Distributed Systems Engineering Journal, Vol. 1, No. 1 August 1993, pp37-47

A previous version of this paper was presented at OpenForum 92 Technical Conference, Utrecht, Nov. 1992, pp. 83-100

User and Mechanism Views of Distributed Systems Management

Jonathan D. Moffett

Department of Computer Science
University of York
Heslington
York YO1 5DD
Email: jdm@uk.ac.york.minster

&

Morris S. Sloman

Imperial College of Science Technology and Medicine
Department of Computing,
180 Queen's Gate,
London SW7 2BZ
Email: mss@uk.ac.ic.doc

revised 11 June 1993

ABSTRACT
This paper discusses the concepts developed within the DOMINO project on Domain Management for Open Systems, and how these concepts are implemented. Domains are a means of grouping objects, distinct from the management policies which are specified in terms of domains. Domains and policies are discussed from the viewpoint of both the manager and the underlying mechanisms which implement them. The emphasis of the user view is on conceptual clarity and the emphasis of the mechanism view is on efficient implementation in distributed systems. Both views need to be implemented, although in some cases there may be a direct correspondence between the two views. The paper argues that keeping managers independent from the domain of objects they manage gives flexibility and simplifies both the user and mechanism views of system management.
Keywords

Management, domains, policy, access control, authorisation.

F.E Smith, the famous advocate, once saw two housewives haranguing each other from upstairs windows across a street. "They will never agree." he observed, "They are arguing from different premises."

1 INTRODUCTION

There is not yet general agreement on the concepts which are needed for the automation of Distributed System Management (DSM). This paper presents a set of concepts for specifying management policy as defined and implemented in the Domino project. Discussions with other research groups and within standard organisations has indicated confusion between the user interface view of management concepts and the underlying implementation mechanisms. Both views are needed to allow users to express their management requirements while enabling efficient implementation of the mechanisms.

The Domino project on Domain Management in Open Distributed System was a collaborative project funded by the U.K. Science and Engineering Research Council and the Department of Trade and Industry. It involved Imperial College, SEMA Group and British Petroleum. This project was concerned with managing very large distributed processing systems which typically consist of multiple interconnected networks and span the computer systems belonging to a number of different organisations. They cannot be managed from one central point, but management has to be achieved by negotiation between independent managers who wish to cooperate while retaining their autonomy.

We identified two separate generic activities of managers in systems:

- Performing **management operations** for monitoring or controlling the behaviour of objects related to a management function such as configuration or accounting.
- Creating, interpreting and monitoring **policies**. Policies are different from management operations. A management operation is an instantaneous activity, whereas a policy is intended as a persistent means of influencing operations. Large scale systems may have millions of objects so it is impractical to specify policies for individual objects. Policies need to be specified and applied to a set of objects as discussed in more detail below.

It was the need to be able to group objects in order to specify a common policy for them which led to the introduction of **management domains**[1] as a key concept in the Domino project. A domain is a collection of managed objects which have been explicitly grouped together for management purposes. Since domains are themselves objects, they may be members of other domains. A domain is also used as a means of structuring the namespace for objects.

---

[1] Abbreviated to *domain* in this paper

The concept of grouping objects should not be confused with that of encapsulation. Hierarchical composition can be used to construct a composite object from several primitive or other composite objects [Magee 1993]. The composite object is viewed as a single object for the purposes of invoking operations and the interface of the composite object hides its internal structure from the user. The component objects are then said to be encapsulated. Encapsulation is an essential concept for coping with the complexity of distributed systems and for building management systems. Although it is sometimes useful to apply a management operation to a set of objects, in many cases the managers have to perform management operations on the individual objects, which encapsulation prevents. Domains provide grouping but not encapsulation.

Section 2 of this paper distinguishes between the user and mechanism views of management, and justifies the need for the coexistence these views in a system. The views are related to the domain concept in section 3 and to policies in section 4. Section 5 shows how domains can represent users and manger positions. Section 6 discusses some of the related work on domains, and section 7 relates our approach to Open Distributed Processing (ODP) viewpoints.

2 USER AND MECHANISM VIEWS

One aspect of DSM which has taken some time to emerge is the importance of distinguishing between a manager's view of an enterprise which is often reflected in the human interface to the management system and the underlying mechanisms used within the system for implementing this view. The following examples will help to illustrate this.

a) Object based approach

The emergence of graphical interface technology has made the object based approach to human interfaces very popular. Objects are represented by icons which can be selected by means of a mouse, and the operations applying to an object can be selected from a menu for invocation on the object. Different icons are used to represent object types, and instances are distinguished by means of names. This approach is supported by object oriented environments which provide the means to define object types, create/delete instances of objects and for objects to invoke operations on each other. The manager's view is thus one of obtaining information on the objects they are managing and performing management operations on them to change their behaviour.

In some cases this view may not be able to map onto the underlying mechanisms. The object being managed may be a hardware component which does not support an object invocation mechanism compatible with that of the manager object. It is then necessary to "front" the managed resource with an adapter which provides a representation of the resource which is compatible with that of the manager. The approach of a managed object separate from the resource object is part of the OSI Management model [OSI 10040] and has been built into both their user and mechanism view even though it is inappropriate for software objects, which can be managed directly. Another justification given for managed objects being independent from the resources they represent is that the managed object can provide an abstraction of the resource for management purposes and hide the resource's normal functionality. Our approach (figure 2.1) simply assumes a managed object may have

multiple interfaces such as a management interface which defines the management operations it supports, an interface for interaction with a file service, plus other interfaces to support application-dependent functionality.

