Imperial College of Science, Technology & Medicine

(University of London)

Department of Computing

Delegation of Authority Using Domain-Based Access Rules

Jonathan D. Moffett

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy of the University of London, and the Diploma of

Imperial College of Science, Technology & Medicine

July 1990

Corrected September 1990

Delegation of Authority Using Domain-Based Access Rules

Abstract

Security is becoming increasingly important for commercial computer systems, which have received little academic attention compared with the security requirements of military systems. In particular, two major concerns of commercial security are discretionary access control and the delegation of authority to give access. It is essential to be able to specify policy for these in a way which gives flexibility while retaining management control. This is the subject of this thesis.

Many of the systems are very large, with distributed processing. Systems such as these have two important characteristics. First, the number of objects in the systems, totalling hundreds of thousands in some cases, can be so large that it is impractical to specify access control policy in terms of individual objects or individual users. We need to be able to specify it as a relationship between groups of users and groups of objects. Second, the systems typically consist of multiple interconnected networks and span a number of different organisations. Authority cannot be delegated or imposed from one central point, but has to be negotiated between independent managers who wish to cooperate but who may have a very limited trust in each other.

We introduce, model and validate two concepts:

- *Access rules*, which are a means of specifying discretionary access control policy between groups of users and groups of objects.
- *Delegation of authority* to create access rules, from an initial possessor of authority who we refer to as an *owner*, via *managers*, to *security administrators*.

Both access rules and the delegation of authority are modelled using *management domains* as the means of grouping objects.

A formal specification of the concepts is given in the Z specification language. Policies for delegation of authority are described graphically by means of Petri nets which map into Z. The concepts are validated by a Prolog animation of the specification on realistic examples. A tool for the analysis and validation of structures of domains and access rules is also provided in Prolog.

Acknowledgements

First I wish to acknowledge the great help and support given me by Morris Sloman: supervisor, guide, friend and squash partner. His encouragement from the start, his great conscientiousness in reading and commenting on all stages of my work and his open-mindedness in allowing my work to develop, have helped me immensely. He has been a lodestone for my direction, a touchstone for my progress, a whetstone to sharpen my ideas on, a grindstone to bring me back to base material on occasions - but above all, a real gemstone!

Thanks too to Jeff Kramer and the Distributed Software Engineering Group for welcoming me into their midst. Kevin Twidle has been particularly helpful with technical support, and my approach to implementation issues owes a lot to discussions with him.

The research students, RAs, lecturers and others whom I wish to thank are numerous. The following are among many who have contributed help, ranging from comments on earlier drafts, through technical help and discussions, to sustaining cups of coffee. Thank you Jaelson Castro, S.C. Cheung, Naranker Dulay, Anthony Finkelstein, Orly Kremien, Jeff Magee, Keng Ng, Anne O'Neill, Diana Protic, Suresh Rasakulasuriar, David Robinson, Sati Sian, John Smith and Andrew Young. Thanks too to for the help of colleagues on the Domino project, John Haberfield and Tony Law of BP, and Tony Jeffree of Sema Group.

I am grateful to Esso Petroleum plc for their financial support during the earlier stages of my research. Later parts of the work have been carried out with the support of the DTI/SERC (grant no GR/F 35197) for the Domino project.

To Jennifer: you next!

CONTENTS

I INTRODUCTION

This thesis is concerned with the modelling and validation of two concepts:

- *Access rules*, which are a means of specifying detailed policy for access control in computer systems. Access rules are based on the grouping of objects by means of *domains* [Sloman 1989].
- *Delegation of authority* to create access rules, from an initial possessor of authority who we refer to as an *owner*, via *managers*, to *security administrators*.

I.1 Motivation & Aims

Motivation

We are concerned with very large distributed processing systems which have two main characteristics.

First, the numbers of objects in the systems can total hundreds of thousands. There are very large numbers not only of target objects, such as files, but also of users, possibly thousands of them. Therefore it is impractical to specify access control policy in terms of individual objects or individual users. We need to be able to specify it as a relationship between groups of users and groups of objects. Methods which require managers to specify either individual users and groups of target objects or individual target objects and groups of users are inadequate for the purpose.

Second the systems typically consist of multiple interconnected networks and span a number of different organisations. Authority cannot be delegated or imposed from one central point, but has to be negotiated between independent managers who wish to cooperate but who may have a very limited trust in each other.

We can distinguish three distinct but connected motivations for the work in this thesis: modelling the use of authority in large organisations, whether centralised or distributed, modelling the controlled delegation of authority between independent managements in distributed systems and defining the use of discretionary access control as a management tool.

In fact the three motivations reduce to two, as soon as one examines the effect of delegation of authority in a centralised organisation. Another word for 'delegation' could be 'distribution'; although all managers in an organisation may ultimately be governed by the same entity, in practice they act independently within the limits of their authority, and

negotiate with one another in the same way as truly independent managers. We have found that a solution to the problem of representing authority in a centralised organisation also provides the mechanisms needed in a distributed system.

Discretionary access control can be regarded simply as an aspect of computer security. However, once we accept that there must be some controls placed upon the actions of people who work together in organisations, it can be regarded as one of the primary tools with which managers define policy in a system. We model all actions within such a system as operations carried out on objects, and the giving and withholding of access rights is the equivalent of the granting and removal of authority.

General Requirements

One characteristic of management is that it is concerned with treating collections of similar objects in a similar manner, for reasons of both efficiency and principle. Managers do not wish to take individual actions to give individual people access to individual resources. They would prefer instead to take the action in relation to a group of people and a group of resources. Further, for many purposes they do not make decisions about the authority of people, but of positions. Typically the decision is that 'the Payroll Clerk should change the Payroll Master file', and John should have that authority only because he occupies the position of Payroll Clerk. We need to be able to express authority in these terms.

Therefore the use of discretionary access control requires that we solve the problem of doing it in terms of groups of users and resources, and of organisational roles occupied by users.

Our requirements for delegation of authority are twofold: to represent a management structure and policy as a framework within which an organisation can function; and to provide mechanisms for the transfer of authority from one agent to another within that framework. Two conditions must be met for the latter: firstly, resource owners should be able to delegate authority for operations on their resources to other system users, within the mandatory constraints of the system; and secondly, users must be prevented from performing operations on resources without prior authority to do so.

Here are some examples of requirements for delegation mechanisms:

- An owner can delegate responsibility to a security administrator to give access authority on his objects to users in a defined part of an organisation.
- An owner can pass on control of some of his objects to another user. He needs the option to allow or forbid that person to further pass on control. He may or may not retain ownership himself also (joint ownership), but he needs the ability to remove control again from a user he has given it to.

- Users can have personal domains[1] where they are able to control anything they create.

Aims of the Thesis

There are two aims of this thesis: to describe a method of providing discretionary access control in very large systems, by means of access rules; and to describe how access authority can be delegated from the original owners of objects, through managers and security administrators, to system users.

I.2 Our Approach

We can characterise our approach as follows.

a) Descriptive Method

Our thesis uses four different, interlinked, methods of describing our model:

- A formal description, on which our other methods are based, uses the Z language [Hayes 1987, Spivey 1989] and is contained in Appendix A.
- An informal English language description, forming the body of this thesis, is used as the primary method for communicating the model to other people. There is no formal translation from Z to English, but the informal description has been continually validated against the Z.
- Domain diagrams using our own notation, refined from [Robinson 1988b], and Petri net diagrams [Peterson 1981] of authority transitions. These are precise formal translations from the Z language, and their interpretation is described in Appendix A.8.
- Finally, to give confidence that the model actually works, there is a Prolog animation to provide a prototype implementation. See section VI.1 and Appendix B. It does not solve the problems of obtaining reasonable performance in a real-world distributed environment, but it has been invaluable as a means of ensuring that a syntactically correct formal description makes sense in a realistic example.

b) Object-Based Approach

We base our approach on an object-oriented view of systems. All entities in our model, including those which represent authority, are viewed as objects which export a set of

[1] We use the term *domain* informally at this stage to denote a set of objects; it is defined more precisely in chapter II.

operations to be invoked by the sending of messages to the object. This has eased the problem of reaching a consistent and rigorous model.

c) Mechanism for Grouping and Structuring Objects

One of our primary aims is to handle the problems of scale by grouping and structuring objects. The use of *management domains* [Sloman 1989] has enabled this and also provides a means of representing organisational positions which are occupied by users.

d) Modelling of Authority

We needed a means of modelling authority which would use the expressive power of domains. Authority is represented by *access rule* and *role domain* objects, in which the kind of authority is denoted by the type of object, and both the possessors of authority and the scope of their authority are represented by domains and domain expressions.

I.3.1 What is New in Our Approach

There are three aspects of our work which are new:

a) *Access rules* - There is a need to group both users and target objects in the specification of access control policy, in order to make security administration possible. This has long been recognised in commercial systems, e.g. the RACF and ACF2 access control systems [IBM 1985 & CA 1988] for IBM 370 systems. However, these are *ad hoc* designs, driven by the characteristics of the system, and are in no way general models. Surprisingly, we have been unable to find any general access control model which provides for the flexible grouping of both users and target objects; Access Control Lists (ACLs) and capabilities only allow for the possibility of grouping one or the other. This thesis introduces the access rule concept.

b) *Delegation of authority* - we introduced the concept of delegation of authority in computer systems in [Moffett 1988] in a rather informal and less methodical way. This thesis provides a more consistent and formal approach.

c) *Formal definition of domains* - although the domains concept, introduced by [Robinson 1988b] and developed by [Sloman 1989] is not new in our work, we here introduce a formal definition of domains in the Z language.

I.3 Thesis Outline

The organisation of this thesis is as follows. Chapter II reviews the background to our work, and related work, in four sections: Security Concepts, Discretionary Access Control, Domains and Management Concepts. Chapter III describes our approach to discretionary

access control using domain-based access rules. Chapter IV, on delegation of authority, has four main parts: the first illustrates why access rules, by themselves, are inadequate as a mechanism for controlled delegation of authority; the second describes the requirements for delegation of authority by means of a sample management policy; the third introduces the concept of role domains for representation of management authority, demonstrates how it can achieve the requirement which we have stated; the fourth discusses more general management structures and policies. Chapter V discusses the implementation issues which we have identified, grouping them by subject: Prolog prototype, domains, access rules and the delegation of authority. Chapter VI provides two extended examples of authority in organisations: the first, composed by ourselves, is intended to illustrate the requirements of a typical commercial organisation; the second takes the example used in [Yu 1989] and illustrates that its demands can be met by our model. Chapter VII presents conclusions, a critical evaluation of our work and suggestions for future work.

Appendix A presents a formal description in the Z language of the work in chapters III and IV. Appendix B is a Prolog version of Appendix A, with our first example used as data; also, a prototype set of queries for validating and evaluating a domain-based system is provided.

I.4 Terminology and Conventions

We use the term *user* for the object which initiates an operation, whether the object is a principal representing a human user or an agent acting on behalf of a principal. The choice of *user* rather than *subject* or *client user* is a matter of historical compromise. Similarly we use *target* or *target object* for the object on which an operation is invoked.

We conform to the definitions in the OSI Security Architecture [ISO 1988] and Frameworks [ISO 1989a, b & c], except as noted here:

Policy and *Rule*. We have not made any attempt to define these terms except by usage. However, in general, we have used 'Policy' when describing constraints and procedures which are fixed, and 'Rule' when describing dynamically changeable access rules. Our usage conforms to well-established commercial practice but differs from [ISO 1988], where rule-based policies are defined as applying globally.

Authority is implicitly defined in [ISO 1989b] as being a human agent who presumably has the legitimate power which we define to be authority. There is no conflict between these two different usages.

Access rules as we describe them are an example of ISO's *identity-based security policy*.

The symbols used in diagrams have conventional meanings and formal interpretations which are explained in Appendix A.8.

I.5 Note on the Formal Specification

Although the main body of this thesis is in informal English language, throughout our work we have maintained a formal specification which parallels it. We first used Prolog for this purpose, but its lack of modularity or type-checking made it unsuitable for our purpose. We chose the Z language [Spivey 1989] because of the good training courses which were available and its widespread industrial use.

The reason for use of a formal specification has been primarily to achieve precision, particularly in order to avoid the ambiguities which threatened to arise in the definition of domains. We have also used it for proofs; in particular, to prove that the example management policy which we have used, has the desired properties of being hierarchical and in a single chain (see IV.7.5). The English language, the Z specification and the diagrams have been developed in parallel, with each of them being first for different parts of the work. The full Z specification is contained in appendix A.

The full Z specification was finally used as a basis for generating the Prolog program which is discussed in section VI.1 and listed in appendix B.

II BACKGROUND & RELATED WORK

The structure of this chapter is as follows. Section II.1 discusses general security concepts, in order to provide the context for our work. Section II.2 introduces and discusses the concepts in *discretionary access control*, of which our work forms a part, and shows that previous models are inadequate for realistic demands. Section II.3 gives an outline of work on *domains*, which are a basis for our approach. Section II.4 discusses general management concepts and *authority*.

II.1 Security Concepts

II.1.1 Definitions

Security is a very general term, with no agreed precise definition. We show the concepts with which we are concerned in figure II.1, based on standard computer security textbooks such as [Parker 1981].

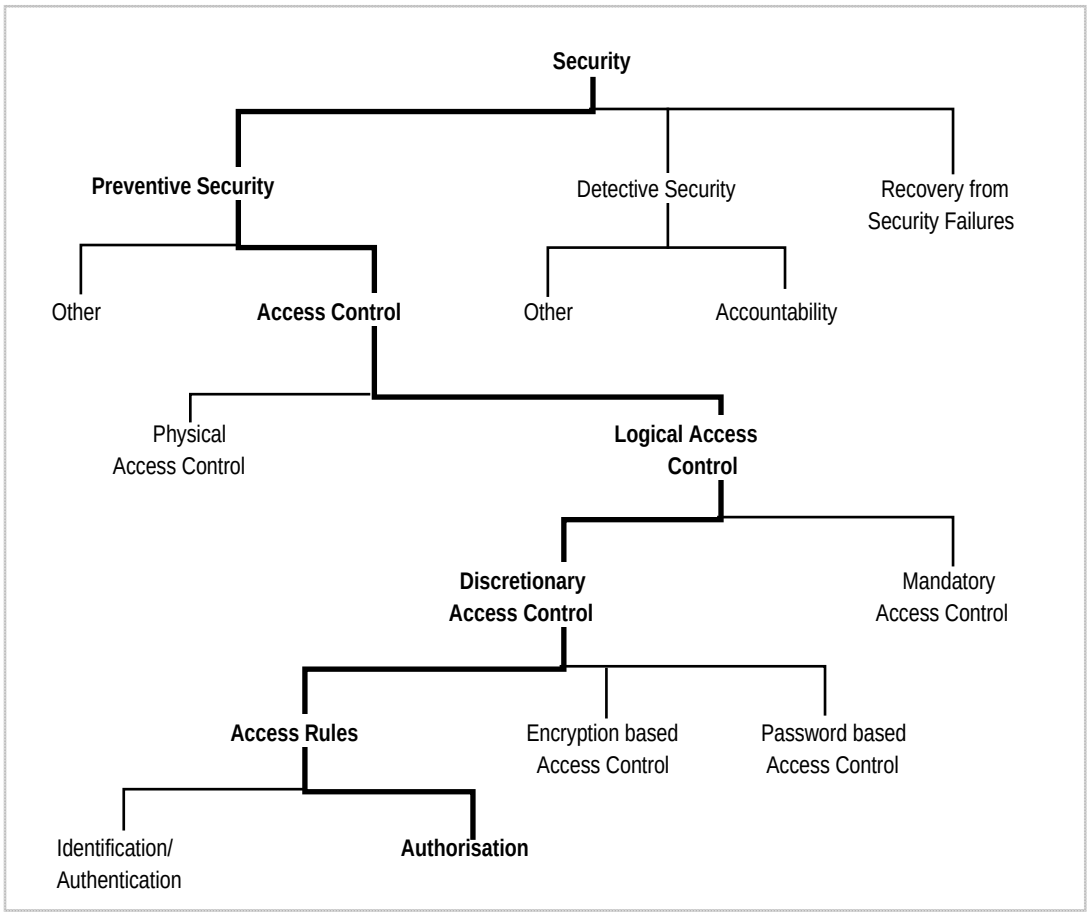

Figure II.1 Security Concepts

Security is concerned with three types of activity:

- *Preventive Security* - protecting physical and information assets from accidental or deliberate unauthorised modification, destruction or disclosure (security violations)
- *Detective Security* - detecting security violations as early as possible
- *Recovery* - recovering as far as possible from security violations if they occur. We are not concerned with this aspect of security in this thesis.

Access control is an aspect of Preventive Security which ensures that the operations which users and processes can perform on computer resources are done in a controlled and authorised manner. [Voydock 1983] classifies the attacks that access control must deal with into three categories: unauthorised release of information, unauthorised modification of information and unauthorised denial of use of resources. Access control is divided into *physical* and *logical access control*, both of which are necessary for a secure system. Physical access control uses physical mechanisms such as locks on doors. We are concerned mainly with logical access control - the protection of computer based resources from people who have already gained physical access to the computer, whether in close proximity to it or via telecommunications.

II.1.2 OSI Standards and Security

The scope of security, as expressed in the Open Systems Interconnection (OSI) standards [ISO 1988] is limited to the protection of communications, but most of its concepts apply more generally.

OSI Management Functions

OSI communication system management standards partition overall management into areas of functional responsibility [Klerer 1988]. The main functional areas defined by OSI are:

- Configuration Management,
- Performance Management,
- Fault Management,
- Security Management,
- Accounting.

Security Management is concerned with maintaining the security mechanisms of the system, e.g. access control, encryption facilities and physical security. Its aims are to ensure the protection of resources by preventing unauthorised access to them and monitoring exceptional states such as unauthorised access attempts.

An alternative view, [Sloman 1990], is that security should be regarded not as a management function but as a service which may itself require management, e.g. encryption is a service which requires key management. However, it is not important for our purposes which view is held, as we concentrate on access control as a service, and access control mechanisms.

OSI Security Architecture

ISO Standard 7498/2 [ISO 1988] defines the OSI Security Architecture. Its three main concepts are *Security Services, Security Mechanisms* and *Security Management*. The services are made available to users, and are supported by the mechanisms.

The *Security Services* defined are:

- Authentication - both peer entity authentication and data origin authentication,
- Access control - protection against unauthorised use of services accessible via OSI,
- Data confidentiality and integrity,
- Non-repudiation.

The *Security Mechanisms* by which the services are provided are:

- Encipherment,
- Digital signatures,
- Access control mechanisms,
- Data integrity mechanisms,
- Authentication exchange mechanisms,
- Traffic padding,
- Routing control,
- Notarization (trusted third party assurance).

There are three categories of OSI *Security Management* activities:

- System Security Management, concerned with the overall OSI environment, and including overall security policy management, interaction with other OSI management functions and interaction with the other categories of management.
- Security Service Management is concerned with the selection and imposition of mechanisms to provide a security service.
- Security Mechanism Management is concerned with particular mechanisms, e.g. key management.

We are concentrating on particular access control mechanisms which could be used to provide an access control service.

II.1.3 Trusted Computer Evaluation Criteria

There are two main manuals issued by the US Department of Defense which are of interest to us. The Trusted Computer Evaluation Criteria [DoD 1985] deals with access control to individual systems, and the Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria [DoD 1987] extends this to networks.

Mandatory and Discretionary Access Control

Logical access control is divided by [DoD 1985] into two categories: mandatory and discretionary. *Mandatory access control* enforces polices which are built into the design of the system and cannot be altered except by installing a new version of the system. [DoD 1985] does not define it formally, but an example is the policy that in 'multi-layer' security systems data cannot be read by a user with a lower security classification than has been assigned to the data. It defines *discretionary access control* mechanisms as those which allow users to specify and control sharing of objects with other users. The 'C2' level discretionary access control policy is defined as requiring mechanisms which ensure that objects are protected from unauthorised access and that access permission is only assigned by authorised users. We adopt this policy as the basis for our approach. It is an *identity-based security policy* in the terms of [ISO 1988]. Several other discretionary access control policies, based on other mechanisms, are available, such as passwords or encryption for individual resource protection; they are outside the scope of this thesis.

