '+' or a '−' appended, denoting a positive or negative policy respectively, e.g. '**A**+' denotes a mode of positive authorisation. Negative mode policies take precedence over positive ones, and by default actions are not authorised. Negative obligations should be read as 'obliged not to'. An obligation to do an action does not imply the authorisation to perform it; this must be specified as a separate authorisation policy.

Some conceptual examples of authorisation and obligation policies (not expressed in the notation) follow.

- information technology manager must protect the company's assets (*positive obligation*)
- primary archiver must backup department's files every night (*positive obligation*)
- secondary archiver must not backup department's files (*negative obligation*)
- all archivers can read all department's files (*positive authorisation*)
- ordinary users are not permitted to read backups (*negative authorisation*)

3.4 Trigger

The actions defined in an obligation policy may be triggered by an event, but authorisation policies (when enabled) are consulted whenever a relevant action is attempted and thus they do not require the specification of a trigger. An event trigger is specified using a subset of the GEM event expression notation (Mansouri-Samani and Sloman, 1995). The simple events able to be specified in the policy notation are primitive events and two types of time events. Primitive events may be either internally generated (within a manager), or notified by an external monitoring service component. Some examples follow.

- *primitive event*

```
on invalid_login                        /* internally generated event */
on /monit_serv/off_site/invalid_login   /* externally generated event
                                           from /monit_serv/off_site
                                           monitoring service component */
```

- *time point event*
  at [8:00] /* *at 8:00 (every morning)* */
  at [23:00 15/Mar/1995] /* *one specific point in time* */

- *time period event*
  every [20 ∗ min] /* *every twenty minutes* */
  every [3 ∗ day + 12 ∗ hour] /* *every three and half days* */

Only simple events are supported by the policy notation as a separate monitoring service is available to generate a composite event from multiple primitive events using composition operators. For example, a composite event could be defined in terms of a sequence 'e1; e2' i.e. e1 occurs before e2, or the non-occurrence of an event '(e1; e2) ! e3' i.e. e1 is followed by e2 with no intervening e3. Each composite event may also have an optional guard, which specifies a constraint to be met before an event will be triggered. An automated manager would register with the monitoring service to receive the events needed to trigger the obligation policies. Details of the monitoring service can be found in (Mansouri-Samani and Sloman, 1995).

Events may have attributes which can be accessed when the event is notified (see pm2_2 in the following section for an example). In a high-level policy both the trigger and action parts may be expressed in abstract forms as comments, for example:

```
arch0 O+ every /* night */ archiver {
      /* backup to safe store */
} *files ;
```

3.5 Subject

The subject of a policy specifies those objects which are to be obliged or authorised to perform the action specified. Users and automated managers are examples of typical subjects. The subject of an obligation policy determines where the policy is to be distributed, since it is to be interpreted by the subject.

The subject of policy is specified as a domain scope expressions (Becker, Sloman and Twidle, 1993) which are specified in terms of explicit objects and domains where membership is indeterminate at the time of policy specification. Scope expressions represent sets of objects, and allow union, difference and intersection operators over sets of domains and objects. An advantage of specifying policy scope in terms of domains is that objects can be added and removed from domains to which policies apply without having to change the policies.

The syntax for domain scope expressions is given as part of Appendix A. The interpretation of scope expressions is from left to right, and the symbols have the following semantics.

any : refers to any object
\+ : set union
− : set difference
^ : set intersection
∗ : if applied on a domain object returns a set that contains all
direct and indirect members of the domain and the domain object itself;
otherwise returns a set that contains the object itself — if an integer
constant is present, then the domain structure is traversed that many
levels down, which can be used to limit the search space

@ : if applied on a domain object returns a set that contains all direct members of the domain; otherwise returns the empty set

{} : returns a set that contains the object on which it is applied (note that this is implicit for objects without braces as a shorthand)

The following example authorisation policy uses full paths (which are not used in other examples for the sake of brevity), and is expressed in terms of members of Manager Position Domains, so that the policy applies to current members of these domains. It specifies that the system and security administrators are permitted to query and delete the computing department files except those in the Head of Department's (HOD) domain.

```
secpol1 A+
@/ic/computing/system_administrators +
@/ic/computing/security_administrators {
        query(); delete()
} @/ic/computing/files − @/ic/computing/HOD_files;
```

Each scope expression (and trigger) may have an optional label for enabling easy reference within the policy statement, of the form '*label*:*expression*'. For example, the following obligation on a printer manager obliges it to mail users if their job fails. In it 'pm' refers to the subject, 'ms' the target, and 'pf' is used to access the event's attributes.

```
pm2_2 O+ on pf:print_failed
pm:printer_manager {
        pm.mail_via(ms, pf.user, pf.message)
} ms:mail_server ;
```

3.6 Action

An action specifies what must be performed for obligations and what is permitted for authorisations. It consists of method invocations or a comment (for high-level policies), and may list different methods for different object types. For example, different types of storage devices may have different reset capabilities.

```
arch2 O+ on backup_complete archiver {
        "cdworm": stop(),
        "tape": rewind()
} @backup_devices ;
```

Multiple actions also can be specified separated by semicolons. For authorisation policies this means that all of the actions are authorised (or forbidden). For obligation policies, the subject must sequentially perform all the actions specified. There is no notational support for parallel actions, however, depending upon the underlying implementation, these may be achieved by specifying multiple obligation policies with identical triggers.