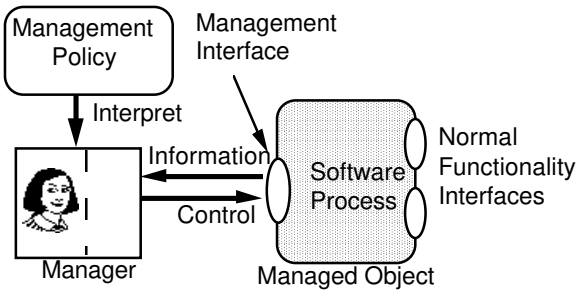


Figure 2.1 Domino Management Interactions

There is another example of low level mechanisms which confuses the higher level object interaction view in the OSI Management Model. The concept of a local agent in a computer is a common means of implementing remote invocations; the agent receives an invocation message from a remote manager and then performs the invocation on a local object. Again this has been built into the OSI management model rather than being considered a transparent implementation mechanism. If the OSI model is represented directly in an object based implementation a manager would invoke an operation on a remote manager agent, which would invoke an operation on a local managed object, which would finally invoke an operation on the resource being managed, rather than the manager directly invoking operations on a managed resource object. (See Figure 2.2)

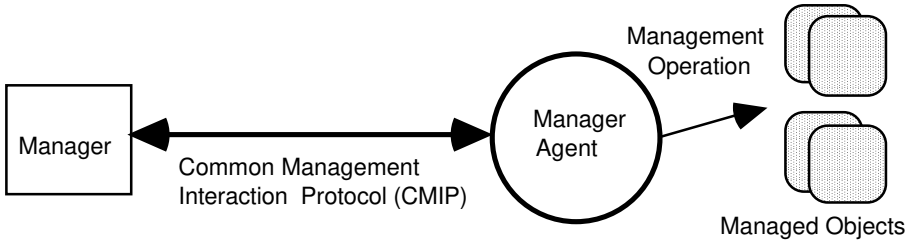


Figure 2.2 OSI Management Interactions

b) Management domains

Typical graphical window interfaces to computer systems provide the concept of a directory or folder to contain files or programs. The directory is a naming context so that object names need only be unique within a directory. The human manager needs a similar view for grouping objects in order to apply management policies or for convenience. A domain provides this means of grouping objects for management purposes. From the manager's viewpoint, domains contain named objects, but it is necessary to permit objects to be members of multiple domains to reflect overlapping responsibility or different types of management responsibility for the same object. For example the manager responsible for the security of an object may be different from the manager responsible for maintenance of that object.

The underlying implementation mechanism for domains has no concept of containment of the object itself. Instead the domain holds a reference to an object and provides a mapping from a text name string to an internal object identifier and address (see section 3.3). Users refer to objects by names or using a pointing device, while the system refers to them by object identifiers.

c) Access rules

Access rules specify discretionary access control policy in terms of the set of operations which any of a set of users is authorised to perform on any of a set of target objects. This also is a concept which users find intuitively easy to deal with, but does not implement efficiently. It would involve unreasonably long searches to evaluate access requests if the system had to search through all access rule objects in order to decide whether to allow the request. Therefore, for the Domino project, we are implementing the access rules by means of Access Control Lists (ACLs) attached to target object domains. The user sees and makes policy in terms of access rule objects, but the system maps these access rules onto ACLs, which it uses as the mechanism to interpret policy when it receives an access request. See section 4.

d) Users

Although users of a computer system may be human, managers manipulate user objects representing registered users within the computer system. These user objects define the resources allocated and the services accessible to the user. There are substantial advantages to be gained from a mechanism in which user objects are implemented as a special kind of domain, which we call a **user representation domain** (URD). The URD represents the persistent aspects of a user from one logon to the next and is analogous to a home directory in a Unix system. When the user logs on, process objects which temporarily represent the user (c.f. a Shell process) become members of the URD and gain the privileges and authority which belong to that user. When the user logs off, the processes cease to be members and no longer have the user's privileges. In this case the user interface view of user objects does not use the concept of a domain at all, but the system uses the mechanism view of a domain.

We believe that it is important to maintain both a distinction and a relationship between the user interface and mechanism views of management concepts. As with any system, the user requirements must drive the mechanism, but it must be subject to the condition that it can be implemented efficiently. The user interface and mechanism views in Domino *both* have to be implemented; human managers have to be able to work with objects which are implemented in accordance with their view, while the underlying system has to be able to use a mechanism which provides an efficient implementation of the functions seen by the manager.

3 DOMAINS

We have already observed the need to be able to group and structure objects for management operations and policies. This grouping should be hierarchical to reflect the management structure in many organisation. Figures 3.1 is an enterprise view of its personnel structure expressed as a tree. Hierarchical structuring methods have come into existence because it is,

in practice, impossible to manage large numbers of objects in a purely linear way, hence the widespread use of tree structures of objects in computer systems e.g. files systems as shown in Figure 3.2. Exactly the same reasoning applies to management policies; a means of hierarchical grouping of objects must be integrated into any system for defining policies. Policies may be expressed in terms of hierarchy, e.g. "members of an organisation have authority over those below them in the hierarchy" or "access to a directory gives access to all its subdirectories".