Network Security

[DoD 1987] deals with security in networks of computers, and must therefore be considered for its relevance to distributed computing. At the C2 security level its main emphasis is on access control by individual hosts, and possible approaches to authentication across a network. At this level it is consistent with our approach of treating access control to computer-based resources as primarily that of protecting individual computer systems, together with adequate communications transmission security, as dealt with by e.g. [Voydock 1983].

II.1.4 Accountability

Accountability, the ability to trace actions affecting security to the responsible party, is one of the fundamental concepts of computer security [DoD 1985]. All systems providing access control need accountability, at least for selected actions such as those of security administrators. This influences the choice of which access control method is used. A method such as file passwords, in which the user does not have to explicitly identify himself,

provides no support for accountability. On the other hand the access rule approach relates users to their actions as an integral part of the mechanism. In addition it can provide direct support for the implementation of accountability by extending the definition of access rules to include the ability to request the logging of authorised accesses.

II.1.5 Authentication

The goal of authentication is to counter attacks based on the usurping of identities of entities which operate either independently or on behalf of other entities. It depends upon the association of an identifier with an entity, and is defined as the means of providing adequate assurance of identification [ISO 1989b].

Access rules depend for their effectiveness upon the reliable identification of users, so they require users to authenticate their claims to use identities on the system. We assume that an effective identification and authentication system is used, and concentrate on the authorisation of access attempts by users who are authenticated at the appropriate point.

To make this assumption, we need to be satisfied that there are realistic schemes for authentication in distributed systems. [Needham 1978 & 1987] introduced an encryption-based scheme for large networks. A working example of a service based on this scheme is Kerberos [Steiner 1988], which provides an authentication service for MIT's Athena project. [ISO 1989b] is the basis for introduction of authentication standards into an OSI environment. We can therefore reasonably assume the availability of authentication.

II.1.6 Integrity

Integrity is an important concept, particularly in commercial systems. It forms the basis of a model of commercial security by Clark and Wilson [Clark 1987]. The aim of integrity is to prevent the unauthorised alteration of data in order to prevent fraud and errors. They introduce the concept of a well-formed transaction which is certified to preserve the integrity of the system's data, buttressed by enforced segregation of duties to deter fraud. The model does not define an aspect of access control, but depends upon access control and authorisation as effective mechanisms. It is therefore consistent with, and complementary to, our work.

II.1.7 Trust

An important characteristic of distributed systems is that they may involve cooperation between independent agents who do not wholly trust each other. Therefore the concept of *trust* is relevant to us. The subject has been dealt with in several papers. We have

reservations about the need for an elaborate treatment of the concept but we cover it here for two reasons: for completeness; and because a brief discussion of trust illustrates the boundaries of our own subject.

[Neely 1985] introduces *trust domains*, which are constrained to behave in certain ways and are entitled to expect other trust domains also to conform to constraints. Domains themselves are defined to be of type 'node' or 'link'. Trust describes not only a relationship between two objects but also the behaviour expected of one of them. Possible relationships between two trust domains - 'contain', 'adjoin' and 'associated' - are shown graphically but not defined formally. The informal specification of trust domains, and their specialised nature, means that it would be difficult to build directly on this work.

[Robinson 1988a], dealing with a capability based access control system, specifies that the destination node must *trust* the sender to have acquired a valid capability and have acquired the right to use it, and the sender must trust the destination node to interpret the capability correctly. He requires trust relationships to be reflexive, symmetric and transitive, thus deriving an equivalence class which is referred to as a *sphere of access*. Since many real life trust relationship are neither symmetric nor transitive, his work is of very limited application.

[Rangan 1988] bases his concept of trust on a modal logic of the beliefs of agents in a distributed system, using Kripke-style semantics. A trust specification is an additional axiom in the logic. He bases the concept on the assumption that there is a global state of a distributed system which can be referred to, which we cannot assume.

[Stepney 1987], discussed below in section II.2.3, uses trust relations between Authorities in a model of access control.

The main applicability of the concept of trust is in the *motivation* of decisions to carry out actions, and not the semantics of the actions themselves. Thus if A is deciding whether to delegate authority to B, the degree to which A trusts B is one relevant factor in his decision. Trust considerations will also affect general policy about the constraints which may limit A's action. However, trust is not relevant to the semantics of setting the constraints, or to the semantics of delegation of authority once A has made his decision.

In this thesis we are dealing almost entirely with semantics and not motivation. We work within an environment in which we make coarse assumptions about being able to trust key system components, such as the reference monitor, but we do not need to consider trust between objects in the system. We accept that this is important and interesting, but it is not our subject.

II.2 Discretionary Access Control

The reference monitor concept, which we first discuss, is relevant to both mandatory and discretionary access control. We then discuss issues relating to the latter only.

II.2.1 Reference Monitor

The *reference monitor* concept is described in [Anderson 1972], and is a fundamental basis for the Trusted Computer Evaluation Criteria [DoD 1985]. We therefore use it as a starting point to model an access control system. The function of a reference monitor is to enforce the authorised access relationships between users and target objects of a system. It is a trusted component of the system. All operations requested to be carried out on an object are intercepted by the reference monitor, which only invokes the operation if the access is to be allowed. See figure II.2.

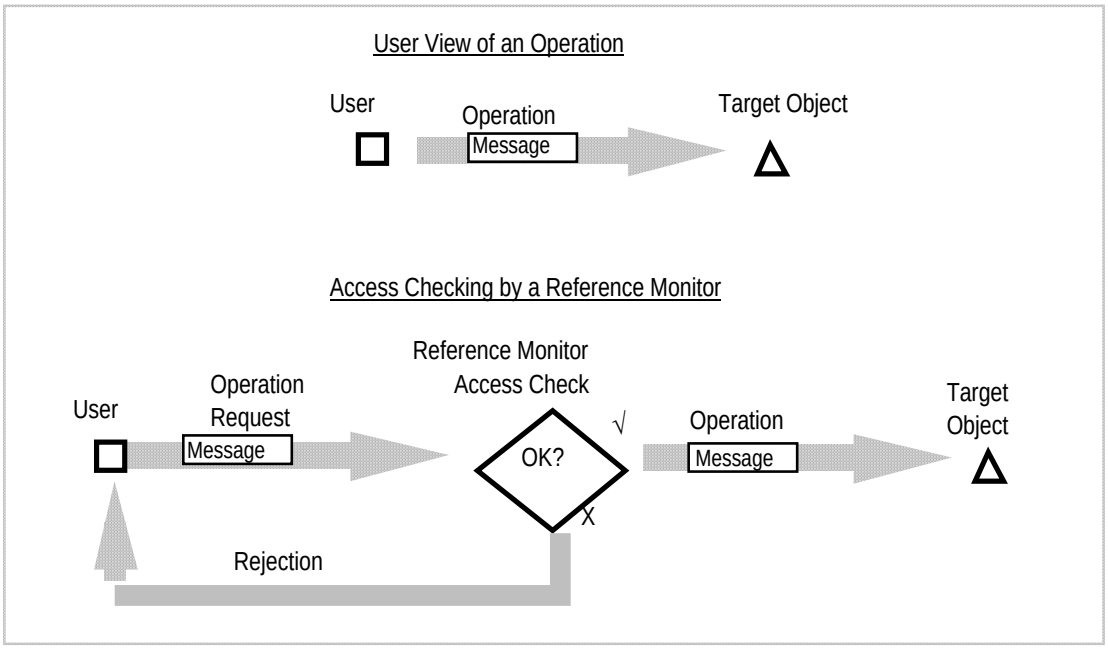

Figure II.2 Reference Monitor Access Check

The figure illustrates several points. Access control is carried out by the system, not by the object itself. The operation of the reference monitor is transparent to the user provided the access is authorised. If the access is not authorised, the operation is not invoked and the user is informed by a failure message. Systems vary, as a matter of policy, in how explicit they are in explaining the reason for failure; most systems inform the user that there has been an access violation but some, in order to confuse potential malicious access attempts, merely inform him that the access has failed without explaining why.

[Anderson 1972] states that the design requirements for a reference monitor mechanism are that it must be tamper proof, it must always be invoked and it must be small enough to be subject to analysis and testing, the completeness of which can be assured. The reference monitor concept has been refined in [ISO 1989c] into two separate components, an Access Control Enforcement Facility (AEF) and an Access Control Decision Facility (ADF), where the AEF intercepts the operation request, asks the ADF for a decision on whether the request is authorised and then enforces it. See figure II.3. This separation is useful when considering implementation, particularly for capabilities (see section VI.3.4).

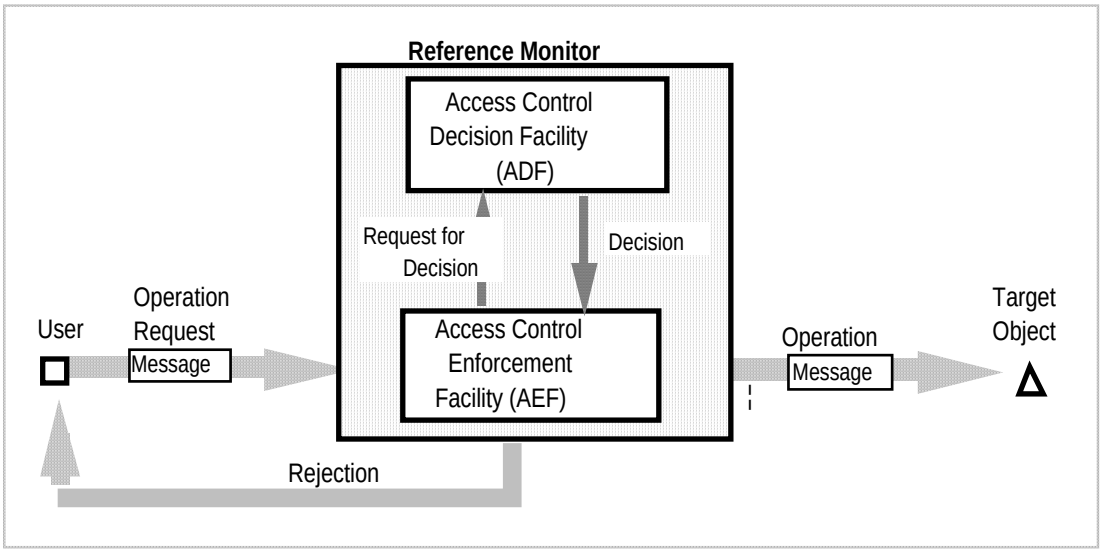

Figure II.3 Reference Monitor Refined by AEF & ADF

II.2.2 Reference Monitors in a Distributed System

Hierarchical Protection

Estrin has pointed out that more than one level of protection may be required in a system. Her papers [Estrin 1985, 1987] on Controls for Interorganization Networks are concerned with the need to allow outside users limited access to an organisation's resources. She describes the problem as one of enabling organisations to share logical networks in order to share computing resources, and shows an essentially symmetrical picture of organisation networks. We prefer to view the problem as one of access control for each individual organisation, and present her requirements in those terms.

MIT is an organisation with a number of computers which are connected by a network. Users can gain access via the organisation's own terminals or by external terminals via a gateway. See figure II.4. The requirement is to enable selected external users from ABC and XYZ to have access to certain of these computers while preventing their access to other

computers. Estrin proposes to meet this requirement by the creation of 'logical networks' which are subsets of the MIT network which ABC & XYZ may use.

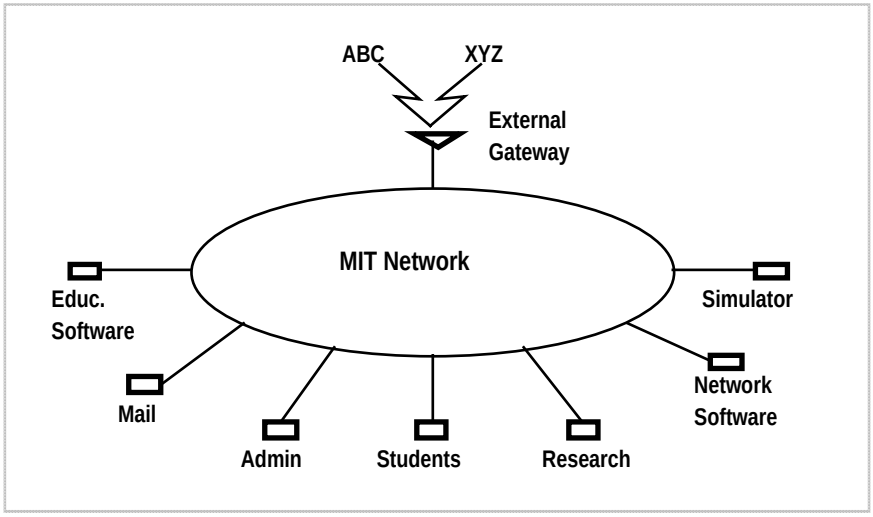

Figure II.4 MIT Computers and Network

We view the problem as one of access control; protection is required at more than one level. The installation policy will be to prevent external users both from using the private network at all and also from using specific computer systems and servers, without explicit authorisation in each case. This is in addition to the protection of individual target objects which may be accessed through a (protected) server. Thus an operation request from an external user may pass through a hierarchy of checks before it is carried out. These can be represented as reference monitor checks on operations requested on various types of object, as shown in Table II.1

Description	**Object Type**	**Operation**
Access to Installation Network	Network	Connect
Access to a computer system	Computer	Log_On
Access to a Server	Server	Use
Operation on a target object	target object	exported operation

Table II.1 Required Reference Monitor Checks

The network, the computer system, the server and the target object are all regarded as individual objects on which an operation is requested, and therefore four separate access checks are required. The operation requests may be explicit or implicit, depending on the

implementation; for example there may be an implicit request to connect to a network associated with an explicit log on request, and an implicit request to use a server associated with an explicit request for an operation on an object controlled by the server. Although we have represented all these requests as being similar, it is clear that in any implementation both the representation of the request and the method of access checking will be different for the different requests.

Untrusted Reference Monitors

So far we have assumed that there is a single reference monitor trusted by both user and object. However, a characteristic of distributed systems is that cooperating agents do not necessarily trust each other, and this may extend to the reference monitor. The owner[2] of the target object on which an operation is requested may not trust the same reference monitor as the owner of the user requesting the operation, or there may be circumstances in which the owner of an object is prepared to authorise an operation request but the owner of the user is not. For example the owner of a server object may allow unrestricted access to a service for which a charge is incurred but the owner of users using the service may wish to restrict its use to selected users.

We must assume that in the general case there is no global reference monitor through which all messages are validated. Therefore the user's owner may require a reference monitor to validate an access request, separate from the reference monitor required by the object's owner. See figure II.5. A mutually trusted authentication service is required, so that the two reference monitors may authenticate each others' messages.

[2] Informal use of the term 'owner' to denote a controlling agent

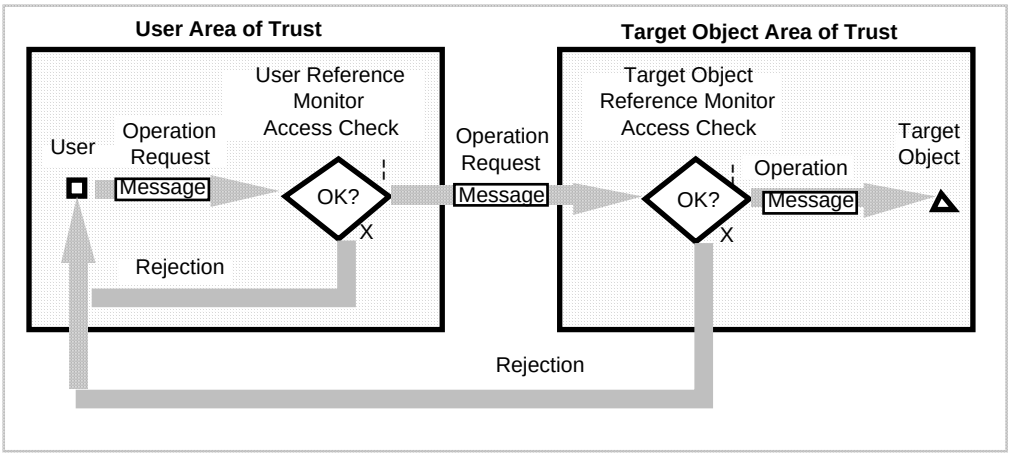

Figure II.5 Separate Reference Monitors for both User & Target Object

Note that the above discussion makes no assumption about the method of validation of access requests which either reference monitor uses. They may use completely different methods of access control. Also, each may require a different trusted method of authenticating the identities of users. The 'user' reference monitor may be able to rely on the system's validation of the user's identity at log on time, while the 'object' reference monitor may require an identity which has been authenticated by a mutually trusted authentication service, e.g. Kerberos [Steiner 1988].

Message Protection

Another consideration in distributed systems is that the communications medium between user and object may be insecure. In that case the integrity of the message conveying the operation request cannot be relied upon without additional precautions. A message arriving via an insecure medium may have been falsified, either in its originator or its contents, or be an unauthorised replay of a previously valid message.

In this case the origin and contents the message must be authenticated. Since the user or object is not necessarily co-located with its reference monitor, message authentication may be needed for any of the message transitions shown in figure II.5:

- User to User Reference Monitor
- User Reference Monitor to Target Object Reference Monitor
- Target Object Reference Monitor to Target Object.

The method of authentication is not discussed further in this thesis. It is assumed that a message authentication service, e.g. one based on ANSI standard X9.9 [ANSI 1986], will be used.

II.2.3 Models of Discretionary Access Control

In the field of access control, the greater part of the work has been concentrated on mandatory access control in order to achieve a multi-level secure system as defined in [DoD 1985]. Relatively little has been done in the area of discretionary access control.

Access Matrix

The starting point for our discussion of discretionary access control is Lampson's *access matrix* [Lampson 1974]. Using the assumption of zero global default access authority the matrix defines for each user-object pair in the system the operations which the user may perform upon the object: the reference monitor will allow any operation whose name is in the corresponding cell and refuse it if the name is absent. Figure II.6 shows a portion of an access matrix.

Target Objects / Users	Obj1	Obj2	.....	ObjN
User1	Read	Read, Write	.....	Read
User2		Write	.....	
.....	.....	.....	.....	.....
UserN	Read		.....	Read, Write

Figure II.6 Portion of an Access Matrix

The access matrix, unmodified, suffers from three major disadvantages as a model for access control.

The first is that for any large system, whether centralised or distributed, it is impracticably large. Compression and grouping techniques, of which the use of domains is an example, can alleviate this problem.

The second is that it makes the assumption of global knowledge of the users and objects which are in the system. This assumption is untrue for distributed systems, so the full matrix can never be constructed.

The third is that, although there are particular cases where it is desired to give one or more identified users access to one or more identified objects, this is not the general case. The more typical case for the use of access control in management is where a manager, as part of his formulation of policy, wishes to give a domain of users access to a domain of objects. The manager may not know, even at the time that he formulates the policy, what are the members of those domains. He certainly cannot predict, in many cases, what will be the members of those domains in the future.

We require several characteristics for an access control model which are lacking in the access matrix:

a) The ability to manage large groups of objects in a uniform way and to structure those groups, in order to manage large scale systems.
b) No need for global knowledge of the state of the system.
c) No need for changes, as far as possible, when the individual users and objects in the system change. We call this 'timeproofing'.
d) The ability for managers to specify access in the terms of roles and groups which can be mapped onto individuals and objects, i.e. domains of control.