For authorisation policies the methods referenced in a policy are supplied by the target objects. However, for obligation policies, the methods may be sourced from the targets (the default scope) or subjects. These different sources are distinguished syntactically by labelling the source and prepending the action (or attribute in the case of constraints) with the label. For example, in the following obligation policy the subject (labelled with 'a') provides the backup action.

arch1 **O+ at** [23:00] a:archiver { a.perform_backup() } *files ;

The methods specified in actions may have parameters. For obligation policies, the parameters should be constant values or expressions which can be evaluated. For the particular case of obligation policies where the action is provided by the subject, the target is the default parameter when none is given (as in arch1 above), otherwise the target can be explicitly placed in the parameter list using a label (as in pm2_2 above). The following policy demonstrating the use of parameters obliges the archiver to mail the system administrator when problems occur with the backup procedure.

arch3 **O+ on** bf:backup_failure archiver {
    mail("system_administrator", bf.reason)
} mail_server ;

For authorisation policies the parameters are pseudo labels because they are simply identifiers which provide a handle for accessing the parameter values of an invoked operation. The values can then be used in constraint evaluation. For example, the following policy allows the archiver to use the mail method of the mail server which has two parameters, the first of which must be system_administrator. No constraint is placed on the message contents which is the second parameter.

arch4 **A+** archiver {
    mail(user, _)
} mail_server
when user == "system_administrator" ;

3.7 Target

The target or targets of a policy are objects on which the activities specified by the policy are performed, for example, files, printers, workstations, services and users. They are specified in the same way as subjects using domain scope expressions.

Authorisation policies are distributed to the targets' host security components which enforce them (see Section 8).

3.8 Constraint

The constraint attribute limits the application of a policy, e.g. to a particular time period, or making it valid after a particular date. A constraint takes the form of a predicate, similar in style to C expressions, which may refer to event attributes from the trigger part, system state, subject and target attributes, and parameters of actions and events. For example, the first constraint in the following uses the system attribute of local_time to specify a valid time period, and the second specifies a start date.

- **when** local_time >= [9:00] && local_time <= [17:00]
- **when** local_time > [1/Jun/1995]

Domain scope expressions may be used in constraints using the '**in**' operator, as in the following policy which authorises the security administrator to write policies subject to constraints on the mode and subject and target scopes.

```
secpol2 A+ security_administrator {
        set_policy(_, mode, _, subject, _, target, _)
} *maps/policies
when subject in *sec_admin_subjects &&
        target in *sec_admin_targets && mode == "A+";
```

It may also be desirable to allow limited method invocation within a constraint, with methods from the subject and possibly the target and system. These would be 'limited' in the sense that they should perform no action (or rather, effect no state change) since this would belong in the action, but could perform simple queries and data conversions. For example, a policy concerning a printer manager maintaining a current job queue might need to know the user associated with a given job number, which could be provided by querying the printer manger with a method invocation, as in the following example.

```
pm2_3 A+ u:@users {
        remove_queued_job(job)
} pm:printer_manager
when pm.owner(job) == u ;
```

3.9 Parents, children and cross references

In order to record the policy refinement hierarchy, policies contain references to their parent and children policies. In addition, a manual cross reference list of policies is kept for any policies not strictly related yet of possible interest, e.g. a cross reference could refer to a similar yet unrelated policy on which a policy has been modelled, or to the authorisation policy granting permission for an obligation policy's activity. These relationships are written with a keyword (either '**parent**', '**child**' or '**xref**') followed by a list of domain path names, as in the following example of arch3 from above, now shown specifying its parent and containing a cross reference to a related authorisation policy arch4.

```
arch3 O+ on bf:backup_failure archiver {
        mail("system_administrator", bf.reason)
} mail_server
parent arch0
xref arch4 ;
```

All reference information is stored in a dual fashion, so that a parent knows its children and similarly a child knows its parents. Note that the editing tool provides support for maintaining policy hierarchy and cross reference information.

4 Policy examples

In this section several scenarios with related policies are presented in order to clarify the notation and concepts.

4.1 Travel agent franchise

Consider the scenario of a travel agent (tr_agent) selling a franchise to an information network provider (net_prov), so that the provider can use the bought software to deal with sub-contractors, link products together, and provide advice and sell products to its customers.