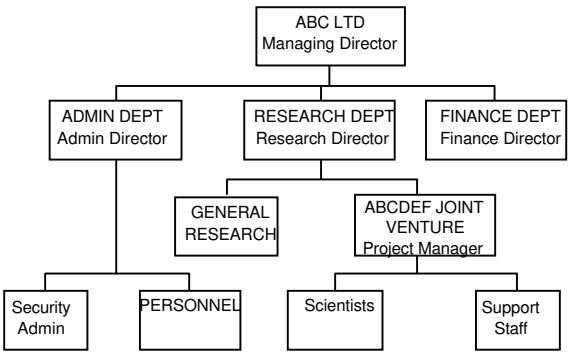


Figure 3.1 Organisation Tree

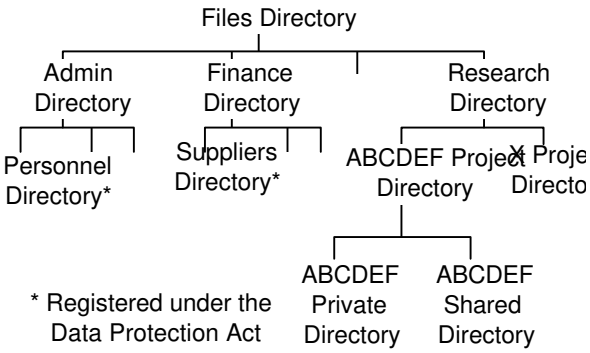


Figure 3.2 Directory File Structure

In order to represent these concepts we have defined **domain** objects, with attributes which include a **policy set** and a set of **constraints**. Their purpose is to group objects together for management purposes, while not affecting the objects' normal functionality. Domains are described in [Sloman 1989].

3.1 The Policy Set and Subdomains

The policy set of a domain consists of an enumerated set of member objects to which the policy associated with the domain applies. From the point of view of a user it would have been quite feasible to define domain membership in terms of a predicate on object attributes, e.g. "the set of all users under 65 years age". However determining the set of members would require the underlying mechanism to examine every object within the system to check whether it met the constraints of the predicate. Also tracking current domain membership as the attributes of distributed objects change would incur very heavy overheads (see our comments on constraints below). This is an example of problems with the mechanism view which forced us to drop a concept from the user view. However, another motivation for the decision is that it avoids potential logical paradoxes, such as Russell's Paradox: is the class of all classes which are not members of themselves a member of itself? This definition cannot be expressed by us and therefore the paradox cannot arise.

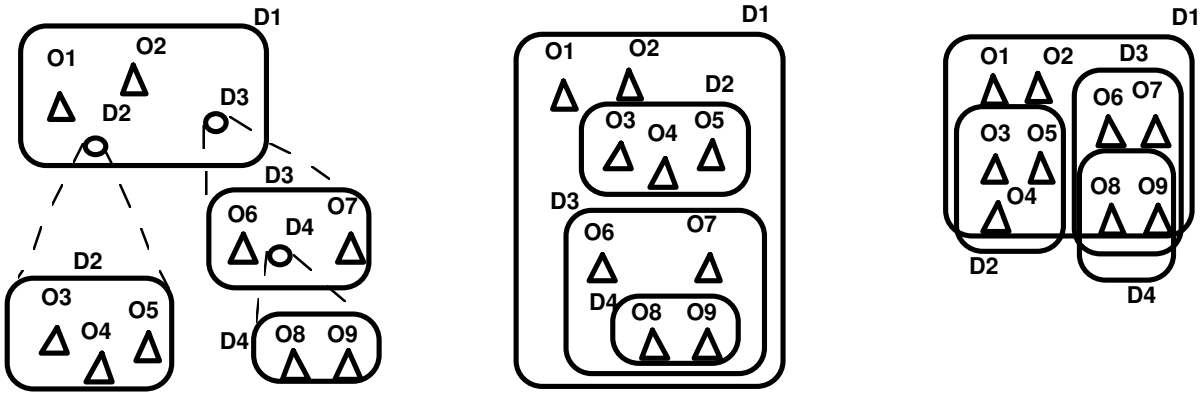
There are some definitions of "domain" which do not make it clear whether the members are defined by enumeration or predicate. For example the ISO Security Frameworks Overview [ISO 10181-1] says: "A security domain is a set of elements under a given security policy

administered by a single authority for some specific security relevant activities." We are given no indication of whether its members are defined by the predicate "under a given security policy ....." or whether they are an enumerated set of objects to which a policy is to be applied.

The policy set achieves a simple grouping facility by enumerating the set of objects which are **direct members** of the domain. Membership of domains has no effect on the state of the member objects. However, the power of domains for structuring is provided by the concept of **subdomains** which are domain objects that are members of other domains. In order to understand this, it is useful to explain the subdomain, overlap and subset relationships between domains.

If one domain object is a member of another, the first is referred to as a **subdomain** of the second. In figure 3.3a, we refer to D2 as a **direct subdomain** of D1, and to D4 as an **indirect subdomain** of D1. An object is a **direct member** of a domain if it is in the domain's policy set. An object is an **indirect member** of a domain Dx if it is a member of a domain Dy which is a subdomain of Dx. When we refer to a "member" (unqualified), we mean a direct or indirect member. If an object (domain or other) is a direct member of a domain, then the domain is its **parent**, and if it is a member, then the domain is its **ancestor**.