We also want to retain two important characteristics of Access Matrices:

e) The relating of identified user objects to access control, which therefore provides a basis for accountability (see II.1.4).
f) The ability to specify access control at the level of individual, as well as groups of, users and target objects; access control is used in many systems by individual users as a means of protecting and sharing individual resources such as directories and files.

Other Abstract Models

Most models of discretionary access control use either ACLs and Capabilities, discussed below.

One abstract model which contains a substantial discretionary element is [Stepney 1987]. This models access control in terms of Principals (users in our terms), Authorities (users with authority) and Servers. Authorities can generate Statements about access rights and may be relied upon by other Authorities who make access control decisions on behalf of Servers. Statements may contain both discretionary and mandatory elements. Distribution is considered, by means of trust relations, but not the problems of scale and grouping.

[McLean 1990], in a general discussion of the modelling of computer security which concentrates on mandatory aspects, provides a place in his framework for discretionary security but does not go into any detail.

II.2.4 ACLs and Capabilities

Lampson's access matrix is most easily implemented by storing it in distributed form, associated either with user objects or with target objects. If lists of pairs of target object identities and operations are associated with user objects, the pairs are called *capabilities*, while if lists of pairs of user object identities and operations are associated with target objects, the lists are called *Access Control Lists (ACLs)*. An example of simple ACLs is in Unix, where each file has associated with it a set of file permissions; for each of Owner, Group and Public it is recorded whether or not Read, Write or Execute is permitted.

The use of either of these mechanisms preserves points e) and f) from our discussion of the access matrix, while making some contribution towards dealing with the limitations:

a) Problems of large scale systems may be eased by a mechanism such as Group permission for ACLs. However, since there is an ACL associated with each target object, the security administrator of a large scale system still has the problem of dealing with tens of thousands of separate items, which it is impossible to handle individually. Suitable tools may ease, but do not remove, the problem.
   Identical considerations apply to capabilities, when considering the management of large numbers of users.
b) Both ACLs and capabilities remove the need for global knowledge.
c) Some implementations of them may provide 'timeproofing' by the intelligent use of defaults and their limited grouping abilities. For example the Unix group permission remains valid as users move in and out of the group; and a default access permission which applies to all new files in a directory would remove the need to set the permissions for every single file individually.
d) Neither ACLs nor capabilities can enable flexible specification of access permissions in relation to roles and groups of users and target objects.

So neither approach, viewed as a complete model of discretionary access control, fully satisfies the requirements which we have stated. We do not, therefore, discuss them at length here as access control models. A comprehensive survey is provided in [Landwehr 1981]; there are a number of more recent implementations, e.g. the Andrew system [Satyanarayanan 1989] uses ACLs with protection domains and Amoeba [Mullender 1990] uses capabilities. Both ACLs and capabilities may be considered as implementation mechanisms for access rules, and are discussed as such in chapter VI.

II.3 Domains

An approach to the problems of very large distributed systems is provided in [Robinson 1988a & b], which introduce the concept of *domains* as a means of managing systems. Management of systems implies being able to take a view, not of objects, but of groups of objects. He defines domains as named groups of objects to which a common policy, such as access control, can be applied. His work was relatively implementation-specific, and one of our motivations is to generalise it for use as a management tool for specifying access control policy.

Domains are a fundamental concept in our view of systems. Apart possibly from root objects, all objects are created within domains, and they are the means used to structure systems so as to be manageable. They are described in detail in [Sloman 1989], but are summarised here.

Domains can be used for several purposes:

- Referring to a collection of objects as a single named group.
- Defining hierarchically organised groups of objects, either to reflect existing hierarchical organisations or to organise very large groups of objects.
- Relating individual objects to roles or positions.

II.3.1 Domains as the Scope of Management

Management requires the ability to perform operations on defined resources. The scope of this is not unlimited, either in the range of objects over which it extends or in the operations which can be performed on them. In our model we can simply enumerate the operations but a more powerful means of representation is needed to describe the range of objects covered. Using domains we can achieve both flat and hierarchical grouping, and also refer to user objects indirectly through the roles and positions which they occupy. For example domain hierarchies can be used to describe the structure of an organisation and the scope of authority of managers over resources and other users. This is illustrated in detail in the examples in chapter V.

II.3.2 Object Oriented Approach

The Domains concept relies on an object oriented approach, which we use throughout. Everything, including access rules themselves, is treated as an encapsulated entity presenting a set of operations as its interface. It is recognised that actual implementations will not take such a uniform approach. In particular access rules are unlikely to be implemented as objects, but will be entries in lists or databases. However, there are conceptual gains from

taking a uniform approach at this broad specification level, and questions of representation can be delayed until implementation.

'Type' is used in a very broad sense here. Two object instances are said to be of different Type if their external interface or specified behaviour differ in any respect.

II.3.3 Definition of Domains

A *domain* is an object, of type `domain`, whose purpose is to represent a set of objects which may be resources, workstations, modems, processes, etc, depending on the purpose for which a particular domain is defined. One attribute of a domain is its *object set*, which is a set of identities of objects which are referred to as *domain members*. A dual attribute of every object is its *Domain Set*, the set of identities of domains of which it is a member.

Objects are capable of being members of more than one domain at a time. Domains are persistent even if they do not contain any objects - it must be possible to create an empty domain and later include objects in it.

Domains do not encapsulate the objects themselves - managers or external objects may interact directly with an object in a domain. The objects in a domain do not all have to be of the same type, but the domain maintains information on the interfaces to be supported by objects in it. There is a constraint on domain membership that all objects in it must support the minimum operations required.

II.3.4 Operations on a Domain

This outlines the operations which can be carried out directly on a domain.

Create or destroy an Object - All Create and Destroy operations are viewed as operations on their containing domains. Domain objects can be created or destroyed like any other objects. Creation both creates an instance of an object and adds it to the object set of a domain, while Destroy both removes the object from the domain's object set and destroys the object instance.

Include Object in a Domain - includes an object into a domain by adding its identity to the object set of the domain. It does not affect the state of the object or its membership of other domains, except that the object updates its Domain Set. Domain objects can themselves be included in other domains, and are then referred to as *subdomains*.

Remove Object from a Domain - removes an object from a domain without affecting the state of the object, by removing its identity from the object set.

List Objects in a Domain - returns a list of the members of the object set of a domain.

Since domains contain information on the interfaces to be supported by objects in them, operations are also needed to change this information.

II.3.5 Domain Set Relationships

The possible set relationships between domains are important both in creating access rules and in understanding their effect.

Disjoint - Two domains are defined to be *disjoint* if their object sets are disjoint.

Overlap & Subset - 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.

Subdomain - The representation of hierarchical organisation can be accomplished by creating a domain object, referred to as a *subdomain*, as a member of another domain, see figure II.7. We refer to D2 as a *direct subdomain* of D1, and to D4 as an *indirect subdomain* of D1. Also we refer to D1 as a *superdomain* of D2, D3 and D4.

An object is an *indirect member* of a domain Dx if it is a member of a domain Dy which is a member or indirect member of Dx.

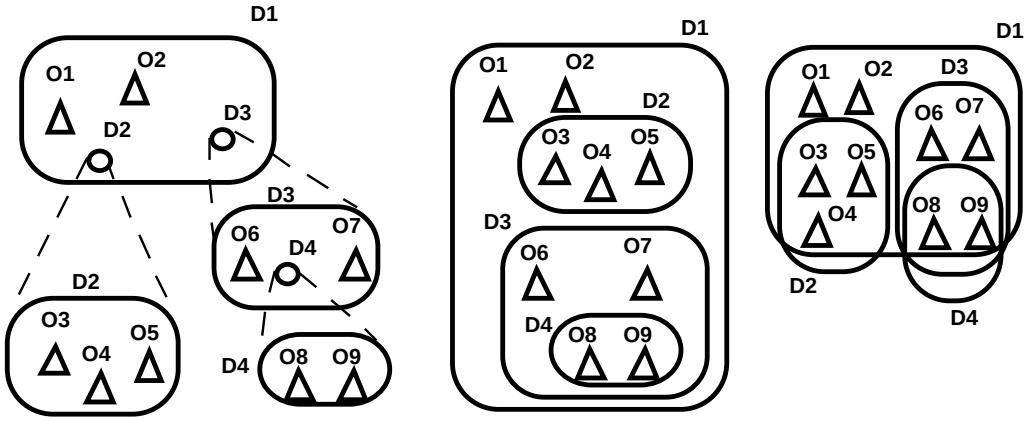

Figure II.7 Subdomain and Subset Relations

An alternative notation can be used as shown in figure II.7.b. It is a simpler way of showing a subdomain relation, but it is ambiguous, as it can also be seen as a subset relation. In a static situation the distinction is unimportant, as the evaluation of the domain set membership has the same result. However, the effect of removal or inclusion of an object in D2 will have

different results for the evaluation of the membership of D1, depending on the sort of relation. We use the notation of figure II.7.b in this thesis for compactness. In all cases, except where noted, it denotes the subdomain relation.

II.3.6 Related Work on Domains

The use of an access matrix which incorporates domains is described in [Linden 1976]. In our terminology, these domains correspond to a set of capabilities for a particular user.

There are a number of groups in the USA working on security management who also use the concept of a domain for distributed system management. Estrin's inter-organisational networks [Estrin 1987] correspond roughly to a physical domain with a common set of access rights. [Nessett 1988] introduces the concept of separate source registration and destination domains for authentication. This is developed by [Gomberg 1987], as a means of controlling logon across distributed systems.

As noted above, [Satyanarayanan 1989] uses ACLs with protections domains for the Andrew system. Subdomains are defined, and access rights are inherited from their superdomains.

II.4 Management Concepts

We discuss here the management concepts which underly our view of the delegation of authority.

II.4.1 Responsibility of Human Users

It is assumed that there are human users of the system, and that humans are ultimately responsible for the actions of the system. In many situations they will use automated agents to carry out this responsibility, but they will retain responsibility for the results, and are therefore required also to retain the power to control the agents. Thus we will always be able to trace responsibility back to a human user, who we call the *owner* of the system.

II.4.2 Ownership

Ownership in this thesis is intended to denote a concept as close as possible to normal legal ownership of goods and property. All computer systems in a country such as the UK process resources with identifiable legal owners who have legal powers over them. We regard ownership as the starting point for delegation of authority.

As a very rough approximation, ownership of an object implies the legal power to perform any feasible operation on it, together with responsibility for the operations which are

performed. There are of course limitations to this power; for example an owner of personal data may not disclose it except under the terms of data protection legislation, and the owner of petroleum spirit at a bulk distribution plant must ensure that an automated system dispensing it to tanker lorries does so in accordance with certain safety regulations. Restrictions of this kind have to be represented in computer systems as mandatory constraints, and are beyond the scope of this thesis.

An owner need not be a single person. Joint ownership is feasible in computer systems, just as it is for bank accounts, but rules (one signature, or two?) have to be agreed for exercising the powers of ownership in these cases. We may also recognise the possibility of ownership by corporations, committees and other 'legal persons'. However, we do not attempt to include corporate legal persons within the scope of this model; it should normally be sufficient to recognise the authorised representative of the legal person, such as a company secretary representing a board of directors.

Many previous authors, e.g. [Lampson 1974], have assumed that the user who creates an object automatically becomes its owner. We do not hold that view, but distinguish between ownership of objects and the delegated power to create them. A data processing clerk who submits a job to create a new version of a file of bank accounts is not the owner of those accounts, but is carrying out the task of creating the file as an agent of the owner of the bank.

We have extended the concept of ownership, for convenience, to allow one user to be the 'owner' of another. We use it identically to ownership of resources, to mean the starting point for the delegation of authority, so the 'owner' of a user is his ultimate employer, rather than an immediate or intermediate manager. The unfortunate overtones of slavery should be ignored.

II.4.3 Authority

We define *authority* as power which has been legitimately obtained, following [Weber 1948]. Other definitions are possible, e.g. [Scruton 1983], who defines it as legitimacy with or without power, and [ISO 1989b] which as we have already noted defines it to be a human agent who has the legitimate power. There are also a number of ways in which legitimacy may be obtained, but for our purpose it is obtained in accordance with the currently applicable laws. We are concerned here with particular aspects of authority as applied to discretionary access control: with the means by which authority can be granted and removed dynamically; and with the delegation of these means within an organisation in a controlled manner.

For the purpose of discussing authority we assume that we start with an owner, who can then dispose of or share his ownership or delegate a subset of his powers, to another person.

Provided that this is done legitimately we will describe this other person as having gained authority.

In a computer system authority is normally represented as the legitimate ability to perform defined operations on objects through specified interfaces.

Delegation of authority is an essential concept in large organisations, where the overall managers of the organisation cannot possibly take responsibility at a detailed level. Managers need to be be able to delegate authority to subordinates, who are given a defined subset of the manager's powers, to exercise in accordance with formal or informal policies (which are not modelled in this thesis). At one level, a divisional manager may be given the authority to run an entire section of an organisation, being required to account for his activities periodically to his seniors in macroscopic terms, such as profitability. At another level, a clerk may be given the authority to perform certain transactions, and must have every one approved manually after the event by his supervisor.

Delegation of authority must be possible in computer systems. Therefore it is necessary to be able to identify the people (or positions) to which authority has been delegated and the powers they have been given. We aim in particular to define these powers in the case of security administration, and to provide a framework for their definition in other applications.

We introduced the concept of delegation of authority in access control systems in [Moffett 1988], dealing with it rather informally and intuitively. [Yu 1989] uses the concept in a specific application. We here define it more precisely and generally, using an extended model of access rules based on the use of domains.

The distinctive factors in the management of distributed systems have been described in [Sloman 1989]. We concentrate here on one in particular: a distributed processing application may span the computer systems belonging to a number of different organisations, with no central authority. Authority cannot, therefore, be delegated or imposed from one central point, but has to be negotiated between independent managers who wish to cooperate but who may have a very limited trust in each other. They may wish to give each other access to their computer systems which is closely controlled both in the scope of the target objects which can be accessed and the operations which may be performed on them.

We need to show how the concept of delegation of authority can be extended to distributed systems in a natural manner. This can then provide a sound basis for managers to delegate access authority to members of independent organisations while retaining the maximum level of control.

Separation of Responsibilities

Separation of responsibilities is an important control concept which is familiar in the context of auditing, e.g. [Waldron 1978]. It is designed to ensure that no-one has excessive authority. [Clark 1987] has pointed out that it should be modelled in computer systems. It requires that different aspects of certain transactions should be carried out by different users, so that no one person can carry out the transaction autonomously. An example is the authorisation of payment of suppliers' invoices, where the input of invoices to a computer system must normally be carried out by a different user from the person who can trigger off the actual release of payments. Neither can carry out the other's function, so the payments cannot be made without their cooperative activity. Requirements for separation of responsibility have to be specified at an application level but support for it needs to be provided by the access control system. We may call this *horizontal* separation of responsibility because a single transaction is forced to be split into two actions.

Another type of separation of responsibilities, which we will call *vertical*, is the separation of two actions which are hierarchically related. Two examples are:

- The control requirement that the people responsible for developing computer systems should not also be responsible for operating them.
- The requirement that people responsible for granting access authority should not be able to grant it to themselves if they themselves need access. This requirement, in relation to Security Administrators, is studied in detail in this thesis.

Global Default Access Authority

We make the assumption that there is no inherent right of access of any kind. If a person is not the owner of an object, and has not been given authority, then the computer system should refuse all access.

The alternative to a global default policy of no access is one of universal access - all users have access to all objects unless they are explicitly restricted. This is a perfectly viable policy, and it is believed that the current discussion could be amended to use it with relatively few alterations. However, we are using the 'no access unless authorised' policy for this thesis because of our basic assumptions about ownership and authority which are the foundations for that policy.

II.4.4 Management

We do not attempt to define management completely. There are two main aspects of it to be considered in the management of computer systems: ability to perform management

operations and delegation of authority. Some management operations such as configuration management require the ability to operate upon groups of objects in a uniform manner, while others such as the introduction of new terminals to the system are typically done on one object at a time. For the purpose of this discussion management operations are regarded as simply another form of functional operation while the delegation of authority is access control management.

Managers

For small organisations, it is sufficient to say that the person at the top, the Owner, should carry out management operations and control the delegation of authority directly, just as he controls all the money budgets. However, in a large organisation this is impractical. The owners, whether the Government or shareholders, delegate budget responsibility to a Board of Management, who in turn subdivide both the budget and areas of responsibility to Managers, who have a budget over which they have full control. A manager who has a budget is in most respects in the same position as an owner, except that he cannot transfer control of his budget; these structural powers are retained by the owner.

The main difference in our model between an Owner and a Manager is that an Owner can pass on all his powers, while a Manager can create Security Administrators but not otherwise pass on his powers as a manager. In our model a manager has a rather restricted function. This is because we only deal with one limited aspect of management and not with the various management operations which a manager may also have.

II.4.5 Security Administration

The concept of a *Security Administrator*, also sometimes called a Security Officer or a System Administrator, is familiar in commercial computing. The main features of a Security Administrator are:

- He is not (normally) the owner of system objects.
- Limited authority over system objects is delegated to him by their owners and managers, so that he can give access authority on those objects to system users.
- Although he can give access authority to other users, he does not normally have authority to perform operations on the objects himself.

II.4.6 Policies and Rules

In general English usage, policies are the plans of an organisation to meet its objectives, taking into account the constraints on it. They are hierarchical in nature; the policy of one

level of an organisation translates into the objectives and constraints of the next level down. At some point general policies are translated into specific actions.

The theme of this thesis is the definition, not of policy, but of the specific actions which express some aspects of policy, i.e. access rules. We do not attempt to examine how a person in authority reaches the decision to make certain access rules, but only to model the way in which they can be made in a controlled manner once the decision has been reached. The modelling of general policy is a subject for further study.

II.4.7 An Example Management Structure

We provide a detailed model of the delegation of authority in chapter IV. To do this satisfactorily we need to define the organisational roles which we will be using in the model. Our organisational model recognises four different roles for its users: User, Security Administrator, Manager and Owner. We explain here informally the power that each one has; this is then modelled in terms of domains and access rules in chapter IV. Note that it is an application decision whether a user in a management role has authority over himself; we specifically allow for this to be excluded for Security Administrators, but we do not enforce it as part of the model.

These roles are powerful enough to model a large organisation and are thought to be reasonably general in their applicability. However, we are aware of organisations with a more complex set of management roles, and others which are simpler. The approach to modelling delegation of authority with an arbitrary set of roles is covered in chapter IV.

User

Normal users of the system, acting in any role other than those described below, have no inherent ability at all to grant access authority.

Security Administrator

A *Security Administrator* can give authority to users to perform operations on objects, but the scope of authority which is delegated to him may be limited: the target objects on which he can grant access authority may be limited to specific domains; and.the users to whom he can grant access authority may also be limited to specific domains. Note that if he is not himself a member of these domains he cannot grant access authority to himself; this achieves the desired separation of responsibilities between granting access and having access.

Manager

A *Manager* has a defined set of objects (users and/or target objects) over which he has authority. He can define a set of users and/or a set of target objects as the scope of authority of a Security Administrator.

Owner

An *Owner* has a defined set of objects (users and/or target objects) over which he has authority. He can give away or share ownership (by 'share' we mean create joint ownership rather than partitioning the objects between two owners), and delegate the Manager authority over these objects, to other users.

II.4.8 Related Work on the Delegation of Authority

There is little related work on the way in which authority, the legitimate power to perform actions, can be transmitted in large organisations. [Lampson 1974] and [Snyder 1981] make explicit assumptions that the authority to give access rights is bound in with the access rights themselves. [Stepney 1987] recognises that it is separate, but explicitly regard it as being outside the scope of the model. [Satyanarayanan 1989] recognises the need to control the manipulation of protection domains, but does not yet provide it.

II.5 Summary of Background and Related Work

This chapter has reviewed general security concepts and related them to the OSI and USA Department of Defense standards. Discretionary access control is related to other concepts, including accountability, authentication, integrity and trust.