A set of policies are specified which describe the authorities the network provider has once a sale has gone through. One such policy is given following.

```
/* net_prov has the right to place tr_agent software in its environment. It also
may make as many copies as it wishes. */
pol_1 A+ net_prov { install() ; copy() } tr_agent_software ;
```

Similarly, the travel agent has a set of obligations to uphold.

```
/* net_prov will receive support from tr_agent's marketing activity */
pol_2 O+ tr_agent { /* provide marketing support */ } net_prov ;
```

The travel agent makes stipulations about certain marketing data about which it wants to be informed.

```
/* net_prov will provide tr_agent with information about the users of the software
*/
pol_3 O+ net_prov { /* provide marketing reports */ } tr_agent
child pol_3_1 ;
pol_3_1 O+ at [23:00] net_prov {
      /* generate and dispatch report for purchase specific and browse
      (non-purchase) specific information */
} tr_agent
parent pol_3 ;
```

As remuneration for providing its software, the travel agent expects payment, consisting of a one-off fee, an annual fee and a percentage of the profits.

```
/* payment from net_prov to tr_agent */
pol_4 O+ on delivery net_prov { pay(one_off_fee) } tr_agent ;
pol_5 O+ at [15/4/−] net_prov {
      pay(annual_fee) ; pay(profit_percentage)
} tr_agent ;
```

A '−' in the date of a trigger indicates that this portion of the date is not considered in determining when to generate the event, e.g. in this case an event is generated every year on the fifteenth of April.

4.2 Printer manager

Consider a system where a system administrator has been assigned the task of providing and maintaining a printing service for users.

```
pm1 O+ @roles/system_administrator {
        /* provide and maintain a printing service for users */
} /* printing service */
child pm2_1, pm2_2, pm2_3 ;
```

The administrator sets up a manager to spool print requests and to handle queueing. All users except for a class of restricted users are given permission to submit print jobs.

```
pm2_1 A+ @users − @restricted_users {
        submit_print_job()
} printer_manager
parent pm1 ;
```

There is an obligation placed on the manager to inform users of failed jobs.

```
pm2_2 O+ on pf:print_failed
pm:printer_manager {
        pm.mail_via(ms, pf.user, pf.message)
} ms:mail_server ;
```

Authorisation is given for users to remove their own jobs.

```
pm2_3 A+ u:@users {
        remove_queued_job(job)
} pm:printer_manager
when pm.owner(job) == u
parent pm1 ;
```

4.3 Active badge system

Consider an active badge system in which users are equipped with badges which can be actively polled by sensors located in different rooms (Harter and Hopper, 1994). In addition, the badges may be signalled to give a small audio notification. A service keeping track of this information for each badge can be queried for the location information. Some policies in the context of this setup are given below.

The location information could be considered sensitive, so the following policy authorises a certain group of users to to access the location information for the user winston.

```
ab0 A+ @winston/friends {
        get_location()
} badgeman/winston ;
```

Location information can be put to good use by users, for instance, an enter_room event could be used to identify to which printer a user's output should be sent, and to lock and unlock terminals.

```
ab1 O+ on er:badgeman/winston/enter_room
badge_service {
        set_printer_room(er.user, er.room)
} printer_manager ;
```

```
ab2 O+ on lr:badgeman/winston/leave_room
winston/personal_manager {
        lock()
} winston/xserver
when lr.room != "101" ;

ab3 O+ on er:badgeman/winston/enter_room
winston/personal_manager {
        unlock()
} winston/xserver
when er.room == "101" ;
```

The badge could also be used as a paging device, setup as in the following example, to beep when mail is received and the user is away from their terminal.

```
ab4 O+ on new_mail
winston/mail_monitor {
        beep()
} w:badgeman/winston
when w.room != "101" ;
```

5 Policy editor

The tool used for viewing and editing policies, *pled*, consists of two different types of windows. The *policy window* is used to examine and edit the attributes of a policy, including relationships with other policies, i.e. parents, children and cross references. The *hierarchy window* is used to view and manipulate a set of policies, displaying either parent–child relationships or cross references. These windows are now further described.

5.1 Policy window

A policy window displays an individual policy in detail, listing its various attributes and displaying its relationships to other policies. A snapshot is shown in Figure 2. It supports the following functions.

- create a new policy, either with or without a relationship (child, sibling, parent or cross reference) to the current policy
- edit an existing policy stored in a policy server
- read and write a policy (of appropriate format) to and from a file, thus allowing policy editing in text editors for advanced users
- check for syntax errors in a policy
- perform operations on a policy (e.g. enable and disable)
- print a policy