Two domains **overlap** if there are objects which are members of both domains. A special case of overlapping occurs when the objects in one domain are a **subset** of the objects in another. This is illustrated in figure 3.3c. An alternative notation for showing a subdomain relation, which we use in this paper for compactness, is shown in figure 3.3b. It is simpler, but could be mistaken for a subset relation. Figures 3.4 and 3.5 show the examples of figures 3.1 and 3.2 expressed in terms of domains in this way.



(a) Subdomain Relation

(b) Subdomain or Subset Relation (ambiguous notation)

(c) Subset Relation

Figure 3.3 Subdomain and Subset Relations

The effect of removal or inclusion of an object from a subset or subdomain has different results on the direct membership of the parent domain. If D2 is a *subset* of D1, as in figure 3.3c, then the addition of an object O10 to D2's policy set does not affect D1 in any way and there is no resulting relationship between O10 and D1. After the operation, D2 is no longer

a subset of D1, but D1 and D2 overlap. If D2 is a *subdomain* of D1, as in figure 3.3a, then O10 becomes an indirect member of D1 and of all parent domains of D1.

Figures 3.4 and 3.5 show the domain representation of the hierarchies shown in figures 3.1 and 3.2. In figure 3.4, Research Dept is a subdomain of ABC Ltd and General Research is a subdomain of Research Dept, so when a user becomes a member of General Research he automatically becomes an indirect member of Research Dept and ABC Ltd. However, if the relationships were subset relationships instead, it would be necessary to perform three separate operations to include the user in the three different domains. For this reason subsets are not a particularly useful relationship for structuring management.

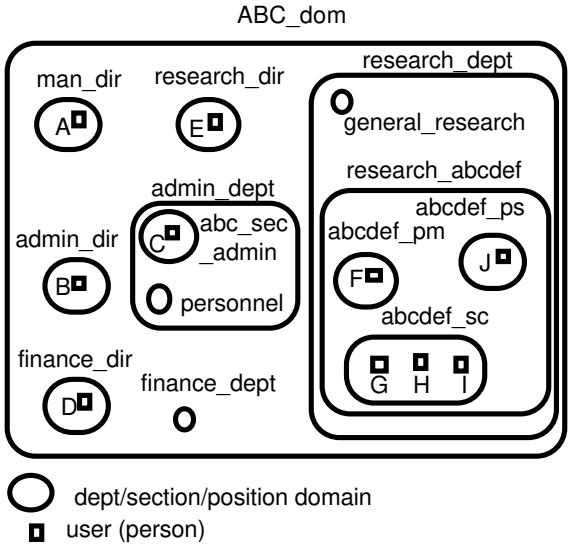


Figure 3.4 Domain Representation of Organisation

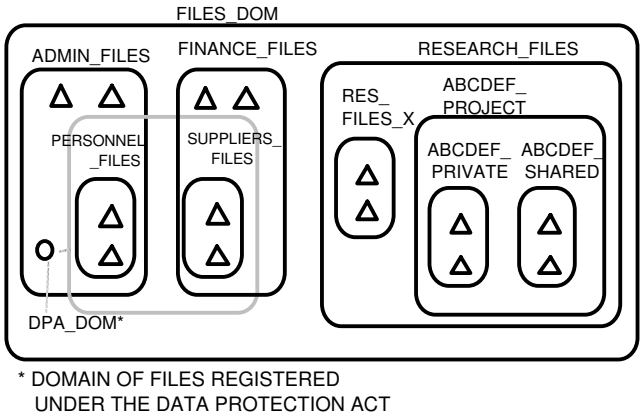


Figure 3.5 Domain Representation of Directory File Structure

The ISO Security Frameworks Overview defines "subdomain" as a subset rather than a membership relationship, which makes hierarchical structures difficult to express.

3.2 Constraints on Membership Operations

We envisage the constraints attribute of a domain being used to place constraints upon the membership of the domain. Some examples of possible application-specific constraints are:

- There must be at least two members of the domain of Security Administrators.
- Processors in a domain must be able to support the M68000 instruction set.
- Destroying an object is not possible if the object is still a member of another domain.

There are of course two possible kinds of constraint. The first, which we favour, is a constraint upon the operations which affect domain membership, typically the **create** and **destroy** operations for any object type and the **include** and **remove** operations on a domain These add and subtract objects to and from the policy set. The predicate defined by the constraint has to be evaluated once only, when the operation is performed. Constraints on

the number of members, object types or other object attributes can be enforced by constraints of this kind. The second kind of constraint is a general predicate, which the system is required to maintain, about the attributes of members of a domain. We regard this as being potentially very difficult to achieve, because every time any application functional operation attempts to change an object's attribute, the system is required to verify that the domain constraints, of every domain of which the object is a member, are not violated. We do not therefore envisage users being permitted to specify constraints of this kind. We still have a lot of work to do on constraints.

3.3 User and Mechanism Views of Objects and Domains

There is no incompatibility between the user view of objects being contained in domains and the mechanism view of actually holding references. The mechanism view is in fact what is implemented, and the user view is provided by filtering out the invisible mechanical features and presenting objects in terms of names, not object identifiers. This is a summary of the differences:

In the user's view, objects are referred to by local names (within a domain) or a domain path name plus local name. In the mechanism view they are referred to by unique identifiers, through addresses. An OID (Object ID), consisting of its address and identifier, is associated with each object.