The reference monitor concept is a useful framework for access control. Reference monitors in a distributed system have special features, including the possibility of lack of global trust in them and their need for message authentication in unprotected environments. The access matrix model is used as a basis for discussion of discretionary access control, but it does not meet the requirements for access control in large distributed systems. It does not easily scale up, it assumes global knowledge of the users and it does not allow users' access rights to be specified by reference to their positions in an organisation in order to achieve 'timeproofing'. The two most common access control models, Access Control Lists and capabilities, were shown to meet some but not all of the requirements. They ease but do not solve scaling problems, and they do not provide adequate flexibility. A more powerful model for access control specification is required.

A specific approach to flexible and powerful grouping of objects is the use of management domains. They provide the ability to specify groups and hierarchies of users and resources, and can also represent users' roles in an organisation.

The management concepts underlying our view of the delegation of authority are based on the responsibility of human users and ownership of resources. Authority, which is legitimate power, is derived from ownership through controlled delegation. Two important related concepts are the separation of responsibilities and the global default policy of no access. An example management structure, powerful enough to model a large organisation, requires at least Owner, Manager and Security Administrator roles together with the ability to delegate down the hierarchy. These concepts have not been covered by previous related work.

Two conclusions emerge from this review. First, there is a need for a means of specifying access control policy which allows the grouping and structuring of large numbers of users and target objects. This is introduced in chapter III, on *Access Rules*. Second, there is a need to describe and control the delegation of authority in computer systems. This is covered in chapter IV, on the *Delegation of Authority*.

III ACCESS RULES

Access rules are the method we have chosen for representing the detailed policy specifications which control the decisions which an Access Control Decision Facility (see section II.2.1) makes on operation requests. They are particularly suitable for dealing with large and complex groups of users and objects.

III.1 Definition of Access Rules

III.1.1 Access Rule Syntax

A basic *access rule* consists of three components:

- User_Domain - the name of a domain of user objects or a more general domain expression (see section III.3).
- Target_Domain - the name of a domain of objects or a more general domain expression.
- Operation_Set, defining the names of the operations which the access rule authorises. Note that we make no assumptions that one operation 'implies' another, e.g. Write permission implying Read permission.

It will be clear that this can be expanded to a portion of an access matrix.

Figure III.1 illustrates a basic access rule.

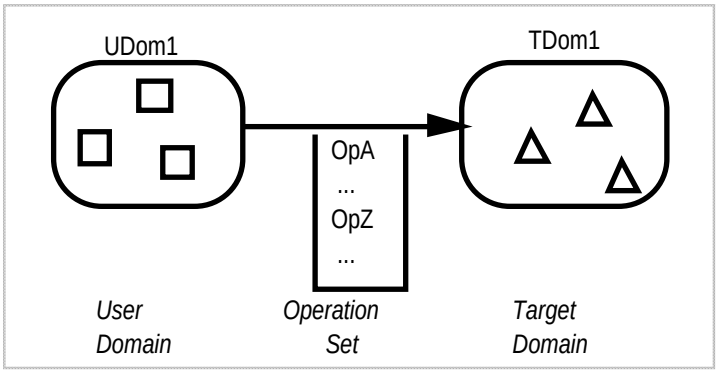

Figure III.1 A Basic Access Rule

III.1.2 Access Rule Semantics

In our model, access rules are the means by which the reference monitor determines whether to authorise an *operation request*, which is a triple consisting of (user, target object, operation name). Note the assumption which we make, that an authenticated user identity is available to the monitor as part of the message which requests the operation.

The monitor authorises a request if an access rule exists which *applies* to the operation request. An access rule applies to an operation request if the User_Domain of the rule *matches* the user in the request, the Target_Domain of the rule matches the object in the request and the Operation_Set of the rule contains the operation in the request.

A domain expression matches a (user or) object if the object identity is a member of the set of identities obtained by evaluating the expression. In the simplest case a domain expression matches a (user or) object if the object identity is a direct or indirect member of the domain (see section II.3.5).

If no access rule exists which applies to the operation request, then authority for the operation does not exist and access is denied.

For an example of applying an access rule to an operation request, see figure III.2.

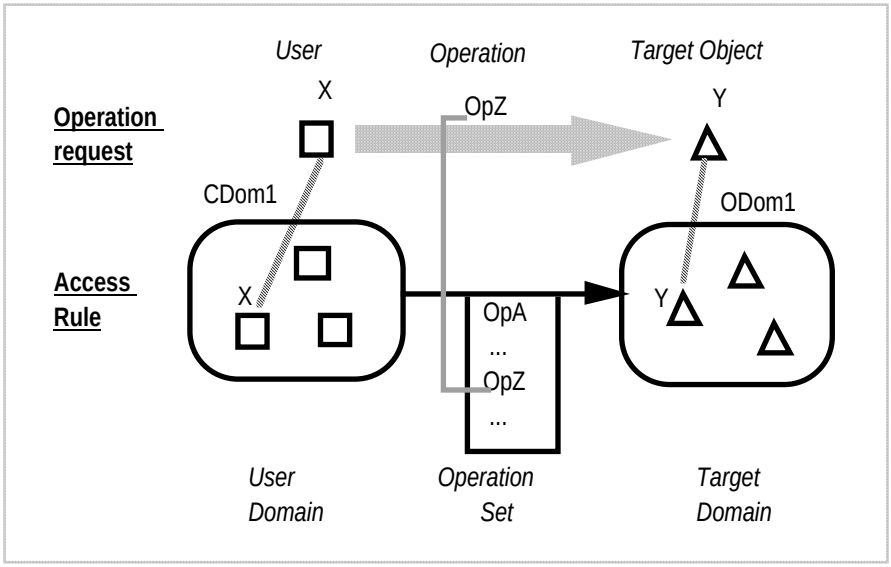

Figure III.2 An Access Rule which Applies to an Operation Request

Note that it is possible that there are overlapping access rules - more than one access rule satisfies the requirement. No conceptual difficulty arises from this, as it indicates that authority has been granted *via* more than one route. Practical difficulties may arise when it is wished to remove access authority from a user, in ensuring that there is no access rule remaining which would allow the user to carry out the operation. Approaches to this are discussed in section III.8.

III.1.3 Multiple Access Rules for an Operation

Some operations affect more than one object. In that case the reference monitor must be specified to require more than one access rule to apply before an operation request is permitted. For example the operation Domain_Include_Object (see section III.7) affects both the object to be included and the domain into which it is to be included. There must be appropriate access rules matching both (or all) objects affected by an operation before the reference monitor will authorise the request.

III.2 Example of Access Rules

We use as an example the Payroll Department of an organisation. It consists of roles for people, and a directory of payroll files, see figure III.3. We assume that the Payroll Manager (viewed as being outside the department he runs) can create whatever access rules he wishes.

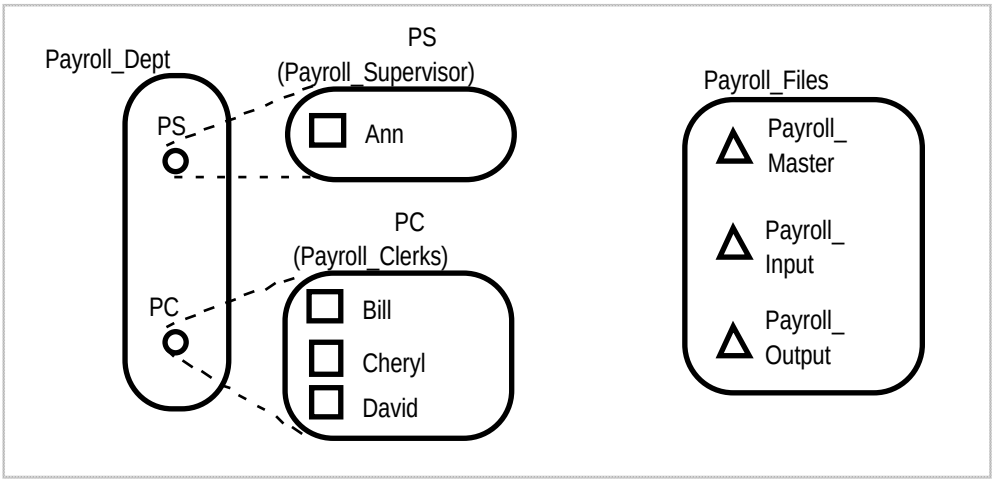

Figure III.3 Domain Structure of the Payroll Department

The Payroll_Manager may not know or understand the names of the files in the Payroll_Files domain or what are the names of the files which will be added to it in future. However, he has delegated to the Payroll_Supervisor the task of maintaining the files, so an expression of this delegation policy is to give the Payroll_Supervisor the ability to Create, Read and Write files in that domain. His policy is also to allow the whole Payroll_Dept to Read the Payroll_Files. These policies are expressed by the two access rules illustrated in figure III.4.

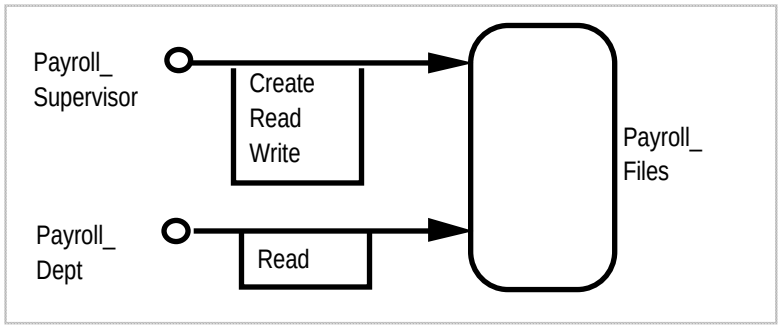

Figure III.4 Example Access Rules

The access rules do not name the users or objects, and so we cannot tell directly from them whether the access by a particular user to a particular object will be allowed. To interpret the access rule we have to know the domain structure, as shown in figure III.3.

Then we can derive a partial access matrix from the access rules and the current state of the domain structure of the Payroll Department. See figure III.5.

Target Objects / Users	Payroll_ Master	Payroll_ Input	Payroll_ Output
Ann	Create, Read, Write	Create, Read, Write	Create, Read, Write
Bill	Read	Read	Read
Cheryl	Read	Read	Read
David	Read	Read	Read

Figure III.5 Partial Access Matrix of the Payroll Department and Files Derived from Access Rules and Domain Structure

Even if the access rules do not change, the matrix will cease to be a valid and complete description of the management access policy as soon as the membership of any of the domains changes, e.g. if Cheryl is replaced by Charles as a member of the Payroll_Clerks domain or Payroll_Print is introduced into the Payroll_Files domain. A new matrix has to be constructed. However, the access rules shown in figure III.4 remain valid.

III.3 General Domain Expressions

The general definition of domains and subdomains is a powerful means of expressing hierarchical membership. It also provides a means of expressing set union - if domain DomA's object set consists of the identities of subdomains DomB & DomC, the set of identities of objects which match DomA is the union of the object sets in DomB & DomC. However, it provides no means of expressing other basic set operations such as intersection (DomB ↔ DomC) and difference (DomB \ DomC). It also provides no means of referring to individual objects except by creating domains of which they are the sole members.

It would be possible to use domain manipulation operations to create a new domain with the required membership, and refer to that new domain in the access rule. However, this is inadequate:

- The manager will require the access rule to refer not to the memberships of DomA and DomB at the time when the rule was made, but to the memberships at the time that the rule is used by the reference monitor. Therefore an access rule which refers to a static enumeration of objects at some point in the past does not meet his needs. This reason argues for the ability to specify domain expressions, incorporating set operations, in the User_Domain and Target_Domain of access rules, in order to achieve 'timeproofing'.
- For some object types, normal users may not be permitted to create new domains. Then if one user wishes to give another individual user access to some files in his personal domain (see IV.6.2), he will not be able to do so unless the recipient is already the sole member of some domain of users. This reason argues for the ability to specify individual objects in the domain expressions of access rules.

There are performance penalties associated with either of these options, and individual applications may therefore not choose them. For example the Andrew project [Satyanarayanan 1989] allows individual users, as well as domains, to be specified in their Access Control Lists (ACLs). But the finest granularity for the ACLs of file objects is the Directory.

We take the view that we should discuss the general case here, and we consider possible restrictions in chapter VI.

Domain Membership - Direct or Indirect

In the normal case an object is interpreted as being a 'member' of a domain if it is either a direct or an indirect member of it (see section II.3.5). This uses the full power of subdomains and is therefore thought likely to be used in most practical cases. However, there may be

situations in which we wish to prevent an access rule applying to indirect members of a domain. One example is when there is an access rule controlling operations on domain objects. If 'member' is taken to mean both direct and indirect membership, it would be impossible to make an access rule which applies to domain objects in a particular domain but not to domain objects in its subdomains. Referring to figure III.6, we could not make an access rule which allows the Domain_List_Objects operation on domain D5 (objects D6 & D7) but prevents the same operation on domain D7.

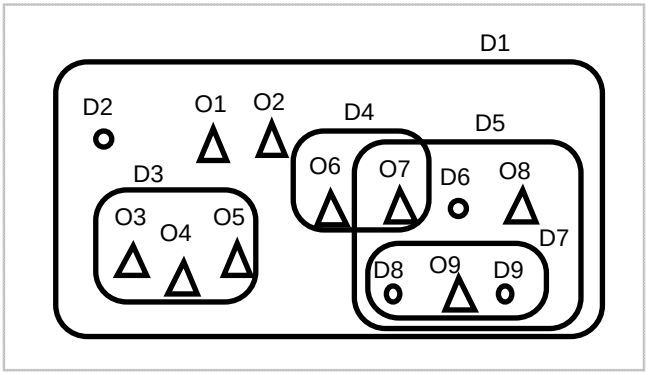

Figure III.6 Domain and Subdomains

Therefore we need a notation which allows us to specify in an access rule that domain 'member' should be interpreted only as a direct member. One possible notation is to suffix the name with an exclamation mark, e.g. the domain objects which are evaluated as members of D1 are D2 - D9 inclusive, but the members of D1! are D2 - D5 only.

Then the access rule with attributes (UserX, D1, Domain_List_Objects) would permit the operation Domain_List_Objects(D7), but the access rule (UserX, D1!, Domain_List_Objects) would refuse it, while both would allow Domain_List_Objects(D5).

Set Intersection

Use of Set Intersection in a domain expression allows the access rule (UserX, D4 ↔ D5, Read) which would allow reading of object O7 only.

Set Difference

Use of Set Difference in a domain expression would allow the access rule (UserX, D4 \ D5, Read) which would allow reading of object O6 only.

Object Enumeration

Another requirement is the enumeration of objects in a User_Domain or Target_Domain, avoiding the necessity to create domains solely for a single instance of an access rule. One or more identities of existing objects may be specified by set enumeration -

{ObjA ... ObjZ}.

Compound Domain Expressions

Compound domain expressions may be introduced by the convention that the elements of the set operations and object enumeration may themselves be domain expressions. They are defined in detail in Appendix A.2.2

III.4 Constraints in Access Rules

The description of access rules so far has viewed them as a simple relation between User_Domain, Target_Domain and Operation_Set. However, we may wish to place constraints on the applicability of access rules, i.e. even though an operation request matches an access rule in its three main components, that rule can only be applied if the constraints are met also. An additional component of an access rule is therefore defined, as a set of Constraints, each with the form:

(constraint name, expression).

The semantics of matching the expression in a parameter of an Operation Request to a Constraint in an access rule are that, if the access rule is to apply to the request, the condition specified in the constraint must be met. We assume that the reference monitor has available information about the user's environment (date and time of request, source of origin, etc), and the parameters of the operation.

Examples of possible general constraints are:

- Time of day and/or day of the week
- The terminal(s) from which the access is to be made.

The possible general constraints in access rules, their method of representation and their implementation are all subjects for further study.

III.5 Logging Switch in Access Rules

Any facility to support accountability requires input on both authorised and unauthorised operations. It is assumed that the reference monitor will provide it on all unauthorised operations, but there may also be a selective need for information on authorised operations also. It is envisaged that one means of controlling this would be by a logging switch in

access rules, which could be turned on in order to trigger the reference monitor to provide the required output.

This possible mechanism is not covered in detail in this thesis, but is a subject for further study.

III.6 Access Rules as Objects

There is a need to modify access rules dynamically during the life of the system. We model this by treating access rules themselves as objects which are instances of an Access_Rule type. An Access_Rule object has these main attributes: User_Domain, Target_Domain, Operation_Set, plus other attributes to define constraints. Access permissions are given and removed by creating, destroying and modifying Access_Rule objects. The need to control the granting and removal of permissions is achieved by imposing rules for the creation, destruction and modification of Access_Rule objects. This is discussed in chapter IV, Delegation of Authority.

Three operations are defined on Access_Rule objects: Create, Destroy and Read. A Modify operation could be defined using a compound of those three operations.

III.7 Domains as Objects

Operations on Domain Objects

Since we view domains themselves as objects, we need to allow for operations on them. We also need to distinguish, for access control, between operations on domain type objects and operations on other objects. For example, a security administrator may need to be able to perform the Domain_Read_Objects operation but be forbidden to perform file-related operations in a domain of files, while the user can perform all domain- and file-related operations in that domain. See figure III.7. In this figure, the access rule for the Security_Admin only gives him the Domain_Read_Objects operation on objects in DomA, so he cannot perform any operations on files in the domain, even Read.

We therefore need to be able to ensure that the operations on domains can be distinguished from operations on any other type of object, either by having different names or by having a type parameter. We could not achieve the desired result by putting the subdomain objects themselves in a different subdomain from the files, because the subdomains themselves contain files to which the security administrator would then have access.

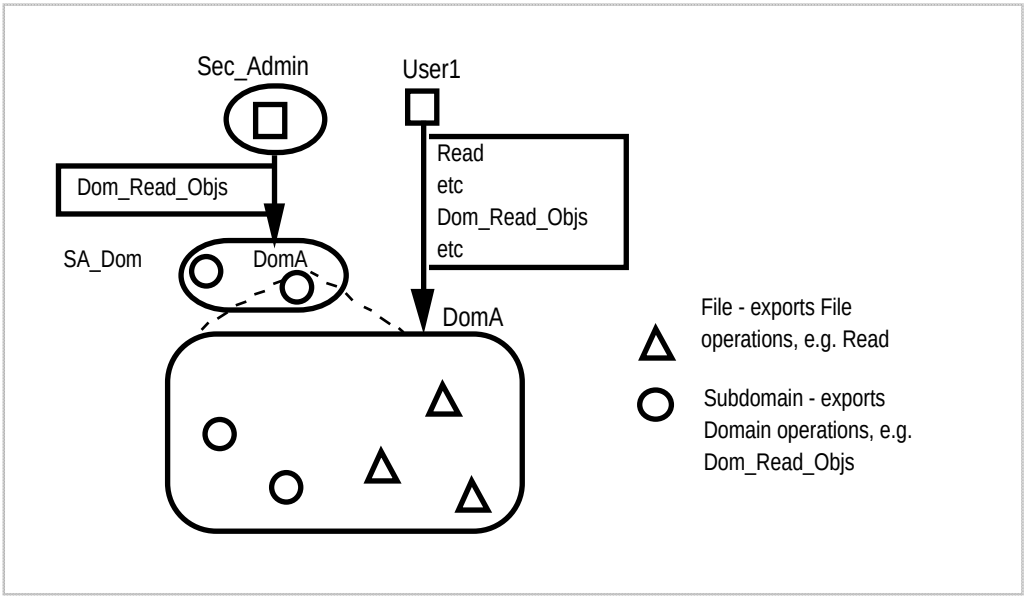

Figure III.7 Access Rules for Security Administrator and User on a Domain of Files

Apart from the specific case of domain objects, distinction between different operations for different object types is not required for access control. Where necessary objects of different types can be placed in different domains.