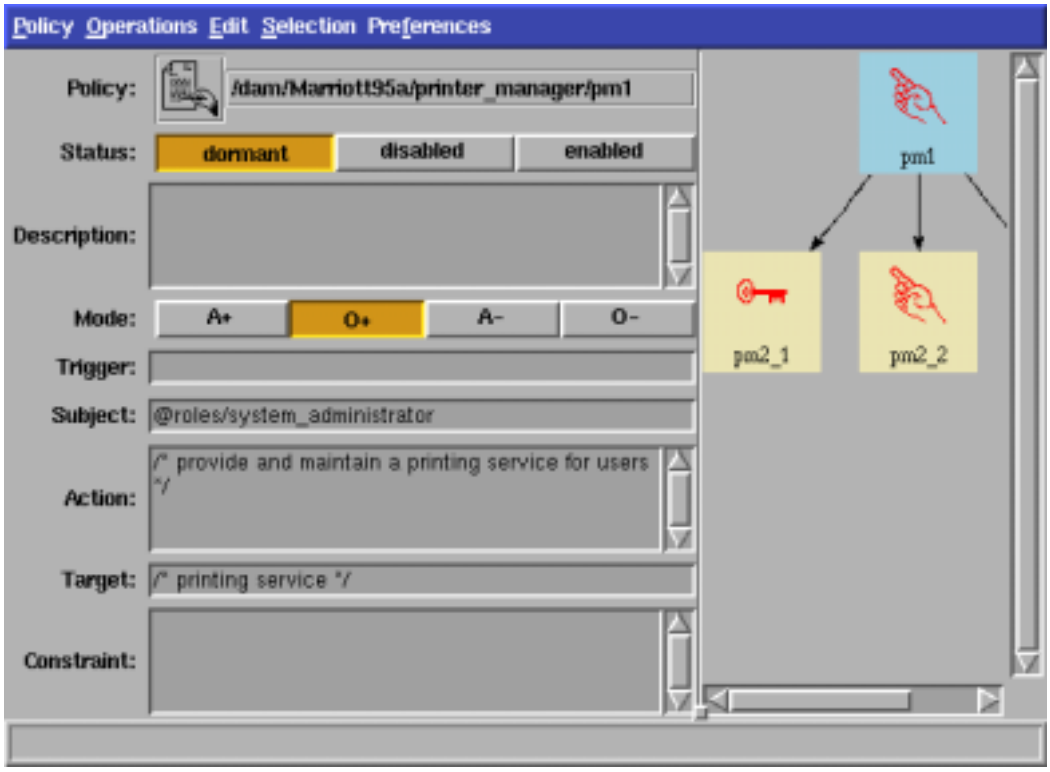

Figure 2: Snapshot of a policy window.

5.2 Hierarchy window

An hierarchy window is a tool for viewing and manipulating a number of policies. It displays any number of policies as icons and displays parent and child relationships between policies, by drawing arrowed lines between the icons, from parent to child. Alternatively, the cross references between the policies can be displayed. From the hierarchy window one can fire up a new policy window to examine a particular policy in detail, or drag a policy icon onto an existing policy window for a similar effect. Using the mouse one can select a group of policies and then perform operations on them, such as enable and disable, print, or write them to a file for manual editing and subsequent reading back.

A snapshot of an hierarchy window is shown in Figure 3. There is a single line at the bottom of the window which is used for displaying feedback to the user. As well as informing the user of operations being performed, when moving the mouse over the hierarchy, information is displayed relating to the object under the mouse cursor. In the figure this line is displaying as much as can fit of the policy pm1. Other information that can be displayed includes the full domain path of a policy, and alternative paths (since it may be a member of more than one domain).

There are various ways of adding new policies to an hierarchy window.

- policy — individual policies can be added
- domain — all policies within a given domain can be added

- server — all policies stored in a policy server can be added
- drag 'n' drop — icons of any of the above can be dragged into the window with the corresponding policies added

It would also be useful to be able to add all policies with a particular subject or target.

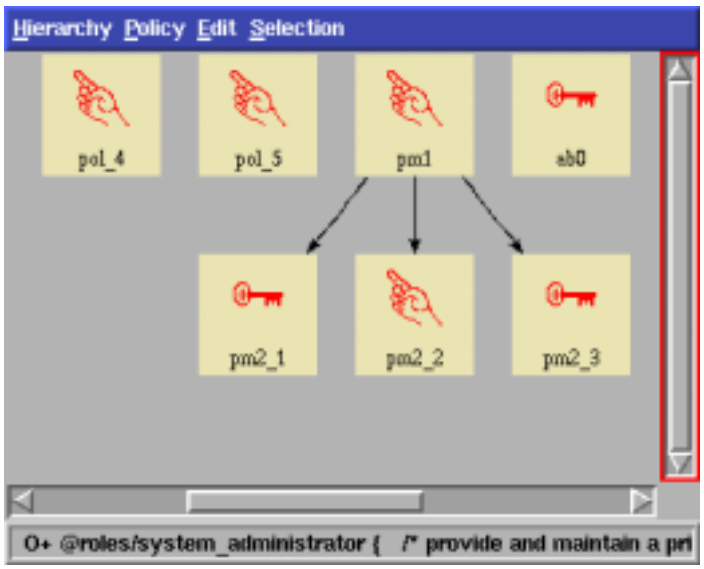

Figure 3: Snapshot of an hierarchy window.