The decision to take this approach for the mechanism was in order to avoid possible inconsistencies when an object is a member of several domains; there is only one copy of the object, upon which all operations are performed. The approach achieves consistency at the cost of having to perform remote accesses when the object is remote from one of its parent domains. A different approach is being considered by the DOMAINS project [DOMAINS 1991], where even in the mechanism view it will be a local shield object which is encapsulated with a manager within a domain. The shield object may cache state information relating to the remote managed object. This requires mechanisms to maintain consistency of the distributed and replicated state information relating to a managed object which is a member of multiple domains. Our experience indicates that the complexity and costs in maintaining consistency far outweighs the benefit of local caching of the comparatively small amount of management information needed for a managed object.

Policies relate to domains, and in order to determine the policies which apply to a particular object, it is necessary to know its ancestors. It is impractical to inspect all domains for this purpose, so in our implementation the domains service maintains a **parent set** for each object, which is a list of its parent domains, enabling derivation of its ancestors. The parent set is the implementation mechanism for the parent/ancestor relationship of the user view.

The relationships between the user and mechanism views of objects and of domains is illustrated in tables 3.1 and 3.2 .

User View		Mechanism View
Object Name	<-------->	OID
Parent Relationship	<-------->	Parent Set
Other Attributes	<-------->	Other Attributes

Table 3.1 User and Mechanism Views of Objects

User View		Mechanism View
Object Name	<-------->	OID
Parent/Ancestor Relationship		Parent Set
Policy Set { Name }	<-------->	Policy Set { (OID, Local Name) }
Constraints	<-------->	Constraints

Table 3.2 User and Mechanism Views of Domains

3.4 Domain Expressions

Domains and subdomains are a powerful means of expressing hierarchical membership and set union, but provide no means of expressing other basic set operations such as Set Difference and Set Intersection. Set Difference can express sets of objects such as "all files in Payroll_Files except Payroll_Master". Set Intersection can express sets such as "all files which are in Payroll_Files and also in Personal_Data". We have therefore introduced **domain expressions**, which allow the expression of a set of objects in a policy by means of a formula containing the standard set operations.

It would be possible to use domain manipulation operations to create a new domain with the required membership e.g. create DomA with the same members as (DomB $\cap$ DomC), and refer to DomA in the policy. However, the policy then applies to membership of DomB and DomC at the time DomA is created and not at the time that its applicability is checked. Static enumeration of objects at some point in the past is not usually what is required in evaluation of policies, and domain expressions in a policy such as an access rule are evaluated at the time the rule is checked.

4 POLICIES

We have recently presented a generic user view of policies [Moffett 1991] which identifies their essential characteristics and models them as objects with the following attributes: **subjects** who are motivated or authorised, depending on the **mode** of the policy, to achieve **goals** on target **objects**. An **access rule** is an example of this generic policy object.

The user view of an access rule is straightforward. It is an object with four attributes:

- **Subject Domain**, which is a domain expression defining the set of subjects of the policy;
- **Operation Set,** a set of permitted operations;

- **Target Domain**, which is a domain expression defining the set of target objects of the policy;
- **Constraints**, which limit the applicability of the access rule, and which we currently see as using generally available information such as date, time of day and the terminal at which the current user is logged on.

An access rule, shown in figure 4.1, maps exactly onto policy objects in which the mode is positive authorisation and the goals are operations in the interface of the target objects of the access rule. There is a good reason for users having a view which shows them access rules, rather than generic policies, and this is that they actually think in terms of making rules which authorise access or remove authorisation, rather than in terms of making generic policies. We envisage that a generic model of policies will provide a framework for analysis of policies to detect conflicts and inconsistencies.

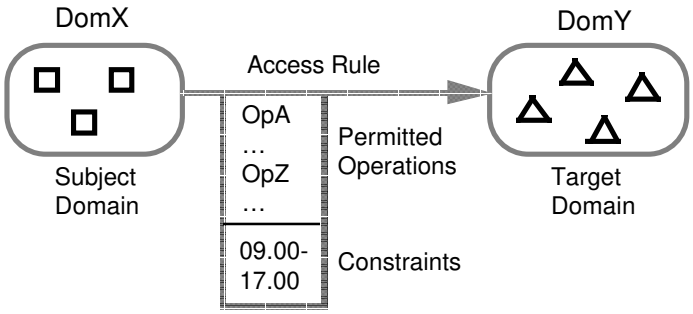


Figure 4.1 An Access Rule

The authority relationships are not determined by the domain hierarchy but by the access rules which specify an authorisation relationship between subject and target domains. The domains determine the **scope** i.e. the set of objects to which the policy applies. An access rule which applies to a parent domain is normally inherited by subdomains which are members of the parent domain, although this propagation to subdomains can optionally be inhibited.

An access rule specifies that any user object in the set defined by the User Domain of the rule is authorised to perform any of the operations in the Operation Set on any of the objects in the set defined by the Target Domain of the rule. There is a fixed policy that in the absence of authorisation by an access rule, an operation request (attempted access) is forbidden. In the user view of the system, whenever a user issues an operation request, the system must search every access rule in the system, allowing the request if an access rule matching it is found, but forbidding it if none can be found. This is clearly completely impractical as a mechanism.

We therefore use two implementations of access rules: an **access rule** object corresponds to the user view (figure 4.2) and is the means by which users specify access control policy. The underlying mechanism which controls access uses Access Control List (ACL) entries for each domain which is specified in the Target Domain expression of the access rule (for implementation efficiency the lowest level of granularity for target objects is a domain). The **Access Control Entries (ACEs)** consist of the User Domain expression and the Operation Set of the access rule object [Moffett 1990]. Every operation request carries with it an

authenticated list of the domains which are ancestors of the user object which makes the request. The system can therefore evaluate the request by reference solely to the information in the operation request and the parent domains of the target object. Details of this implementation can be found in [Twidle 1992]. The relationships between the user and mechanism views of access rules is illustrated in table 4.1 and figure 4.2. Note that the access rule applying to TDom1 in Figure 4.3, propagates to subdomains TDom2 and TDom3 and so appears as entries in the ACL of each domain.