Including and Removing Objects in Domains

The operation Domain_Include_Object affects the object to be included as well as the domain into which it is to be included; even if there was an implementation in which the object did not record its new membership in a Domain_Set attribute, the act of including it in the new domain enables other users to perform operations on it, because of access rules which apply to members of the domain. Therefore permission is required for the included object as well as the updated domain object, when there is a request for this operation.

Two access rules must apply before the operation request is permitted: one, with the operation Domain_Include_Objects, must apply to the user and the specified domain object, and the other, with the operation Alter_Domain_Set, must apply to the user and the specified object to be included.

The same considerations apply to removal of objects from domains.

III.8 Removal of Access

The prompt removal of access authority when necessary is an essential aspect of access control. There are a number of methods of achieving this:

a) It is quite simple to exclude a particular user from accessing specified target objects at the time an access rule is set up by means of a domain expression such as (Users_X \ Users_Y) in the User_Domain attribute. This excludes members of Users_Y from access which they would have been permitted as members of Users_X. Note, however, that this will not override another access rule which allows members of domain Users_Y to have access. Similarly, a Target_Domain expression may specify (Files_X \ Files_Y).

b) *Negative rights*, which override any positive rights, have been used in some systems as a means of rapidly and selectively revoking access to sensitive objects [Satyanarayanan 1989]. This greatly complicates the semantics of an access control system. [Tygar 1987] points out that inherent contradictions can arise from negative rights, without showing how to resolve them. Figure III.8 shows a situation in which User1 both has and has not got access to Obj1.

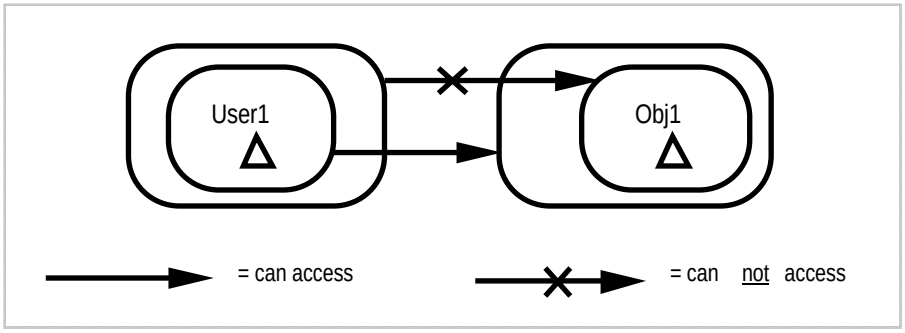

Figure III.8 Contradictory Situation Generated by a Negative Access Rule

We therefore conclude that they should not be introduced into our model and so we only permit positive authorisation.

c) For routine purposes destruction of access rules is adequate, but it will not prevent access if there is another remaining rule which permits access. Determining which access rules permit access can be complicated because typically the user or target object is a member of several domains which are subdomains of ones to which access rule refer. Tools and reporting facilities are therefore needed to permit a security administrator to determine what objects a user can access and what users can access a particular object. The use of these tools would be adequate for routine removal of access rules, but searching a large system for relevant access rules may be rather slow.

d) Denying access to an individual user in an emergency should be performed not by alteration of access rules but by suspension or deregistration of the user. It is instant and effective and does not add complications to the system.

There are thus a number of different strategies which can be applied depending on the circumstances.

III.9 Summary of Access Rules

This chapter describes our model of access rules, and the characteristics of the model. An access rule is a detailed policy specification to enable the reference monitor to decide on an operation request. It defines a set of users and a set of objects, each specified by means of a domain expression, together with the set of operations which one of the users is authorised to invoke on one of the objects. Operations affecting more than one object may require separate access rules to authorise each aspect of the operation.

Domain expressions are needed in order to specify more complex groups of objects than the members of a single domain. A general domain expression permits the basic set operations to be used, thus allowing selective exclusion of domains of objects, and several domains to be enumerated in a single expression. It also allows the restriction of interpretation of domain membership to direct members only and the specification of individual objects.

Two additional requirement for access rules are that: constraints may be needed to limit the location and/or time of day at which the access rule is valid; and a logging switch may be needed as a support for accountability.

Access rules are themselves are themselves regarded as objects, and therefore controls are required on their creation, modification and destruction. These controls are discussed in the next chapter.

Domains are also objects, and the special requirements of access rules for domain operations are: the need for restriction of interpretation of domain membership to direct members; and the need for multiple access rules to control the inclusion and removal of objects in domains.

Prompt and effective removal of access authority is an important requirement. It is not sufficient to create an access rule which does not give authority; it is also necessary to ensure that no other access rule giving authority also exists. The use of Negative Rights is a possibility, but we reject them because of their complicated semantics and the possibility of mutually contradictory rules. Extensive reporting facilities are therefore needed by security administrators to determine access rights. Suspension or deregistration of users can be done for emergency purposes.

The major issue left outstanding in this chapter is the controls on creation, modification and destruction of access rules, i.e.on the delegation of authority. These are discussed in chapter IV, *Delegation of Authority*.

IV DELEGATION OF AUTHORITY

IV.1 Introduction

Since we require that a security administrator must be able to grant access authority which he does not possess himself, the concepts of having access and giving access must be decoupled. We therefore need to distinguish between the normal operations which a user can carry out, and operations which give authority.

This chapter covers the following topics:

- Why access rules alone are inadequate to achieve controlled delegation of authority, and the need for *authority relations* to supplement them.
- An illustration of the authority relationships which implement our sample management structure for delegation of authority, and the required controls on their creation and removal.
- Discussion of possible representations of authority relationships.
- A more detailed discussion of the sample policy, using *role domains* as the method of representation.
- Discussion of alternative management structures which could be used in place of the sample we have used.

Insufficiency of Access Rules

We first discuss why we require a means of placing necessary limits, beyond those provided by access rules themselves, on the contents of Access_Rule objects which are created.

A basic requirement is that for user X to create an Access_Rule object AR1 in a domain AR_Dom (whose type constraints allow objects of type Access_Rule to be members), there must be another Access_Rule object already in existence where X matches the User_Domain, AR_Dom matches the Target Domain, and Create is in the Operation Set. See figure IV.1.

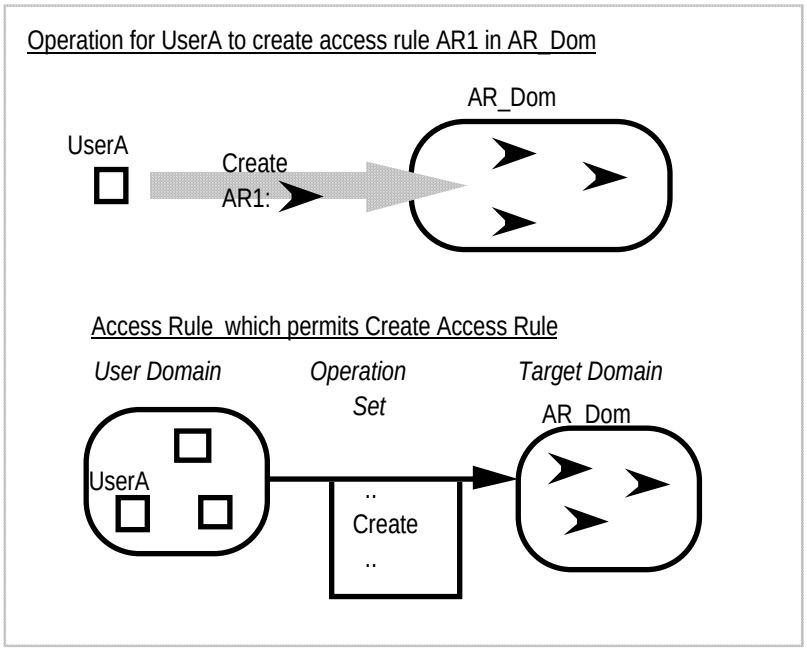

Figure IV.1 Necessary (but not Sufficient) Conditions for Creation of an Access Rule

However, the existence of an access rule allowing UserA to create an Access_Rule object is clearly not a sufficient control by itself. It provides no constraint at all on the contents of the Access_Rule object which is created, so UserA will be able to create Access_Rule objects to enable any user to access any objects. This corresponds to the nomination of him as an all-powerful security administrator, which may be the requirement for certain monolithic systems, but does not achieve the fine grain distributed control which we are aiming for.

We therefore need a means of placing controls on the contents of Access_Rule objects.

Authority Relations to Control the Giving of Access

An *authority relation* is a binary relation between users and other objects (including users). Its purpose is to express relationships concerned with giving access and our example management structure (see section II.4.7) requires different relations to express Owner, Manager and Security Administrator authority. This is in addition to access rules which give users the authority to perform normal operations.

The permitted set of authority relations and access rules and the authority which each gives are part of the mandatory policy of a system, which will need to be designed and implemented as a fixed part of the system.

The management structure and powers described below require an identity-based security policy, but do not otherwise assume any particular approach to access control. They do not

assume the use of access rules as an access control method, or even the use of domains for grouping objects.

IV.2 An Illustration

Example Management Structure

The management structure used here is that described in section II.4.7, consisting of Owners, Managers and Security Administrators. We define an authority relation with four kinds of authority. The Owner and Manager kinds define Owner and Manager authority straightforwardly. We say that a user User1 is an Owner of object Obj1 if (User1, Obj1, Owner) is in the authority relation, and similarly for Manager. Note that Obj1 may itself be a user object.

The authority of Security Administrators in this structure requires two different kinds of authority, SA_User and SA_Target, corresponding to the User_Domain and Target_Domain fields of an access rule. SA_User defines the users for whom a Security Administrator can create access rules (i.e. give access), and SA_Target defines the target objects to which he may give access. We say that a user User1 has SA_User or SA_Target authority over Obj1 if (User1, Obj1, SA_User) or (User1, Obj1, SA_Target) is in the authority relation. We need to define two separate authority kinds for Security Administrators rather than one single kind containing both user and target objects; otherwise in the case of operations upon user objects, e.g. some management operations, we could not distinguish between the users who could be authorised to perform the operation from the users on whom it could be performed.

Using the roles and policy in the example management structure in II.4.7, the authority relations give the following authority:

- If O1 is an Owner of object ObjectA he can add (O2, ObjectA, Owner) to the authority relation, where O2 is any user. O2 is an Owner of ObjectA also. ObjectA can be a target object or a user object.
- If O1 is an Owner of object ObjectA he can add (M1, ObjectA, Manager) to the authority relation, where M1 is any user. M1 is then a Manager of ObjectA.
- If M1 is a Manager of user object UserA he can add (SA1, UserA, SA_User) to the authority relation, where SA1 is any user. SA1 then has SA_User authority over UserA.
- If M1 is a Manager of object ObjectA he can add (SA1, ObjectA, SA_Target) to the authority relation. SA1 then has SA_Target authority over ObjectA.
- If SA1 is a Security Administrator with SA_User authority over UserA and SA_Target authority over ObjectA he can create an access rule which allows UserA to invoke any set of user operations on ObjectA.
- A user UserA can perform a user operation OpX on a target object ObjectA if there is an access rule which matches UserA, ObjectA and OpX.

The possible flows of control, from Owner to User and from Owner to Owner, are shown using a Petri Net diagram[3] [Peterson 1981] in figure IV.2. The Owners and Managers of the user and target object are shown as being distinct for clarity, but it is thought that more typically there would be one Owner and one Manager. Note also the tokens in the Owner condition symbols. They indicate initial conditions, corresponding to our assumption discussed in section II.4.3 that ownership is the starting point for the delegation of authority.

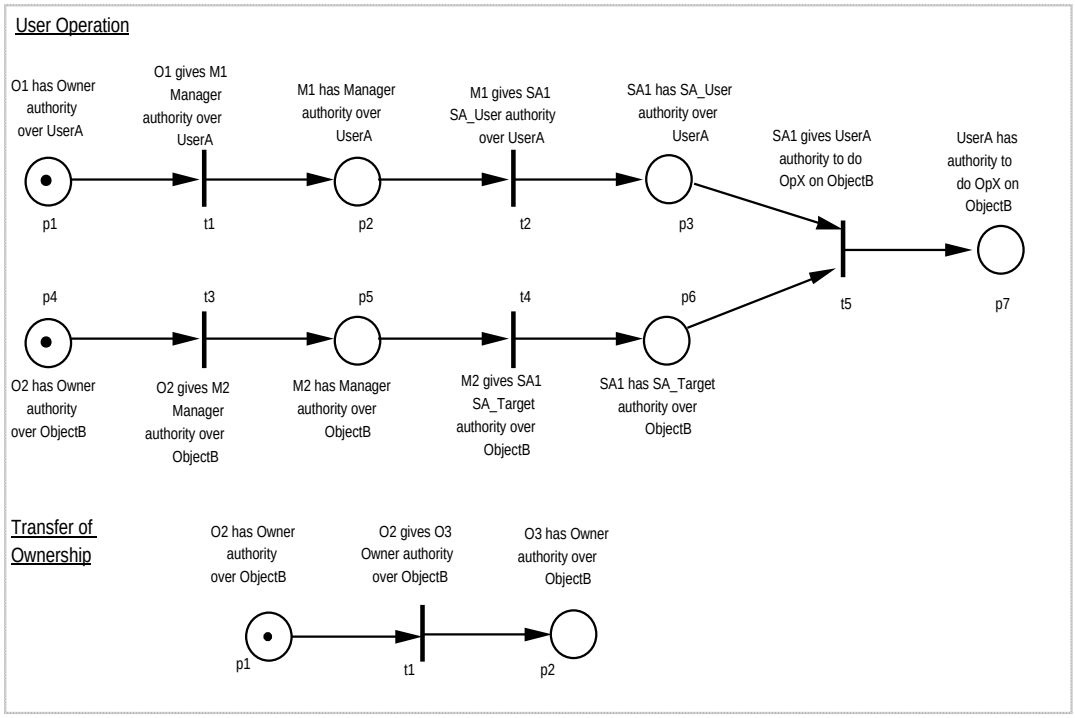

Figure IV.2 Possible Flows of Control

It is apparent that we need to recognise several additional operations in the system, corresponding to the various types of authority relation which can be created: Give_Owner, Give_Manager, Give_SA_User and Give_SA_Target, (plus the corresponding operations for the removal of authority).

[3]We differ from the standard Petri Net convention in that tokens remain in place after a transition is fired.

Operations to Create Authority Relationships

The policy which we have outlined above requires that each authority-giving operation should have controls placed on it in order to limit users' ability to carry it out. We express this by specifying the precondition for each of these operations, which is an authority relationship between the invoking user and the relevant other user and target objects. Table IV.1 shows the pre- and postconditions for each authority-giving operation. We assume that there is a policy that the same requirements apply to removal of authority.

Operation Performed by Invoking User	Preconditions	Postcondition
Give_Owner (*target object*) to *owner*	*invoking user is* Owner of *target object*	*owner* is Owner of *target object*
Give_Manager (*target object*) to *manager*	*invoking user is* Owner of *target object*	*manager is* Manager of *target object*
Give_SA_User (*user*) to *sys admin*	*invoking user is* Manager of *target object*	*sys admin* has SA_User authority over *user*
Give_SA_Target (*target object*) to *sys admin*	*invoking user is* Manager of *target object*	*sys admin* has SA_Target authority over *target object*
Create_Access_Rule (*user, target object, operation*)	*invoking user* has SA_User authority over *user* and *invoking user* has SA_Target authority over *target object*	*user* can perform *operation* on *target object*

Table IV.1 Pre- and Postconditions for Authority-giving Operations

Note that the authority-giving operations are not necessarily operations which are performed directly upon the user objects themselves. We discuss below various possible representations of the authority relation, and this will determine the representation of the authority-giving operation.

IV.3 Role Domains

The discussion so far in this chapter has been entirely in terms of single objects. One clear requirement is to be able to discuss authority relationships in terms, not of single target objects as above, but of sets of objects. We will use the term *scope* informally as a description of the set of objects with which a user has an authority relationship; for instance

the set of objects of which a particular user is Owner is described as the scope of his ownership.

There are several possible methods of representing authority relationships. We have chosen one specific method, *role domains*, for this chapter, as it is the simplest for the purposes of illustration. Other methods are discussed in section VI.4.

A role domain is a specialisation of a generic domain, with additional attributes for each kind of authority which may be possessed by a user, and type Role_Domain. In the management structure which we are using here there are four additional attributes of a Role_Domain object: Owner_Scope, Manager_Scope, SA_User_Scope and SA_Target_Scope, each scope being a domain expression. We map authority relationships onto role domains as follows: a user is an Owner of a target object if he is a member of a role domain and the target object is a member of its Owner_Scope attribute. There are similar definitions for Manager, SA_User and SA_Target relationships.

So membership of a role domain gives a user the authority associated with that domain. If a user is a member of more than one role domain he has the authority associated with each domain, e.g. someone may be the manager of more than one department in an organisation.

The x_Scope attributes define the sets of objects which are the scope of each kind of authority for members of that domain. If one of the attribute values is null, then domain members do not have any authority of that kind.

The operations Give_Owner, Give_Manager, Give_SA_User and Give_SA_Target map onto operations to add members to the appropriate scope attributes of role domains.

Note that, in addition to the appropriate managerial authority, access rules are needed to allow the user the ability to perform the operation on a Role_Domain object to alter its attributes. Throughout the examples below these access rules have been omitted for simplicity of exposition. They have of course been included in the Prolog example in Appendix B.

IV.4 Management Structure Using Role Domains

We now discuss our sample management structure in more detail, using role domains as the means of representation.

IV.4.1 Security Administrators

If a User is to create Access_Rule objects he must be a member of a role domain which gives him adequate authority via the attributes, SA_User_Scope and SA_Target_Scope, referred to

informally as a Security Administrator Domain (SA Domain for short). The User_Domain of the created Access_Rule object can contain a subset of the objects defined by the SA_User_Scope expression of the SA Domain, and correspondingly the Target_Domain of the Access_Rule object can contain the SA_Target_Scope of the SA Domain. See figure IV.3. In all of the examples which we use in this chapter the User_Domain and Target_Domain are shown as single domains for simplicity, but they could in practice be any domain expression.

The User and Target Scopes of a SA Domain are set up by the Managers of domains of users and objects as described below.

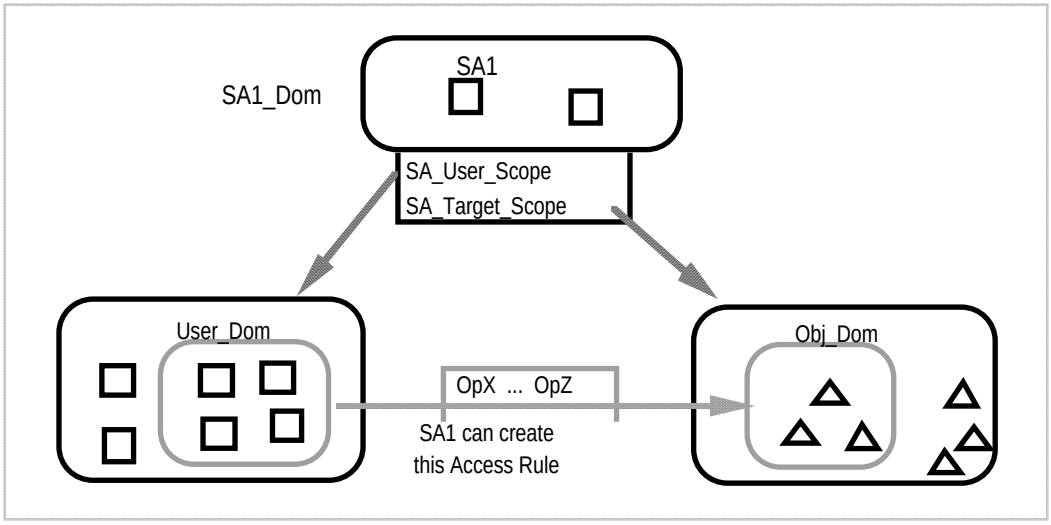

Figure IV.3 Security Administrator Role Domain with User and Object Scopes

The semantics of the checking done by the reference monitor are as follows. It must recognise a request for an operation on an Access_Rule object and go through the following checks before allowing SA1 to carry out an operation to create (in AR_Dom) an Access_Rule object which allows a user in User_Dom to carry out Op1 on an object in Obj_Dom. All of the checks must be passed if permission is to be allowed:

1. Is there an Access_Rule object with SA1 in its User_Domain, Create in its Operation Set and AR_Dom in its Target Domain?
2. Is there a Role_Domain object, say SA1_Dom, where all the following conditions apply:
   a) SA1 is a member of SA1_Dom
   b) User_Dom is a subset of the objects defined by the SA_User_Scope attribute of SA1_Dom
   c) Obj_Dom is a subset of the objects defined by the SA_Target_Scope attribute of SA1_Dom?