6 Policy service

The policy service is provided by as set of management policy servers, *maps*, which provide a persistent data store for policy objects. It is currently implemented using ANSAware. Each maps process contains a server together with any number of policy objects.

Each new policy object that is created by the policy service is given a unique object identifier (*Oid*) which is used internally by the service although users may use the domain path name of the policy. The operations supported by the policy server itself are to create policies and to return the Oids of all policies stored in it. The actual policy objects support editing a policy, retrieving its status, and checking that a policy is syntactically and, to the extent possible, semantically valid. In addition, the operations of distribute, enable, disable, retract and delete act on a policy and alter its status as discussed in Section 2. When invoked on non-leaf-level policies these operations are propagated down the hierarchy. Note that deleting a policy may involve removing references from other policies and deleting any resulting orphans.

As mentioned previously, domains can be used to group policies for management purposes such as partitioning responsibility and access control. For example, the following policy specifies the restriction (mentioned in Section 2) on editing policies with descendants.

```
edit_restrict A+ @users { edit() } p:*policies
when p.status == "dormant" &&
     p.descendants_status == "all_dormant" ;
```

7 Obligation policy implementation

The implementation of obligation policies is distributed among automated managers which interpret policies. An obligation manager is currently being implemented in a custom Tcl/ANSA shell. Obligation policies on human subjects may be implemented through a suitable interface which advises users of the policies which apply to them although this has not been implemented in this system.

The diagram in Figure 4 shows how an automated manager interacts with the various components of a system to implement an obligation policy. The following steps are involved in setting up an obligation policy.

1. An authorised user creates, and edits a policy hierarchy in a *maps* within the policy service and then requests the *maps* to *distribute* and *enable* the leaf obligation policies.
2. A *maps* queries the *domain service* in order to determine the subjects specified in the subject domain scope expression of each policy, and propagates the policy to these objects (*automated managers*) thus completing the distribution.
3. A user then *enables* the policy via the *maps*.
4. The *managers* register with the *monitoring service* (or themselves for internal events), to receive the events specified in their triggers.
5. The *monitoring service* begins monitoring for the events registered, which may include monitoring *target objects*, the *system* and the *managers*.

Once a *manager* is notified of a relevant event, it checks the specified constraint is satisfied, possibly querying the *system* and *target objects* (which are determined by querying the *domain service*). The manager performs the specified action on each of the *target objects* which satisfy the constraint.

8 Authorisation policy implementation

A security architecture is being developed which aims at supporting domain–based access control and allowing the development of secure distributed applications on existing operating systems that do not support distributed security. The access control mechanism used to enforce authorisation policies is briefly described in this section. A more detailed description is given in (Yialelis and Sloman, 1995b).

Following the discussion in the previous sections, the *policy service* permits human users that have the necessary access privileges to specify authorisation policies. The *Access Control Agents*, which reside in every node within the system determine (in cooperation with the domain service) which policies apply to the targets in their nodes and hold copies of these policies.

When a subject intends to invoke operations on a remote target, a *secure channel* is established between these two objects. A secure channel is identified by a unique *channel identifier*

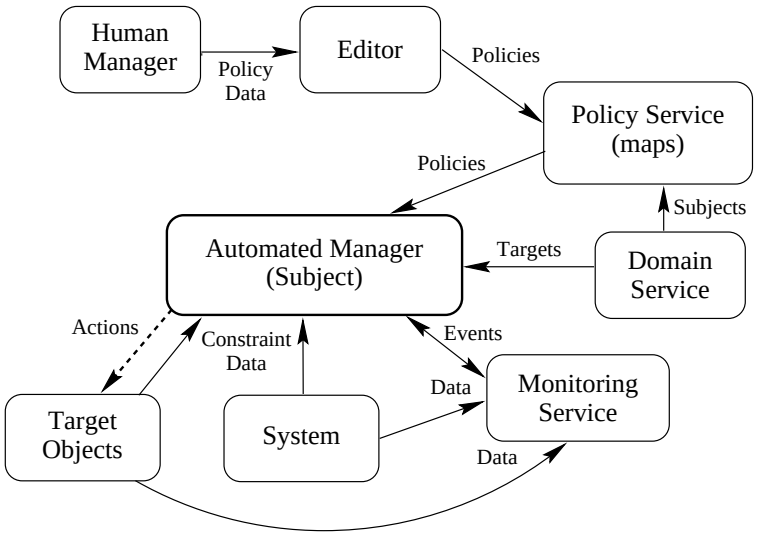

Figure 4: Interactions of an automated manager.

(chid) and is associated with cryptographic information (session key and cryptosystem used for secure communication), as well as access control information. In the framework of an established secure channel, the target Access Control Agent specifies the policies permitting the subject of the established channel to access the target. The list of these policies is referred to as *Enabled Policy List* (EPL) and it is given to the *Reference Monitor* (RM) located in the address space of the target (see Figure 5). A RM makes access control decisions for target objects in its address space using the EPL that applies to a particular subject–target pair (identified by the *chid* of the established secure channel). Note that the determination of the EPL only has to be done once for the lifetime of a secure channel.