User View		Mechanism View
Object Name	<-------->	OID
User Domain (Name Based DE[2])	<-------->	
Operation Set	<-------->	Combined to form an ACL entry which is an attribute of every domain affected by the Target Domain expression.
Constraints	<-------->	not yet implemented
Target Domain (Name Based DE)	<-------->	Every domain affected by the Target Domain expression holds an ACL entry.

Table 4.1 User and Mechanism Views of Access Rules

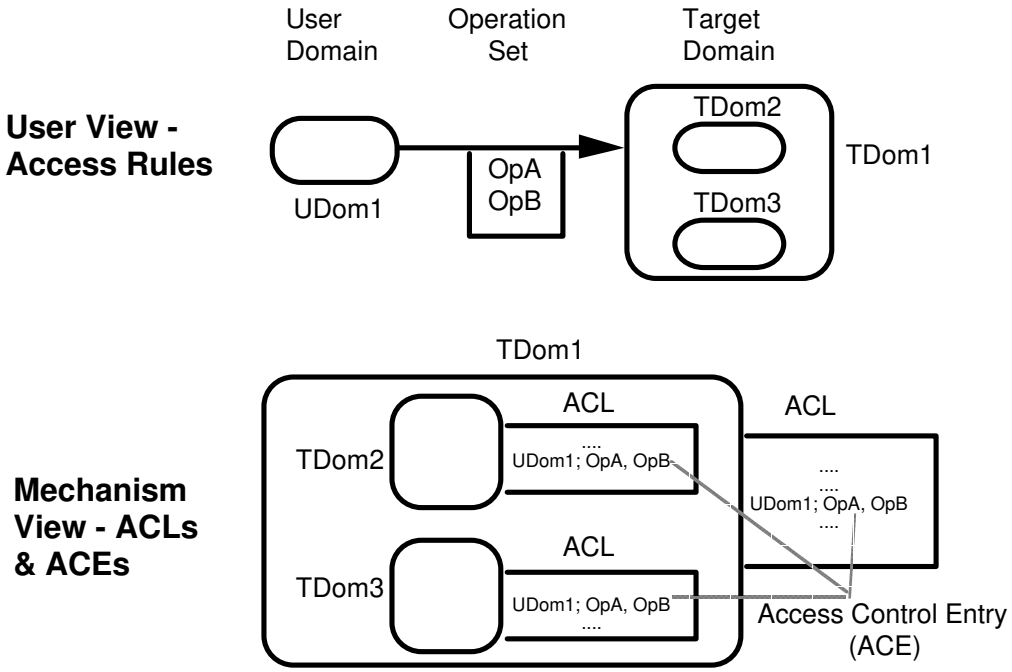


Figure 4.2 An Access Rule, and its Corresponding Access Control Lists and Entries

There is clearly a problem of ensuring consistency between the views of access rules. Ideally, at the design stage, we should have taken our formal specification of the user view

[2] DE - Domain Expression

and either refined it to the mechanism view while ensuring the preservation of properties or created a formal specification of the mechanism view and proved that the properties were preserved. However, in the absence of tools to support these activities we have opted for an informal design of the mechanism view.

There also a problem of ensuring consistency between views during operation, as the two views can become inconsistent through system errors and failures. We are therefore developing consistency checking and restoration tools in order to diagnose and recover from any inconsistencies.

The mechanisms used to support our access rules can be based on those found in typical distributed operating systems. An ACL entry is specified in terms of domains rather than individual users and maintains a reference to the source access rule, from which it was generated, as an aid to maintaining consistency.

5 REPRESENTING USERS AND MANAGER POSITIONS

Two default domains shown in figure 5.1 are needed to represent human users within the system:

i) A **User Representation Domain (URD)** is a persistent representation of the human user or manager. When the user logs into the system an user interface object is created within the URD and inherits all access rules specified for the URD. An authentication service is still needed to authenticate a user and determine his/her URD.

ii) A **User Personal Domain (UPD)** corresponds to a user's home directory and represents the personal resources which the user "owns". In addition the user may have limited access to other service domains representing the shared resources the user can access.