Separation of Responsibilities for Security Administrators

One of our main aims is to allow a Security Administrator to create access rules for other users while preventing him from giving himself access. This is done by ensuring that the membership of the role domain and of the SA_User_Scope attribute of the domain do not overlap. Then it is impossible that checks 2a and 2b, above, will be passed simultaneously, and the access rule therefore cannot be created.

Let us give a specific example, in which we assume that there is in all cases an access rule which allows Create on Access_Rule objects in a suitable domain. Suppose the domain of all users, All_Users_Dom, has two disjoint subdomains, SA1_Dom (a role domain) and User_Dom, as its members. SA1_Dom has All_Users_Dom as the SA_User_Scope and All_Resources as the SA_Target_Scope. Then members of SA1_Dom can create Access_Rule objects which give members of User_Dom access to members of All_Resources, because the checks above are passed. But a member of SA1_Dom cannot give access for members of SA1_Dom, including himself, because of the failure of check 2b. See figure IV.4.

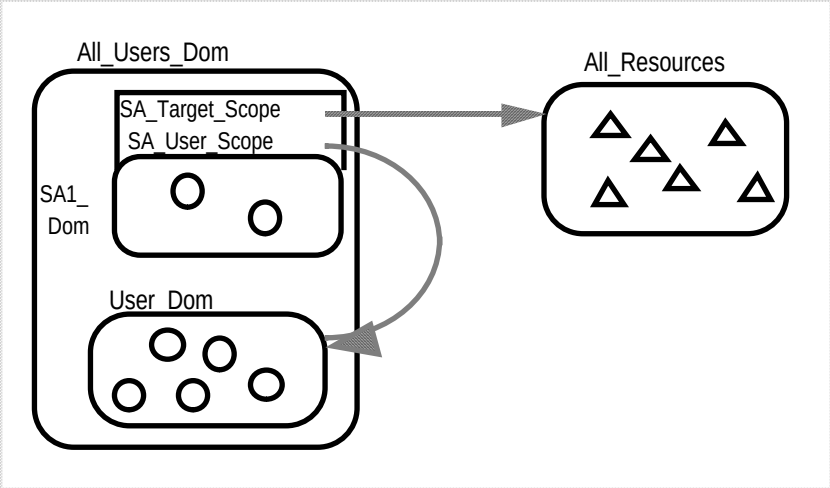

Figure IV.4 Security_Admin Scopes for Creation of Access Rules, which Exclude Himself

There is of course the need for the Security Administrator himself to have access to some objects, but he cannot create the requisite access rules himself. This need is met by the Owner creating a second role domain, which we will call SA2_Dom. Its SA_User_Scope is SA1_Dom and the SA_Target_Scope is SA_Resources. Members of SA2_Dom are then in a position to create access rules with members of SA1_Dom in the User_Domain and SA_Resources in the Target_Domain. See figure IV.5.

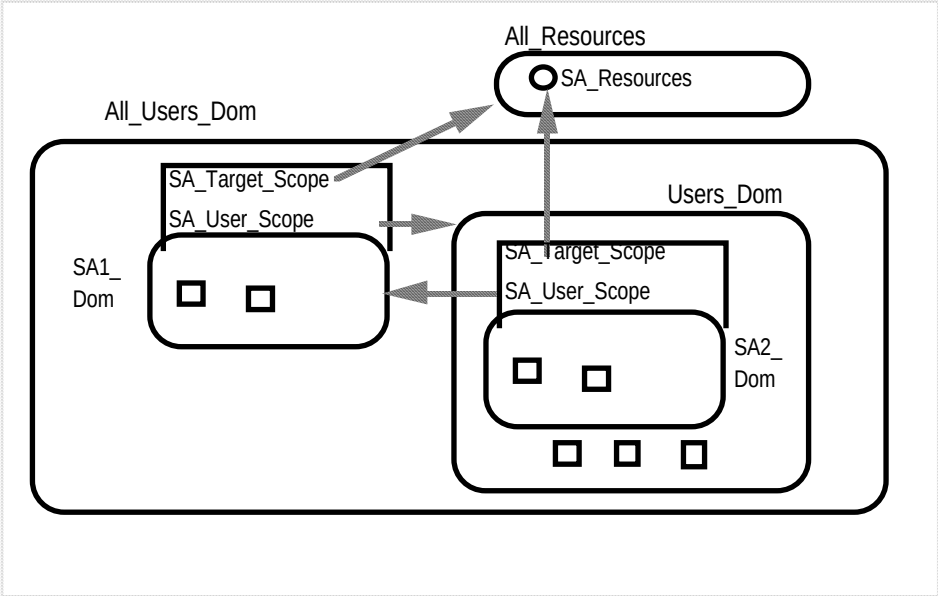

Figure IV.5 Mutually Exclusive User Scopes to Enable Creation of Access Rules for Security Administrators

IV.4.2 Managers

A Manager is defined as a member of a role domain with a non-null Manager_Scope attribute, referred to informally as a Manager Domain. He can alter the SA_User_Scope and SA_Target_Scope attributes of role domains in order to allow a Security Administrator to create access rules for other users.

Alteration of the SA_User_Scope and SA_Target_Scope attributes of a role domain is controlled by requiring the invoker of the operation to be a member of a Manager Domain. Its Manager_Scope must be a superset of the SA_User_Scope and SA_Target_Scope attributes (both their old and new values) of the role domain he is altering. So a Manager can delegate to a Security Administrator the authority to define the objects in the User_Domain and/or the authority to define the objects in the Target_Domain of an Access_Rule object. See figure IV.6. Note that we place no restriction on the objects which are members of the new security administrator role domain. We take the view, in this example, that a Manager should be able to decide to appoint any user as a Security Administrator.

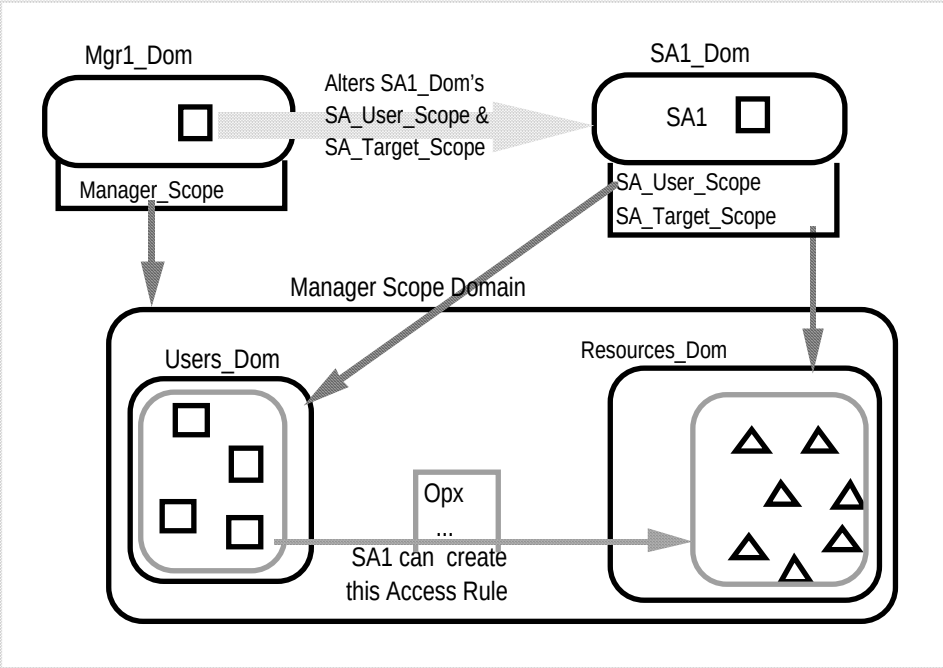

Figure IV.6 Manager Scope Allows Creation of SA Domain with User and Object Scopes

Cooperation of Independent Managers

A Manager can set both a SA_User_Scope and an SA_Target_Scope for a Security Administrator. Another possibility is that two separate Managers cooperate to create Scopes for a Security Administrator, one setting the SA_User_Scope and the other the SA_Target_Scope . For example the Manager of System_A creates a role domain, SA1_Dom, alters the SA_User_Scope attribute to specify all the users in System_A, and includes SA1 in it. The completely independent Manager of System_B alters the SA_Target_Scope attribute of SA1_Dom to specify the objects in Resources_Dom of System_B. SA1_Dom can then, in addition to any authority he has in relation to objects in System_A, create access rules which allow users in System_A to access objects in Resources_Dom of System_B. See figure IV.7. In other words inter-organisational delegation of authority can be created by the cooperation of the managers of the two organisations, with both of them sharing control only to the extent which is required.

The mechanism for achieving this is discussed in section IV.5.

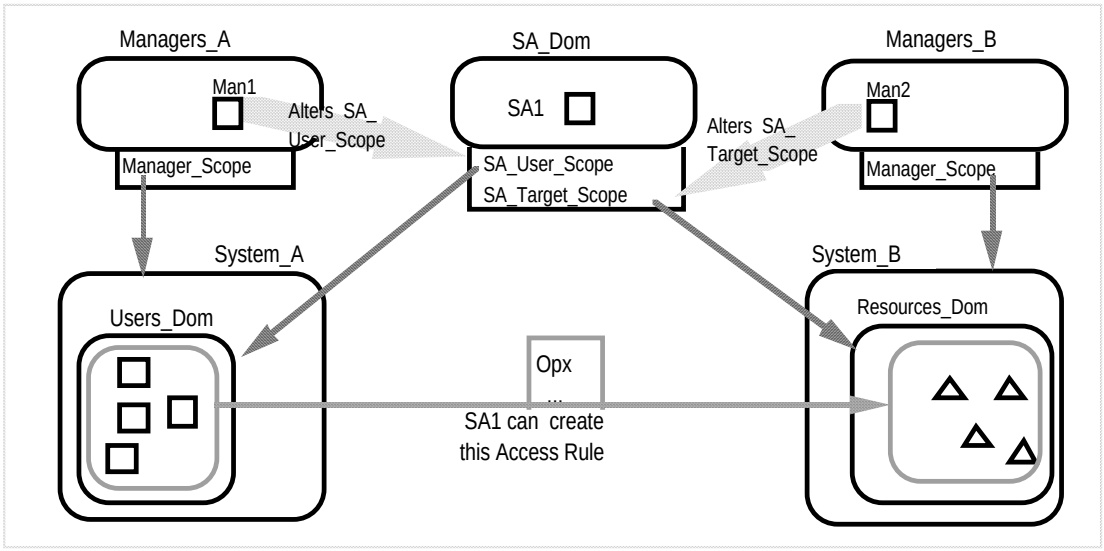

Figure IV.7 Security Administrator Gains User Scope and Object Scope from two Different Managers

IV.4.3 Owners

We have now described how role domains can be used to limit the authority of a Security Administrator to create an access rule, and how they can be set up by a member of a Manager Domain with adequate scope. The third managerial role is Owner.

An Owner is defined as a member of a role domain with a non-null Owner_Scope attribute, referred to informally as an Owner Domain. He can create role domains with Manager_Scope attributes in order to allow a Manager to create role domains for security administrators. An Owner can do the following:

- Alter the Owner_Scope attribute of other role domains, i.e. share or remove Ownership
- Alter the Manager_Scope attribute of other role domains, i.e. create or remove Managers.

Both the before and after values of these attributes must be subsets of his own Owner Scope.

The Owner Domain is exactly the same as the Manager Domain shown in figure IV.6, substituting Owner for Manager. Figure IV.8 shows how it may be used to create a Manager Domain. A similar figure could show how an Owner can create another Owner Domain.

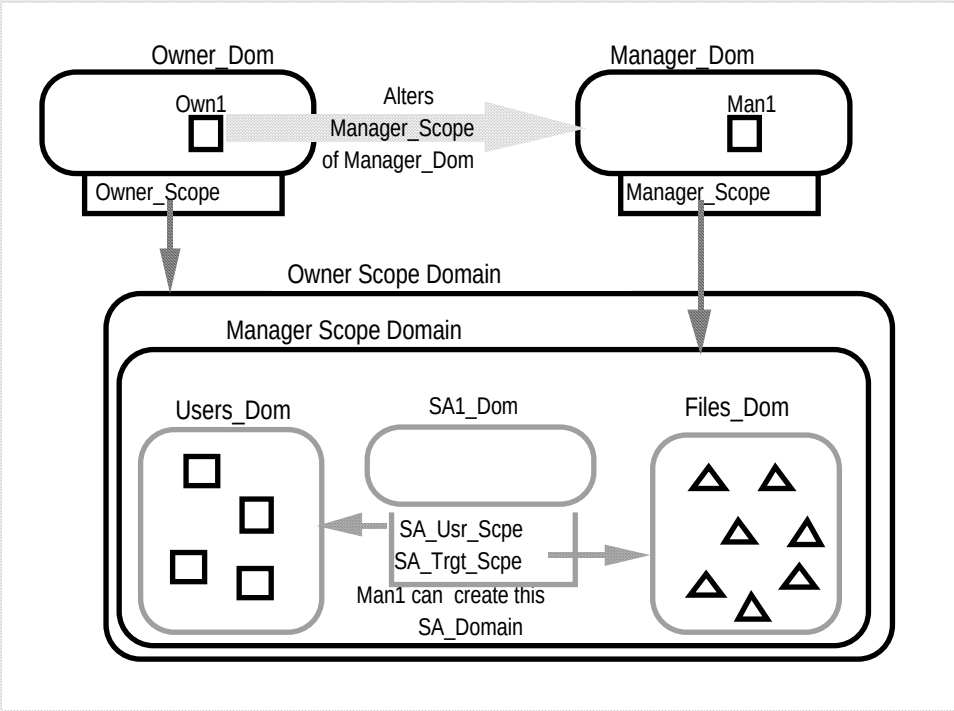

Figure IV.8 Owner Scope Allows Creation of Manager Domain

Ownership of Newly Created Objects

We remarked in II.4.2 that the creator of an object is not necessarily the owner of it. We can see how this follows automatically from our model by looking at the example in figure IV.9. It can immediately be seen that if user U1 creates a new object in Resources_Dom, the ownership of that object , like all others in the domain, remains with Owner_Dom. Indeed, U1 cannot even read the object after creation unless an access rule also allows him the Read operation

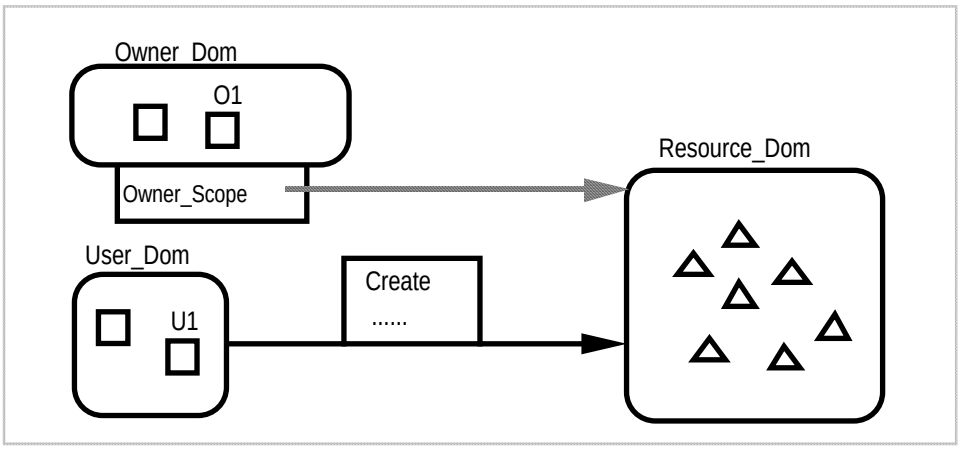

Figure IV.9 Ownership and Creation of Objects

Sharing Ownership

Sharing ownership is very simply represented: more than one user has Owner authority over an object. At this stage we also take a simplistic view of transfer of ownership, by specifying that an Owner should be able to remove Owner authority from another Owner of the same object. This allows both for suicide and for destruction races where the first person to remove the other's ownership wins power. Further constraints on sharing and transfer of ownership will need to be considered.

IV.5 Operations on Access Rules & Role Domains

We have so far discussed operations on access rules as if the only operation ever performed is creation and on role domains as if the only operation ever performed is modification. However, it is also necessary to be able to destroy and modify access rules, and create and destroy role domains.

Destruction of Access Rules

We need restrictions on the destruction of Access_Rule objects in order to prevent the unauthorised removal of authority, just as we do for creation. The conditions are exactly the same. To destroy a 'normal' Access_Rule object the user must have the appropriate SA_User_Scope and SA_Target_Scope.

Creation and Destruction of Role Domains

Operations to create and destroy a Role_Domain object are defined. No special authority is required for them, but they are created with null scopes, and can only be destroyed if all their scopes are null. Where a security administrator domain has been set up by cooperating independent managers it will not be possible to destroy it until SA_User_Scope and SA_Target_Scope have been separately nullified by the two managers.

Modification of Access Rules

We can view the modification of access rules as a compound operation of Read, Destroy and Create.

Modification of Role Domains

Separate operations to modify each of the scope attributes of role domains are defined, because each requires a different authority.

Two independent managers may wish to cooperate in setting up a security administrator domain where the SA_User_Scope attribute value is within the scope of one manager and the SA_Target_Scope attribute value is within the scope of the other. The sequence of actions to set up a security administrator role domain, SA_Dom, of this kind is:

- A user with the appropriate authority, probably the first manager, creates SA_Dom and includes selected users into it so that they may become security administrators
- The first manager, who has Modify_SA_User_Scope permission on SA_Dom and Manager Scope over the domain of users (User_Dom), modifies the SA_User_Scope of SA_Dom, to a value of User_Dom.
- The second manager, who has Modify_SA_Target_Scope permission on SA_Dom and Manager Scope over the domain of targets (Target_Dom), modifies the SA_Target_Scope of SA_Dom, to a value of Target_Dom.

IV.6 Representation of Users

We have so far made no explicit assumptions about the representation of User objects. The concepts of access rule and role domain (or other form of authority relation) are valid for any representation. We have however made an implicit assumption, which is that users, once registered on the system, are objects which persist until positive action is taken to remove them. A real-world user becomes associated with a process when he logs on and detached from it on logging off, but the system must contain a user object which is persistently associated with the user in order to retain information about him.

Also, there are some requirements for users which we have not so far considered. Many system administrations have a policy of allowing users to manage certain resources which are personal to them, e.g. files and directories which they have created themselves and are not part of any formal application. We should like a convenient means of implementing this policy. Also we need to consider a minimum configuration of users and other resources which is required to start up a new system.

For each of these purposes it turns out to be useful to consider users themselves as domain objects. There are other satisfactory representations of users, such as specialised User Profiles, but here we consider the advantages of viewing a user as a specialised form of domain. We discuss this purely from the point of view of a user's authority; other aspects are not considered. A preliminary general view of users viewed as domain objects is given in [Twidle 1988].