The determination of the EPL is based upon the identity and domain membership of the subject which are authenticated by the *Authentication Agents* that reside in the nodes of target objects. These agents use a trusted *Authentication Service* which is based on symmetric cryptography for identity authentication and exchange of session keys, as well as the domain service for domain membership authentication. A detailed description of the authentication mechanism is given in (Yialelis and Sloman, 1995a). When a new secure channel has been established, the session key for that channel is given to the cryptographic facilities in the address spaces of the subject and target objects. So, further communication between these two objects does not involve the authentication agents. Code for the cryptographic and RM facilities are linked to the code of the objects.

Both the Authentication and the Access Control Agent are trusted to act on behalf of all objects in their node. Their use makes authentication and access control transparent to the application objects while it minimises the size of the security components that must be replicated among the application processes. A prototype of this access control scheme is currently being implemented using Orbix.

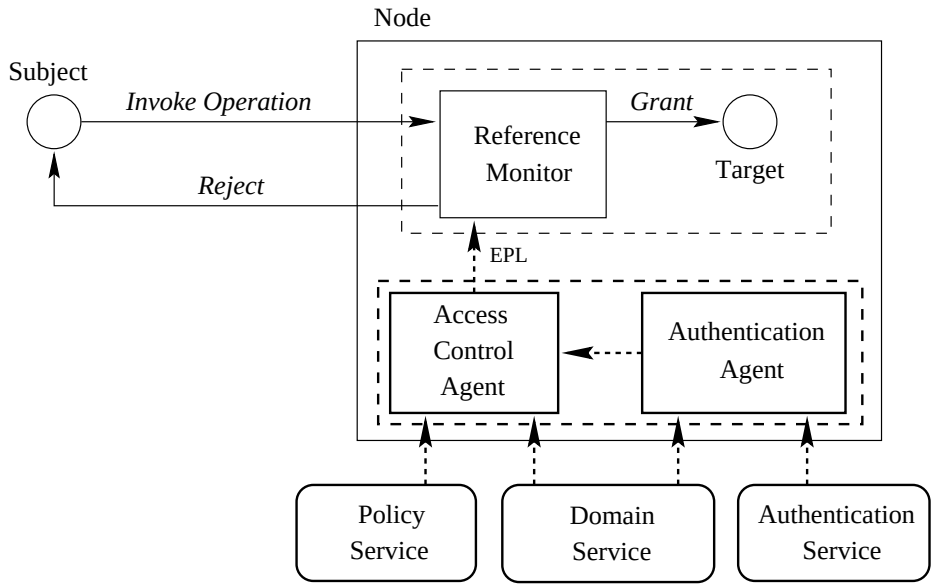

Figure 5: Access control overview.

9 Related work and conclusion

Putter, Bishop and Roos (1995) take a similar approach to our concepts of policies and domains, but their policy objects have two parts. A passive relationship object is very similar to our policy objects and defines a relationship between a manager and target domains. The second part is an active policy object which is a form of proxy manager which tries to achieve the goals specified in the passive policy on behalf of the manager. This active policy object would poll the managed objects in the domain and perform management operations on them. We would model this as a hierarchical management structure with one set of policies defining the manager to proxy relationship and another set of policies defining the role of the manager proxy with respect to the target domains. Wies (1995) also has active policy objects for enforcement and monitoring of policy. All these policy specifications, which define a manager or proxy's behaviour, look like management algorithms i.e. the policy is now encoded into proxy managers. We do not think that the policy service should be used as the means of distributing the management function. Meyer and Popien (1994) have a notation for specifying manager behaviour as policies, but it looks like a manager programming language. Policies need to be sufficiently abstract to be interpreted by managers or reference monitors so that they can be changed dynamically.

Wies (1995) has a template for specifying policies which is a superset of our policy objects. It includes our policy attributes (mode, trigger, subjects, targets, activity, parent reference, status) plus author, management functionality, management scenario, services affected by the policy, life-time of the policy and notifications emitted due to failures etc.. These templates are transformed into management scripts which generate active objects that can monitor the system in order to react to changes. He also uses a similar life-cycle model to ours.

The approach adopted by Koch, Kramer and Rohde (1995) is conceptually similar to ours but they use a rule-based software development environment called Marvel to implement

their policy driven management system. Their rules have predicates and are similar to our obligation policies, but they can specify an alternative list of assertions depending on the outcome of an activity. They do not support generalised authorisation policies, but they can specify in the precondition of a rule that, for instance, user authentication is required. The system has been used for centralised management in a distributed environment but it not clear that Marvel supports distributed management.

The ESPRIT IDSM project use the domain and policy concepts described in this paper, but their notation is based on ISO's Guidelines for the Definition of Managed Objects (GDMO) which is rather verbose. Their policies are implemented within OSI network management platforms. The only form of obligation policies which they have implemented are reporting policies which are translated into OSI Event Forwarding Descriptors (Alpers and Plansky, 1995). Both Bull and Siemens will be including domain and policy services within future releases of their network management platforms.