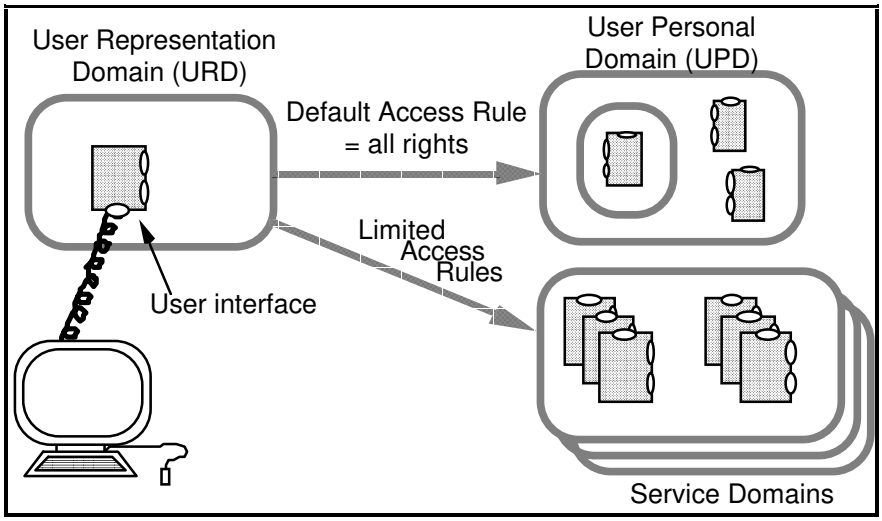


Figure 5.1 Default User Domains

In most cases policies should be expressed not in terms of individual users but in terms of **manager positions** which they occupy so that when the human manager is transferred to another position, the access rules pertaining to the positions do not have to be changed. Since positions are "occupied" and domains have members, the correspondence is so close that, even at the user view, we regard it as satisfactory to see positions as being represented by domains, with constraints on membership operations to ensure that only user objects can be members and that there are the appropriate limits on the number of members. As far as practical, all access rules are specified with respect to a Manager Position Domain (figure 5.2). Allocating a human manager to a position is accomplished by including his/her URD within the position domain. The manager automatically inherits all rights for that position and may be a member of multiple position domains if performing multiple management roles.

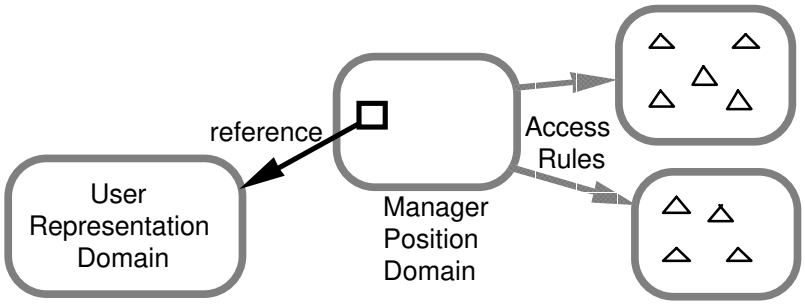


Figure 5.2 Manager Positions

6 RELATED WORK

6.2 Alternative Views of Domains

One feature of several other approaches to the definition of management domains stands out. This is a definition of domain which includes both the manager and the objects which are managed [ISO 10181-1, DOMAINS 1991]. It may also include an implicit or explicit definition of the management operations or goals in the domain. It then enables policy to be expressed in a single object, without the need for a separate policy object. Also, in the DOMAINS approach the domain encapsulates the manager and objects.

Domains should not be defined to include managers as well as target objects, for the following reasons:

- Managers in one organisation may be given (limited) management rights over objects in other organisations. A domain which includes managers from one organisation with objects from another organisation leads to policy and implementation problems. The manager or the managed object has to become a member of a domain "owned" by another organisation. What are the policy implications of this? Does the access control system allow the operation? What if an external manager is only to be allowed a limited subset of the management operations which are available to local managers?
- Encapsulating managed objects plus managers in a domain makes it very difficult to permit the objects or managers to be in multiple domains. Implementing a "shared"

encapsulated object is not easy as the encapsulated object is usually dependent on the encapsulating object for its existence. In some circumstances it may be appropriate to encapsulate a manager with the objects it manages and treat this composite object as a single entity. However the model should permit, but not enforce it.

- A single manager object may be responsible for managing multiple independent sets of objects and so would have to be included in multiple domains.
- There is a need to specify policy in terms of the user's position in an organisation, not the individual user, and so a new concept of position has to be invented.
- There is sometimes a need to treat managers as managed objects in a domain hierarchy which is independent from the domain hierarchy of the objects they manage. How can this be achieved if managers are in the same domains as the managed objects?

We have no doubt that these questions can be solved, but only at the cost of additional complexity. We believe that our simple concept of domain and the use of access rules to indicate the relationship between managers and managed objects makes it much easier to represent the flexible relationships shown in figure 6.1.

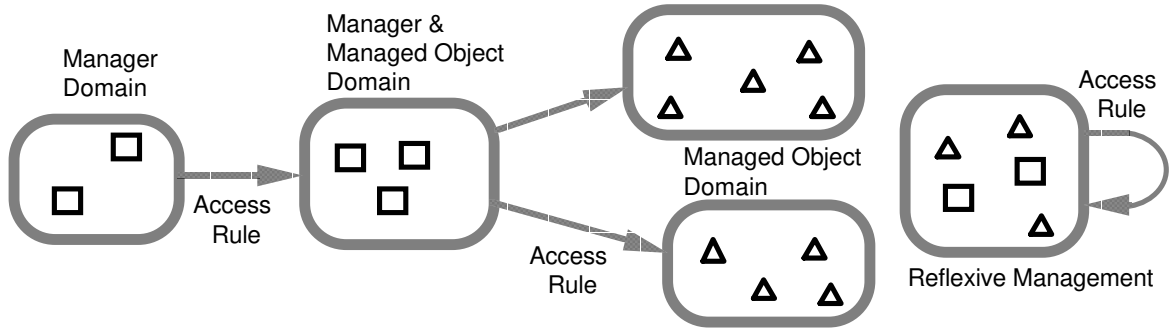


Figure 6.1 Typical management relationships

The DEC Enterprise Management Architecture [Strutt 1991] uses a domain to define a "sphere of interest of a set of managed objects for a manager" - managers are not part of the domain. Only the managers are aware of the domains and objects do not know which domains they are members of. This model is compatible with the Domino approach as it implements a subset of our concepts. There are no domains of managers and relationships between a manager and domain are implicit in the knowledge of the domain name rather than explicitly shown by the use of access rules. They do not use the domains for access control. Domain membership information is held by the name service.