IV.6.1 Users as Domains

We consider user objects to be a specialisation of role domains, which we will call *User Representation Domains*, or URDs for short. URDs will only permit process objects as members, and may have special attributes such as personal details and authentication information. However, these specialisations do not affect this discussion of their use to represent a user's authority, and we will consider them as normal role domains.

A system user can therefore be seen as a URD which is bound to a user process, and thence to a human user when he logs on at a terminal or starts a batch job, and detached from the process when the user logs off or the job finishes. When a URD is not bound to a user process its object set is empty; the process is included in the object set during the activity of binding, and removed from the object set when the binding is released. A URD is created as part of the activity of registering a new user on the system.

Our view of access rules is, of course, that in general any member of a domain has authority to perform operations which are authorised for the domain itself. Therefore the user process automatically gains all the authority of the URD when it becomes a member, and loses that authority when it is removed from it. This is therefore a perfect method of ensuring that a process has all the authority of a user while, and only while, it is representing him.

IV.6.2 Users' Personal Domains

Many systems have an informal concept of a user's personal domain, where the user is automatically allowed authority to give access for other users. A typical general (mandatory) policy is that a user should be allowed to give any other user in the same organisation access to resources in his personal domain. In our terms, a user should have a Security Administrator's authority with an `SA_User_Scope` of all users in the same organisation and his personal domain as the `SA_Target_Scope`. If the user is represented by a URD, then the natural way to express this general policy is that when a new user is registered, the URD should have the values of its `SA_User_Scope` and `SA_Target_Scope` attributes set to the required values.

When a new user is introduced to the system, setting up the required attributes in his URD could be done either automatically by the system or manually by a security administrator for each new user. The latter is undesirable because:

- It is laborious

- It would mean that the security administrator would need to have Manager authority over the system's resources, which is contrary to our principle that he should be able to give authority but not to have it.

It is therefore better that this policy should be programmed into the system to be carried out by a system task which is the Manager of system resources. This is consistent with the policy's status as part of the mandatory policy of the system.

Figure IV.10 shows the position immediately after a new user has been registered. Note that we have chosen to have separate URDs and personal domains for each user, as the access authority requirements for them are different. There is an access rule giving members of the URD (the user process) authority for all operations on the user's personal domain, and the URD is a role domain with SA_User_Scope for the domain of all users, and SA_Target_Scope for the user's personal domain.

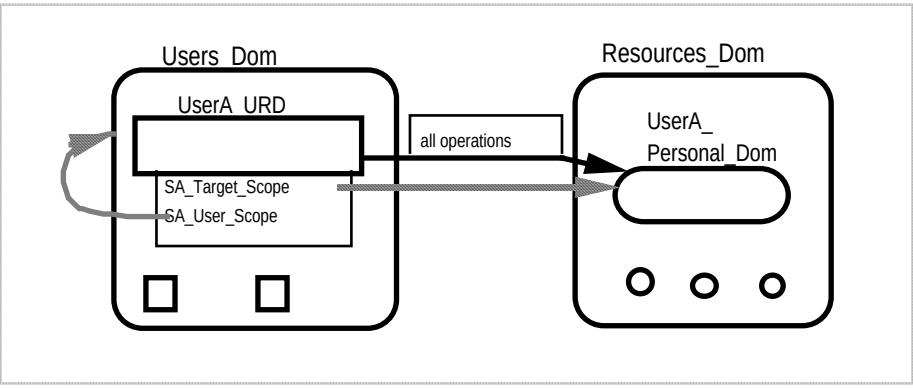

Figure IV.10 Domains and Authority for a Newly Registered User

IV.6.3 System Start-up

In starting up a new system one question which arises is, what is a minimal set of objects from which one can create the other objects which are needed to make a viable system. The following is one feasible approach assuming the representation of users by URDs.

We start with a domain, say System_Dom, containing a root user, Root_URD, which is Owner, etc, of System_Dom, and an access rule which allows all operations on System_Dom. It can create basic domains to contain other access rule objects, users and other resources. See figure IV.11

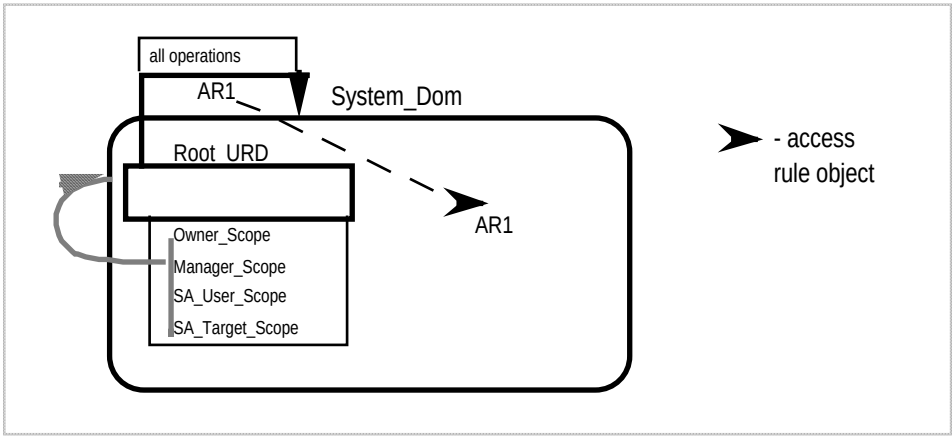

Figure IV.11 A Minimal Start-up System

IV.7 Alternative Management Structures

Table IV.1 (section IV.2) is a representation of the authority structure chosen for a particular system. It is not part of discretionary security, as it is fixed for a given system, but it may vary between systems or even for a single system, if fixed policy changes. We have discussed the delegation of authority with a particular view of a management structure: Owner, Manager and a Security Administrator who requires both SA_User and SA_Target authority. It is clear that a number of different policies could have been chosen. Some of the possibilities are discussed below.

IV.7.1 Grouping of Operations

We could partition the normal operations of the system, i.e. those whose function is not to give authority, into two or more categories. For example we could partition them into management operations and user operations. We can then decide that there will be separate policies for authority to carry out the different categories of operation, e.g. there may be a policy that management operations on an object can only be authorised directly by an object's Owner. It would then be necessary for there to be a different type of access rule for each category of operation, since the conditions to be satisfied for giving and removal of authority for operations would then depend on the category of the operation.

IV.7.2 Extension to Include Operations

A security administrator's SA_Target authority has so far been defined simply as a set of objects. In order to allow limitation of the operations which a Security Administrator can authorise, the definition of SA_Target authority can be extended to include a set of

operations also. A Manager can, for example, permit a Security Administrator only to authorise Read access. This would result in the flow of control shown in figure IV.12, where M1 and M2 may be the same manager. A simplified illustration of the type used for the rest of this section is also shown; it also illustrates that an Owner may transfer Ownership. See appendix A.8 for a formal interpretation of both the Petri net diagrams and the simplified diagrams.

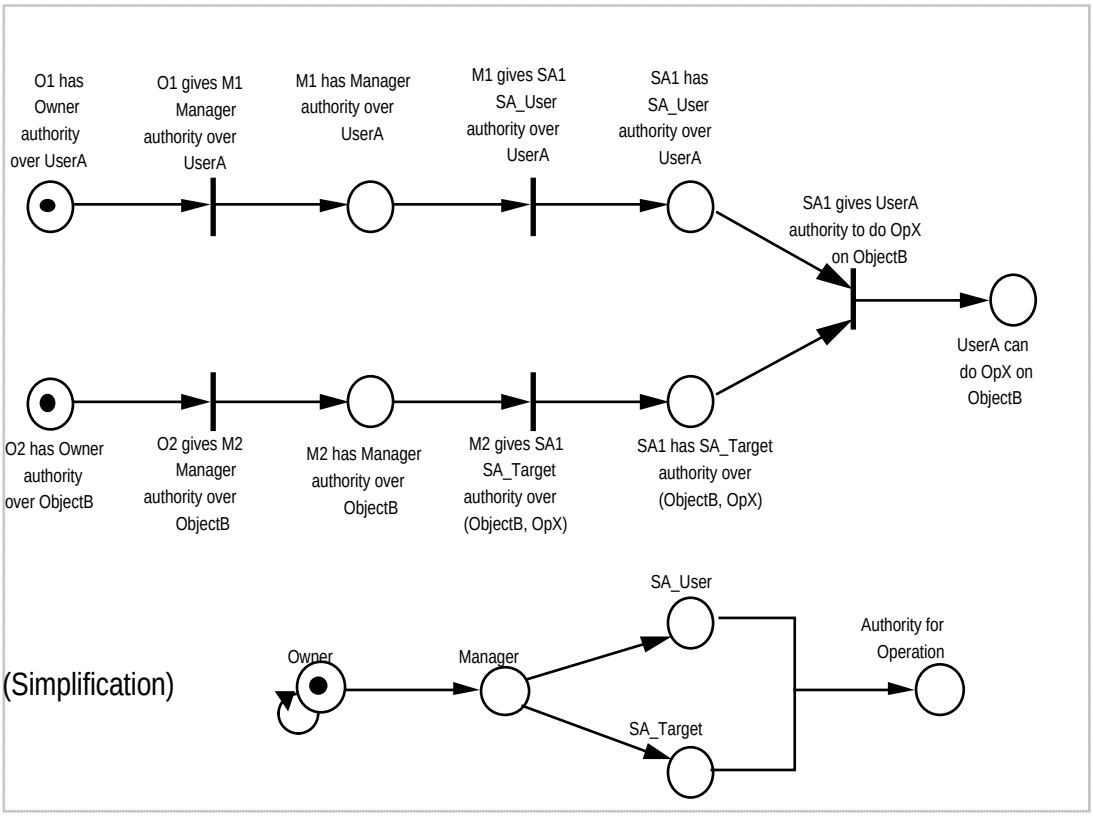

Figure IV.12 Flow of Control from Owner to Security Administrators (Allowing Limitation of Operations)

IV.7.3 Security Administrator Policies

The following alternative policies for a Security Administrator are also possible:

a) Only SA_Target authority is required, figure IV.13 (a). The Security Administrator acts for all of the users in the system, including himself, so we have lost the ability to prevent him giving himself access if we use this policy.

b) No limitation on a Security Administrator at all. This corresponds to a Superuser in a monolithic system, figure IV.13 (b).

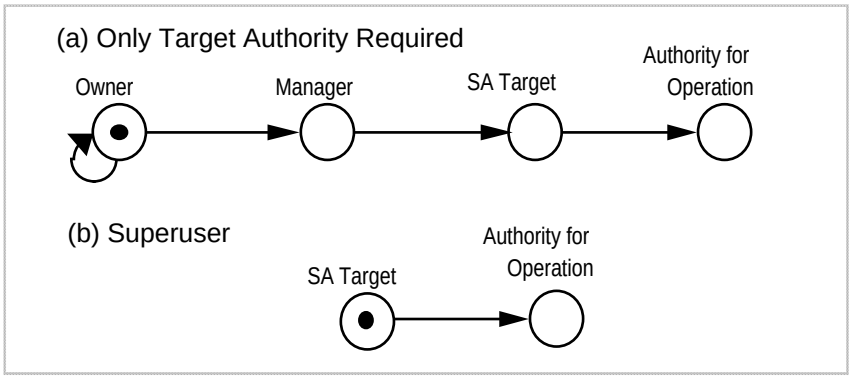

Figure IV.13 Other Policies for Security Administrators

IV.7.4 Managers & Owners

A Manager's authority could be separated into Manager_User and Manager_Target authorities, figure IV.14 (a).

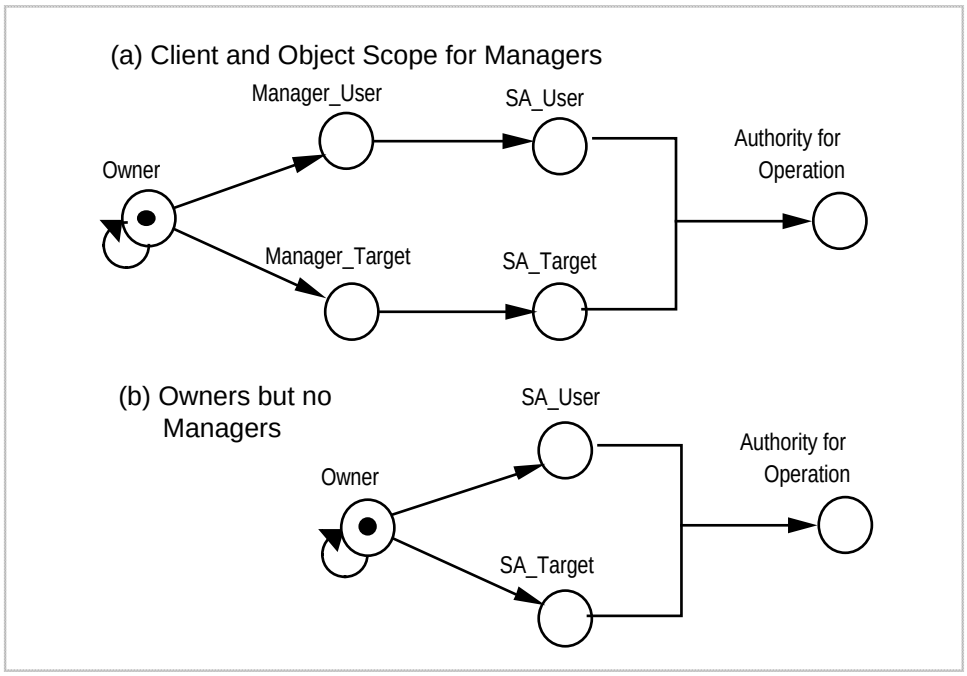

Figure IV.14 Other Policies for Owners and Managers

The policies we have outlined here all assume that Managers can only be created by Owners. There could be a hierarchy of Managers, each able to delegate authority to the level below. A fixed hierarchy can be incorporated into our model. e.g. by defining Manager1, Manager2 and Manager3 levels of authority. Note the effect, in our model, of allowing a manager to give another manager authority at the same level; we have assumed that the giving and removal of authority are treated similarly, and therefore if User1, with Manager*n* authority

over an object, can give User2 the same level of authority over it, User2 can subsequently remove that authority from User1.

The policy may provide for only Owners, but not Managers, figure IV.14 (b).

IV.7.5 Generalisation of Flow of Control

The examples above are all special cases of a more general representation of flow of management control, as shown in figure IV.15.

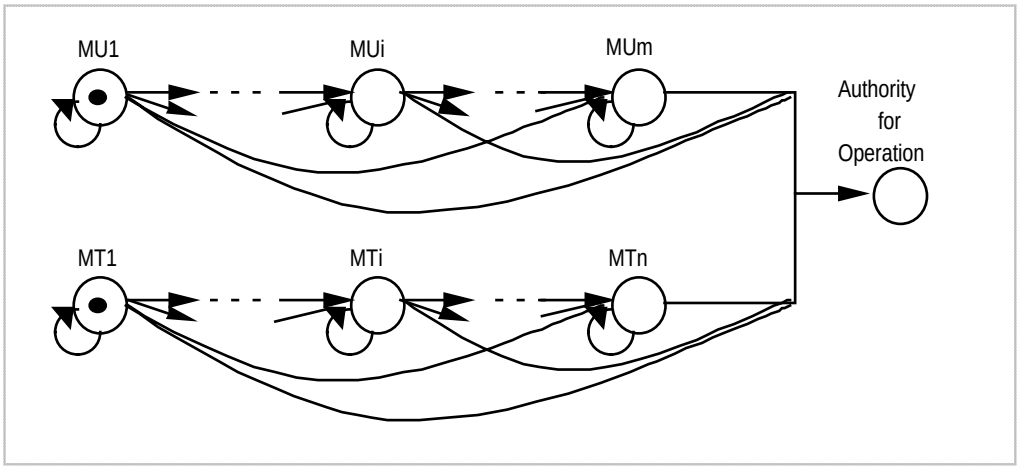

Figure IV.15 Generalised Policy for Flow of Control

MU_1 ... MU_m, MT_1 ... MT_n are totally ordered sets of management authorities, where the MU_i and MT_i represent management authority related to users and (target, operation) pairs, respectively. The MU_i and MT_i sets may overlap. In all the examples above, MU_1 and MT_1 have both been Owner, and MU_m and MT_n have been SA_User and SA_Target.

At least the following requirements must be met by any viable and secure system:

- There must be an initial possessor of authority in any system when it is started up. In the start up system, for every user $User_i$ there must be a user $User_j$ with MU_1 authority over $User_i$, and for every (target, operation) pair ($Target_i$ Op_j) there must be a user $User_k$ with MU_1 authority over ($Target_i$ Op_j).
- *Hierarchical Authority* - The policies must also make it impossible for a user to obtain a higher level of authority than has been delegated to him. There must be no backward loop in the Petri net diagram.
- *Continuous Chain of Authority* - The policies for transmission of authority must make it possible for users to be enabled to perform operations. In the Petri net diagram, there must be no break in the chain of authority between initial Owner and ordinary user.

- *Giving Authority for an Operation* - The effect of the lowest authorities MU_m and MT_n must be to allow the possessor of both to give authority for an operation. In other words if $User_i$ has MU_m authority over $User_j$ and MT_n authority over ($Object_k$, Op_p), he may perform the operation to give $User_j$ authority to perform Op_p on $Object_k$.

Hierarchical Authority - The maximum possible transitions of authority, to ensure that authority always flows downwards, are as follows. A manager user User1, with MU_i authority over a user User3, may give MU_j authority over User3 to another user User2, provided $j \varepsilon i$. Correspondingly, a manager with MT_i authority over a user may give MU_j authority to another user, provided $j \varepsilon i$. In other words, authority may be passed down each management hierarchy, or horizontally, but never up the hierarchy.

Continuous Chain of Authority - One minimum required set of transitions, to ensure that it is possible to proceed from the initial condition to granting of authority for a user operation, is that a manager with MU_i authority should be able to give MU_{i+1} authority, and correspondingly for MT_i. This is represented by the straight lines in the diagram.

Any policy which allows a set of transitions between the minimum and maximum is permissible, although there may be practical disadvantages to allowing horizontal transitions (see above, section IV.7.4), or transitions which skip a level (which imply that a level may be redundant). In our examples we have used horizontal transitions only for Owners, and have not used any transitions which skip a level.

IV.7.6 Segregation of Roles

A further mandatory policy of the system could be for segregation of delegation roles, with the three management roles being mutually exclusive: a User may not simultaneously be more than one of Owner, Manager or Security Administrator for an object. The implementation of this is not necessarily straightforward. If we assume that authority representation is by role domains, it is not sufficient to ensure that only one of `Owner_Scope`, `Manager_Scope` and (the pair of) `SA_Scopes` is not null. In addition it is necessary to ensure that the user is not a member of any other role domain which would violate the segregation.

The discussion in this section makes no assumption about whether this additional policy is in force, but it is stated as a desirable separation of duties by [Yu 1989] and is completely consistent with our approach.

IV.7.7 Further Segregation of Responsibilities for Security Administrators

One commercial organisation of which we are aware takes the segregation of a Security Administrator's responsibilities a stage further than we have discussed so far, by insisting that the user who specifies a new authority or access relationship must be different from the user who actually keys in the command. This is enforced as an administrative rule rather than by the access control system. However, a similar effect could be produced by defining separate groups of operations: Specify_Change would be carried out by one user, but have no effect until an Activate_Change operation was performed by another user. Disjoint authorities for the two groups of operations would enforce the segregation.

IV.8 Summary of Delegation of Authority

This chapter describes the requirements of delegation of authority and our model for it. Access rules alone are inadequate for this purpose, because they do not control the contents of operation requests. It is necessary instead to use the concept of authority relations to model the possible relationships between managers and users. The preconditions for operations required to create these relations are specified to reflect management policy.