Jonscher's (1992) work on modelling access behaviour of database users also has some similarities to our approach. His access rights compare to our authorisation policies and his normative rights have some similarity to our obligation policy although he models both duties and freedoms (liberties) to do actions. Our system does not deal with freedom policies which might be used to permit human managers to override obligation policies, however we are not convinced of the need for liberties in management policy, and in particular automated management. Much of the work on active databases (Gehani, Jagadish and Shmueli, 1992) has triggered actions which are similar to our obligation policies but these databases are not distributed, and the triggered actions are defined statically and cannot be changed without re-compilation.

Lomita is a rule based language for programming the management layer in the Meta system (Marzullo, Cooper, Wood and Birman, 1991). Lomita rules are of the form 'on condition do action', which is also similar to Jonscher's triggered actions, but there is no explicit subject or target — they are defined implicitly.

This paper has presented a notation which can be used to express management policies for distributed systems as relationships between object scopes, expressed in terms of domains. These policies are at such a level that they can be interpreted by managers, thus providing the flexibility of being able to alter policies without having to re-build management components. The parent–child relationships between policies resulting from the process of policy refinement, and the more general cross reference relationships, can be expressed in the notation and the policy tools assist in the creation and visualisation of the refinement hierarchy. Two modes of policies have been identified, authorisation and obligation, and are supported by the one notation. The implementation support for these, which is quite different in each case, has been described.

10 Acknowledgements

We gratefully acknowledge the support of DSTO, Australia for a PhD cadetship. We also acknowledge support of the Commission of the European Union for ESPRIT Projects SysMan (7026) and IDSM (6311). In addition we acknowledge the support of our colleagues in the Distributed Software Engineering Section at Imperial College for comments on the concepts described in this paper. We are grateful to BT for their permission to use their TravelCo scenario as examples of management policies.

References

Alpers, B. and Plansky, H. (1995). Concepts and application of policy-based management, *in* A. S. Sethi, Y. Raynaud and F. Faure-Vincent (eds), *Integrated Network Management IV*, Chapman & Hill, London, pp. 57–68.

APM (1993). *ANSAware 4.1 Documentation Set*.

Becker, K., Sloman, M. and Twidle, K. (eds) (1993). *Domain and Policy Service Specification*, number S-SI-07-I-2-R. IDSM Deliverable D6, SysMan Deliverable MA2V2.

Gehani, N., Jagadish, H. and Shmueli, O. (1992). Composite event specification in active databases: Model and implementation, *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, Canada.

Harter, A. and Hopper, A. (1994). A distributed location system for the active office, *IEEE Network* **8**: 62–70.

IONA (1995). *Orbix Programmer's Guide*, 1.3 edn.

Jonscher, D. (1992). Extending access control with duties: Realized by active mechanisms, *IFIP WG 11.3 Sixth Working Conference on Database Security*, IFIP WG 11.3.

Koch, T., Kramer, B. and Rohde, G. (1995). On a rule based management architecture, *Second International Workshop on Services in Distributed and Netowrked Environments*, IEEE Computer Society Press, pp. 68–75.

Mansouri-Samani, M. and Sloman, M. (1995). GEM a generalised event monitoring language for distributed systems, *Research Report Doc 95/8*, Imperial College.

Marzullo, K., Cooper, R., Wood, M. and Birman, K. (1991). Tools for distributed application management, *IEEE Computer* **24**(8): 42–51.

Meyer, B. and Popien, C. (1994). Defining policies for performance management in open distributed systems, *Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*. DSOM '94.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*, Addison-Wesley.

Putter, P., Bishop, J. and Roos, J. (1995). Towards policy driven systems management, *in* A. S. Sethi, Y. Raynaud and F. Faure-Vincent (eds), *Integrated Network Management IV*, Chapman & Hall, London, pp. 69–80.

Sloman, M. (1994). Policy driven management for distributed systems, *Journal of Network and Systems Management* **2**(4).

Wies, R. (1995). Using a classification of management policies for policy specification and transformation, *in* A. S. Sethi, Y. Raynaud and F. Faure-Vincent (eds), *Integrated Network Management IV*, Chapman & Hall, London, pp. 44–56.

Yialelis, N. and Sloman, M. (1995a). An authentication service supporting domain based access control policies, *Research Report DoC 95/13*, Imperial College.

Yialelis, N. and Sloman, M. (1995b). A security framework supporting domain based access control in distributed system, *Research Report DoC 95/14*, Imperial College.