An application which uses domains to specify the service entitlement of subscribers in a multi-service international mobile cellular network is described in [Sloman 1993]. This also discusses the relevance of domains to accounting.

6.2 ODP Viewpoints

The ISO Open Distributed Processing (ODP) work [Linington 1991, ODP 92] defines enterprise, information, computation, engineering and technology viewpoints for distributed processing. These viewpoints reflect different design concerns and characteristics of a distributed system.

Our user interface view has points in common with both the information and computation viewpoints. It describes the information which a user is dealing with, but it is also explicitly an automated view, because it is the user interface to a system.

At first sight our mechanism view would appear to map onto the ODP engineering viewpoint. However, the mechanism view does not provide the typical engineering transparencies such as location independence or fault tolerance but provides identical functionality with better performance. These transparencies may still be needed, so our mechanism view really corresponds with the ODP computation view.

We believe that our identification of a user interface view is an essential aspect of our system modelling which is not recognised explicitly by ODP. A corollary of this view is a consideration of the performance implications which then forces a mechanism view upon us. Our user/mechanism views appear to be orthogonal to the ODP viewpoints. If they were to be modelled in terms of ODP viewpoints there would probably be a common enterprise viewpoint for both interface and mechanism. However there are information, computational and engineering viewpoints corresponding to the implementation of the user view and a another set corresponding to the implementation of the mechanism view, although there may be some common aspects in the technology viewpoint.

7 CONCLUSIONS

There are several tasks to be performed by users carrying out Distributed System Management. One of these is grouping and structuring large numbers of objects. We achieve this by our concept of Domain objects, allowing hierarchical structuring by allowing domains to be members (subdomains) of other domains. Another task is to define the actions which managers are motivated and/or authorised to do, and the objects which are the targets of their management. This is done by the definition of a generic Policy object, of which Access Rules are a specialisation.

Work on implementation in the Domino project has shown that it is essential to have separate coexisting user and mechanism views of some of the concepts, both of which are implemented. The user view emphsises conceptual clarity while the the mechanism view emphasises efficient implementation. There are substantial differences between the user and mechanism views of Domains, while retaining the same basic structure. On the other hand the structure of the mechanism for Access Rules is completely different from the object which the user sees.

We have developed an approach to Distributed System Management concepts which separates the different concerns of managers into two main types of object which they can work with: Domains and Policies. We have further separated these into User and Mechanism views. This has enabled us to address our work to the correct concern and at the correct level, in order to make sound progress.

ACKNOWLEDGEMENTS

Acknowledgement is made of the contribution to the above concepts by our colleagues in the DOMINO projects, and in particular Kevin Twidle who has been responsible for much of the

implementation of the domain service. We gratefully acknowledge the support of the DTI/SERC under grant GR/F 35197.

REFERENCES

[DOMAINS 1991] Esprit Project 5165 - DOMAINS Basic Concepts, version 2.0, Nov. 1991, Philips Gmbh., PO Box 1980, W 5100 Aachen, Germany.

[ISO 10040] Systems Management Overview, ISO/IEC JTC1/SC21 WG4, 1991.

[ISO 10181-1] Security Frameworks Overview, ISO/IEC JTC1/SC21 N6693, January 1992.

[Linington 1991] Linington P., Open Distributed Processing and Open Management, in Krishnan I. & Zimmer W. (eds), Integrated Network Management II, North Holland, 1991, pp. 553-562 .

[Magee 1993] Magee J., Dulay N., Kramer J., *Structuring Parallel and Distributed Programs,* IEE Software Engineering Journal, Special Issue on Configurable Distributed Systems, 8(2), March 1993, pp. 73-92.

[Moffett 1990] Moffett J.D., Sloman M.S. & Twidle K.P., Specifying Discretionary Access Control Policy for Distributed Systems, Computer Communications, 13(9), November 1990, pp 571-580.

[Moffett 1991] Moffett J.D. & Sloman M.S., The Representation of Policies as System Objects. Proc. Conf. on Organisational Computer Systems (COCS 91), Atlanta USA, Nov. 1991, in SIGOIS Bulletin vol 12 nos. 2&3, pp. 171-184.

[ODP 92] ISO/IEC JTC1/SC21 10746-3 Basic Reference Model for Open Distributed Procesing, Part 3, Nov. 1992.

[Sloman 1989] Sloman M.S. & Moffett J.D., Domain Management for Distributed Systems, in Meandzija & Westcott (eds.), in Integrated Network Management I, North Holland, 1989, pp 505-516.

[Sloman 1993] Sloman M.S., Varley B.J., Twidle K.P. & Moffett J.D., Domain Management and Accounting in an International Cellular Network. in Integrated Network Management III, Hegering H. & Yemini Y. (eds.), North Holland, 1993, pp 193-206

[Strutt 1991] Strutt C., Dealing with Scale in an Enterprise Management Director, in Krishnan I. & Zimmer W. (eds), Integrated Network Management II, North Holland (1991), pp. 577-593.

[Twidle 1992] Twidle K.P. & Moffett J.D., Domino Reference Manual, Domino paper A3/IC/4, June 1992.