A EBNF of policy notation

In the EBNF of the policy notation which follows, the lexical tokens are denoted by single quotes, e.g. `'+'`, or capital letters, e.g. `COMMENT`, optionality is denoted by brackets, e.g. `[description]`, and repetition (zero or more occurrences) is denoted by braces, e.g. `{',' object}`.

```
// *** policy ***

policy:
        [description] identifier mode [trigger] subject '{' action '}' target
        [constraint] [parent] [child] [xref] ';'
;

// *** description ***

description:
          COMMENT
;

// *** identifier ***

identifier:
          IDENT
;

// *** mode ***

mode:
          'A+' | 'A-' | 'O+' | 'O-'
;

// *** trigger ***

trigger:                        // always empty for authorisation policies
          'on' COMMENT
        | 'at' COMMENT
        | 'every' COMMENT
        | event_expression
;

event_expression:
          'on' event_id_label
        | 'at' time_point_const
```

```
        | ‘every’ ‘[’ time_period_expr ‘]’
;

event_id_label:
          [ event_label ‘:’ ] event_id
;

event_label:
          IDENT
;

event_id:
          object
;

time_point_const:
          ‘[’ time_point ‘]’
;

time_point:
          time_of_day
        | weekday
        | date
        | time_of_day date
        | time_of_day weekday
;

time_of_day:
          INTconst ‘:’ INTconst [‘:’ INTconst]
        | INTconst ‘:’ INTconst ‘:’ DOUBLEconst
;
                // 0 <= n1 <= 23, 0 <= n2 <= 59, 0 <= n3 < 60

weekday:
          ‘Sun’ | ‘Mon’ | ‘Tue’ | ‘Wed’ | ‘Thu’ | ‘Fri’ | ‘Sat’
;

date:
          day_or_blank ‘/’ month_or_blank ‘/’ year_or_blank
;

blank:
        ‘_’
;

day_or_blank:
          blank | INTconst
```

```
;

month_or_blank:
          blank | month
;

month:
          INTconst | monthname
;

monthname:
          ‘Jan’ | ‘Feb’ | ‘Mar’ | ‘Apr’ | ‘May’ | ‘Jun’
        | ‘Jul’ | ‘Aug’ | ‘Sep’ | ‘Oct’ | ‘Nov’ | ‘Dec’
;

year_or_blank:
          blank | INTconst
;

time_period_expr:
          period
        | INTconst ‘*’ period
        | time_period_expr ‘+’ time_period_expr
                        // e.g., (2*hour)+(15*sec)
        | ‘(’ time_period_expr ‘)’
;

period:
          ‘msec’ | ‘sec’ | ‘min’ | ‘hour’ | ‘day’ | ‘week’ | ‘month’ | ‘year’
;

// *** subject ***

subject:
          COMMENT
        | [dse_label ‘:’] domain_scope_expression
;

dse_label:
          IDENT
;

domain_scope_expression:
          ‘any’
        | [‘{’] object [‘}’]
        | ‘*’ [INTconst] object
        | ‘@’ object
```

```
        | domain_scope_expression '+' domain_scope_expression
        | domain_scope_expression '-' domain_scope_expression
        | domain_scope_expression '^' domain_scope_expression
        | '(' domain_scope_expression ')'
;


object:
          OBJECT_PATH
        | IDENT
;

object_list:
          object {',' object}
;

// *** action ***

action:
          COMMENT
        | method_call {';' method_call}
;

method_call:
          object_method_call
        | switch_action_element {',' switch_action_element}
;

object_method_call:
          [dse_label '.'] method_name '(' [parameter_list] ')'
;

method_name:
        IDENT
;

switch_action_element:
        switch_selector ':' object_method_call
;

switch_selector:
          constant
        | 'default'
;

// *** target ***
```

```
target:
          COMMENT
        | [dse_label ':'] domain_scope_expression
;

// *** constraint ***

constraint:
          'when' COMMENT
        | 'when' logical_expr
;

logical_expr:
          logical_expr '||' logical_expr
        | logical_expr '&&' logical_expr
        | '!' logical_expr
        | expression
        | '(' logical_expr ')'
;

expression:
          expression '+' expression
        | expression '-' expression
        | expression '/' expression
        | expression '%' expression
        | expression '*' expression
        | expression '==' expression
        | expression '<' expression
        | expression '>' expression
        | expression '>=' expression
        | expression '<=' expression
        | expression '!=' expression
        | '-' expression
        | '+' expression
        | object_attribute
        | object_function_call  // must return some value - no
                                // side-effects
        | system_attribute
        | constant
        | dse_label 'in' '{' domain_scope_expression '}'
;

constant:
          INTconst
        | DOUBLEconst
        | STRINGliteral
        | CHARconst
```

```
        | time_point_const
;

object_attribute:
          [dse_label '.'] attribute_name
;

attribute_name:
          IDENT
;

object_function_call:
          [dse_label '.'] function_name '(' [parameter_list] ')'
;

function_name:
          IDENT
;

parameter_list:
          logical_expr {',' logical_expr}
;

system_attribute:
          'local_time' | 'local_hostname'
;

// *** parent ***

parent:
          'parent' object_list
;

// *** child ***

child:
          'child' object_list
;

// *** xref ***

xref:
          'xref' object_list
;
```