To illustrate this the authority relations for the example management structure from chapter II, and the authority which they give, are described. A Petri net diagram illustrates the possible flows of control within this structure. Petri nets are used because they give a good visual representation of flow of control, while providing a precise notation which maps into a Z specification.

The model is extended from single objects to domains of objects by the introduction of role domains, in which the authority of members of the domain is represented by the sets of objects defined in the attributes of the role domain. For example, the Owner_Scope attribute defined the objects for which members of the role domain are Owner. Two separate attributes are defined for Security Administrators, corresponding to the users for whom they can make access rules and the target objects to which they may be given access.

Separation of responsibilities for Security Administrators is achieved by ensuring that they are not members of the SA_User_Scope domain which they control. Two independent managers can cooperate in permitting access, if one sets up the User Scope of a Security Administrator and the other sets up his Target Scope. The clear distinction between ownership of objects and the power to create them is demonstrated by the fact that one depends upon role domain membership and the other upon an access rule.

An approach to the representation of users by the use of role domains is introduced. Role domains are a good vehicle for ensuring the persistence of users within a system, while allowing user processes which move in and out of user role domains to acquire and relinquish the user's authority. They also provide a natural way for mandatory policy to determine a user's authority over his personal domain.

There are a number of alternate management structures possible within our model, using different policies for the authority required to create authority relations. They are shown using modified Petri net diagrams. They can be generalised to any number of levels of management. Outside our model they are further possibilities requiring additional segregation of roles and responsibilities.

V EXAMPLES

V.1 A Commercial Organisation

We will use as an example one which combines aspects of the examples used by [Estrin 1985] and by us in [Moffett 1988] to illustrate the main points in our exposition. A commercial organisation, ABC Ltd, with a Research Department which is engaged in a joint venture with an external organisation, DEF Ltd. There is a need to organise access control for ABC itself, and also to allow limited access for members of DEF staff to Research Department files. DEF staff gain access from terminals in their own research laboratory via a communications network. After an audit of the security arrangements at both sites, ABC have taken a policy decision to allow DEF limited security administration rights to control the users who can access ABC. ABC's policy is to use the standard management structure as described in chapter IV.

ABC's system consists of a number of computers connected by an company network. The example makes no assumption about what resources are located on what computer, but requires a global identification scheme so that all objects may be referred to uniquely.

V.1.1 Domains in ABC

Basic Domain Structure

The Root Domain ROOT_DOM contains, among others, USERS_DOM for users and RESOURCES_DOM for resources. RESOURCES_DOM contains, among others, a top-level FILES_DOM, which is the equivalent of a root directory. In addition there is a role domain, OWNER_DOM, with Owner, Manager and Security administrator scopes for the whole of ROOT_DOM containing a User object THE_OWNER. See figure V.1.

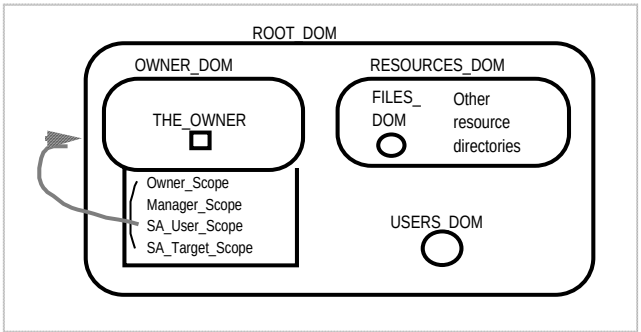

Figure V.1 Root Domain

Organisation

Figures V.2 & V.3 show the organisations as viewed from ABC.

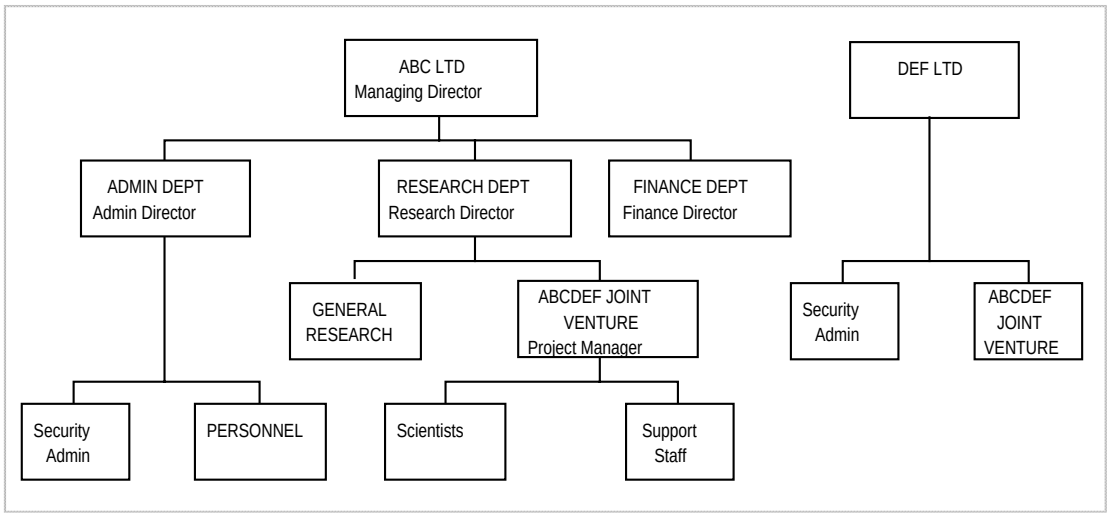

Figure V.2 Conventional Organisation Tree

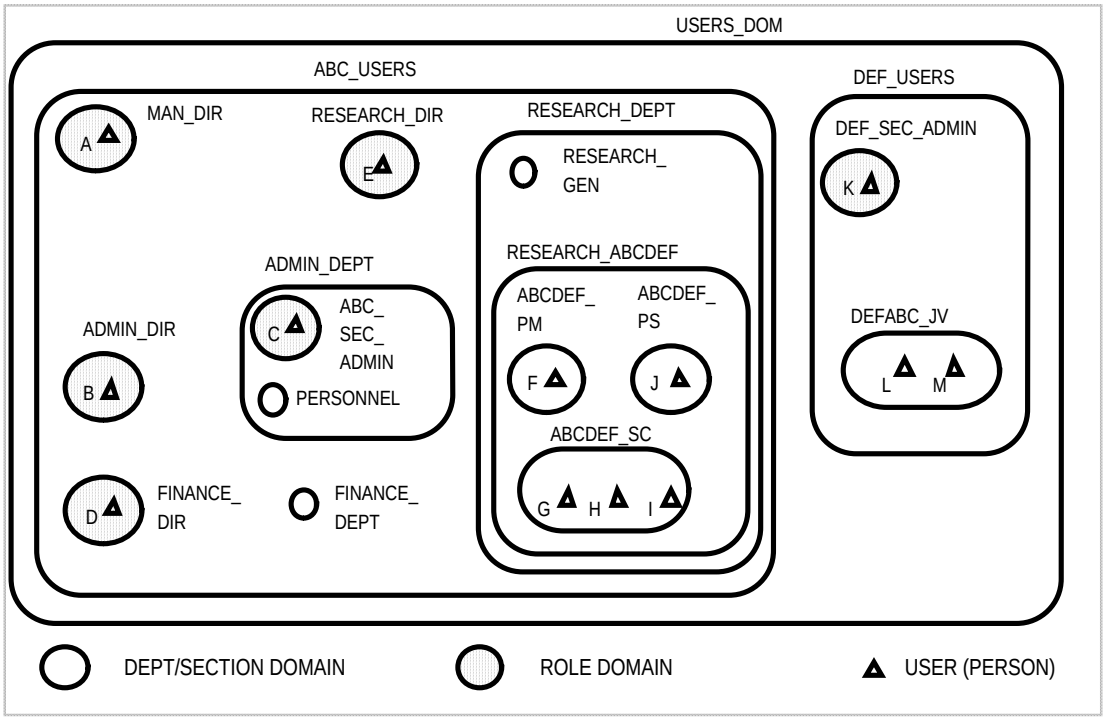

Figure V.3 Domain Representation of Organisation

A conventional organisation chart is shown for comparison, and it will be seen that there is a close correspondence between the conventional and domain representations. However, there is information, about authority, in the organisation chart which is not present in the domain

diagram. We can tell directly from the organisation chart 'who manages[4] whom', e.g. that the Admin Director manages the Security Administrator. We cannot tell that from the domain diagram; it would be perfectly consistent with it for the Security Administrator to manage the Admin Director. Simple domain diagrams tell us only about their membership and not about the authority relations between domains. The extra information is added in the role domain diagrams, figures V.6 and V.7, below.

File Structure

We show a part of the file structure and its corresponding domain representation in figures V.4 & V.5. The Personnel and Suppliers Directories contain data which is registered under the Data Protection Act. Unlike the Organisation structure, no information is lost in the translation between the tree structure and the domain diagram.

Cross-Department Domains

A domain which spans the domains of two departments, such as DPA_DOM in figure V.5, may be required for specific purposes. In this case we assume that members of PERSONNEL need to be able to Read the contents of all files containing personal data, so that they can monitor compliance with the Data Protection Act. The domain cannot be set up by one user but requires cooperative activity. ABC_SEC_ADMIN will have to create access rules which allow a suitable member of ADMIN_DEPT to create an empty DPA_DOM domain within ADMIN_DIR and include PERSONNEL_FILES into it, and a suitable member of FINANCE_DEPT to include SUPPLIERS_FILES in it.

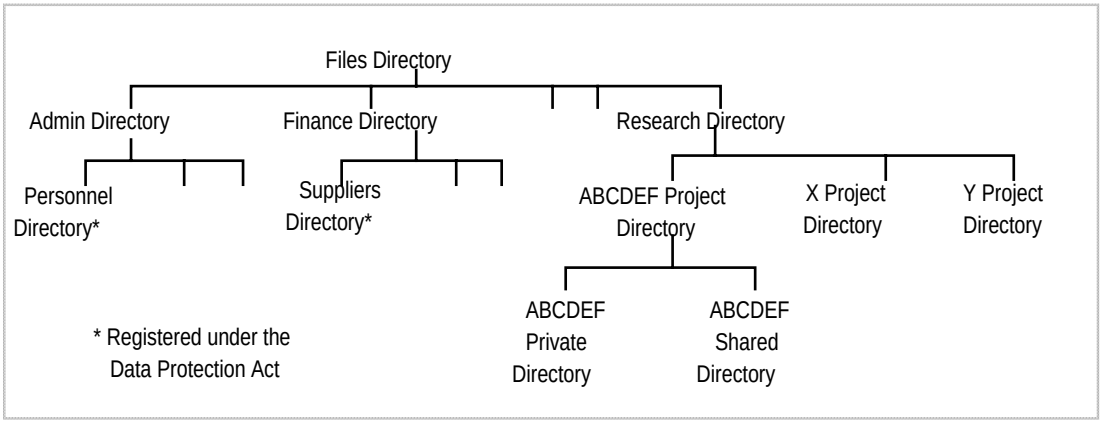

Figure V.4 ABC File Structure

[4] Using 'manage' informally

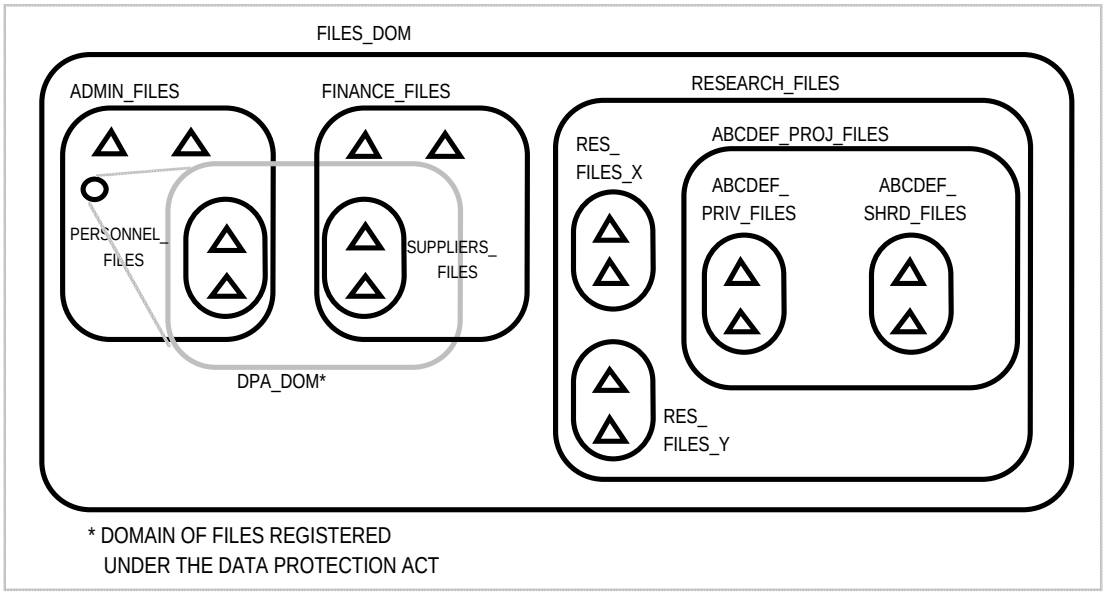

Figure V.5 Domain Representation of ABC File Structure

V.1.2 Delegation of Authority in ABC

The policy of ABC recognises three kinds of authority: Owner, Manager and Security Administrator.

Owner

The OWNER_DOM is Owner of all the users and resources known to the ABC system. This is the starting point, and was shown in figure V.1.

Manager

The occupant of OWNER_DOM makes the following Manager Domains, shown in figure V.6. MAN_DIR is Manager of RESOURCES_DOM (all resources such as files) and USERS_DOM (all Users). Each Director is Manager of his departmental domains of Users and files directory, and in addition ADMIN_DIR is Manager of all external domains such as DEF.

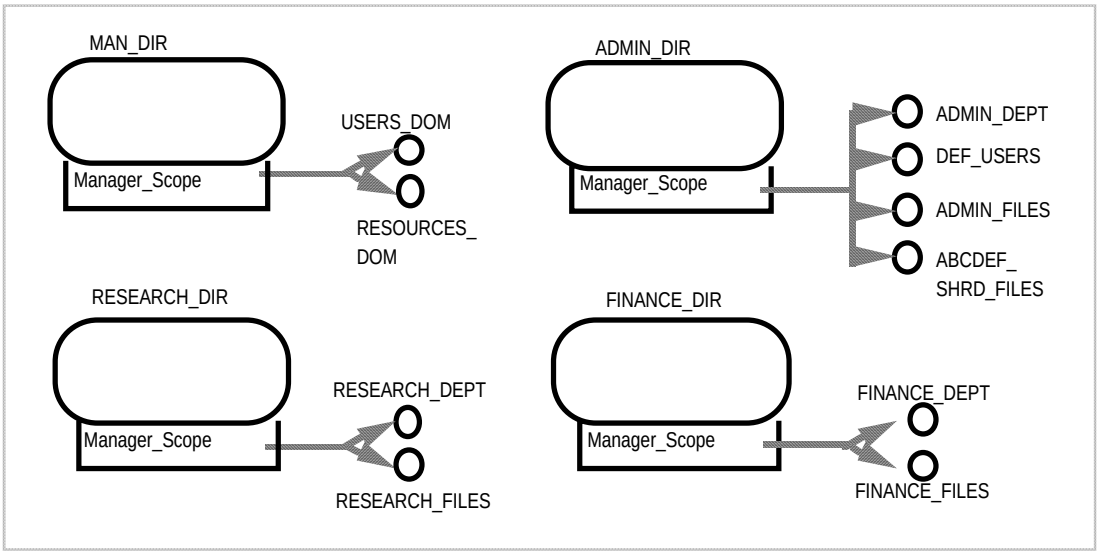

Figure V.6 ABC Manager Relations

Security Administrator

The occupant of MAN_DIR creates a role domain to make ABC_SEC_ADMIN a Security Administrator, with a user scope of the whole of USERS_DOM (except for ABC_SEC_ADMIN itself), including DEF, and a target scope of the whole of FILES_DOM. So members of ABC_SEC_ADMIN can make access rules for all Users (except themselves) to perform operations on any file in FILES_DOM. For simplicity, there is no way for a Security Administrator to give access for members of ABC_SEC_ADMIN to anything; this would have to be achieved by MAN_DIR creating another Security Administrator as described in section IV.4.1.

The occupant of ADMIN_DIR creates a role domain to make DEF_SEC_ADMIN a Security Administrator, with a user scope of DEF_USERS (except itself). The occupant of RESEARCH_DIR then alters the role domain to give it a target scope of ABCDEF_PROJ. So members of DEF_SEC_ADMIN can make access rules for members of DEF_USERS (except themselves) to perform operations on any file in ABCDEF_PROJ directory. Members of ABC_SEC_ADMIN can make access rules for members of DEF_SEC_ADMIN.

Figure V.7 illustrates the security administrators' role domains. It shows clear separation of responsibilities for them. For simplicity we have omitted to show the means by which another user, e.g. the ADMIN_DIR, can make access rules for ABC_SEC_ADMIN.

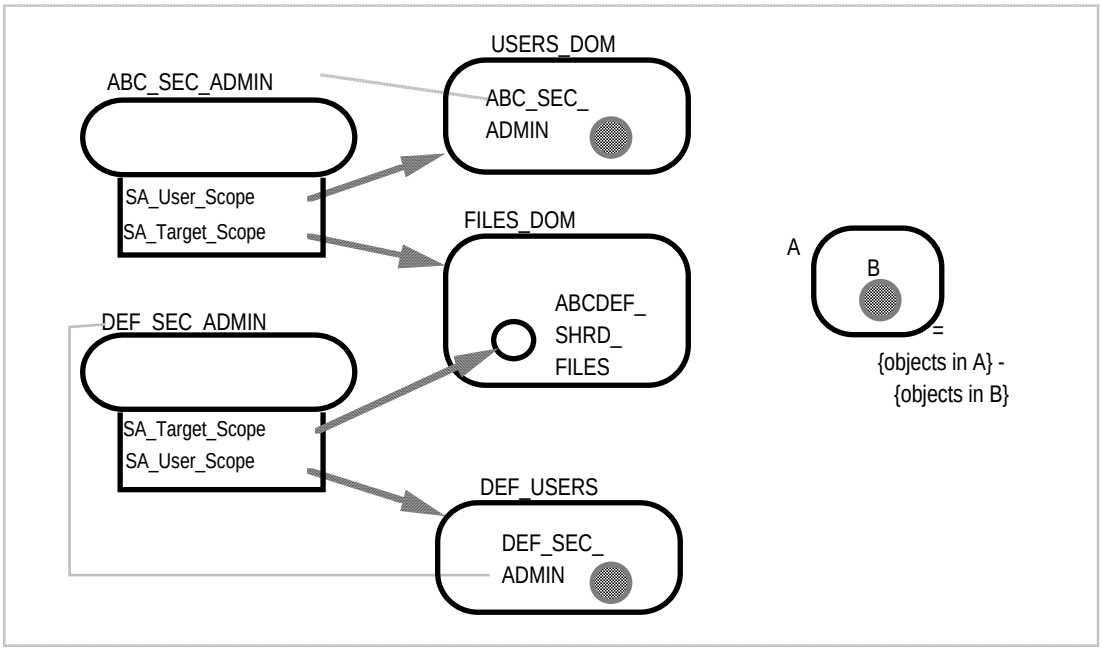

Figure V.7 ABC Security Administration Relations

Management of User Objects in ABC

One of the tasks of a Security Administrator in ABC is to maintain the User objects in the system, by creating, altering and destroying them when required. MAN_DIR has created an access rule allowing members of ABC_SEC_ADMIN to operate on all users in USERS_DOM, and ADMIN_DIR has created an access rule allowing members of DEF_SEC_ADMIN to operate on all users in DEF_USERS. Figure V.8 illustrates the access rules which enable security administrators to manage user objects.

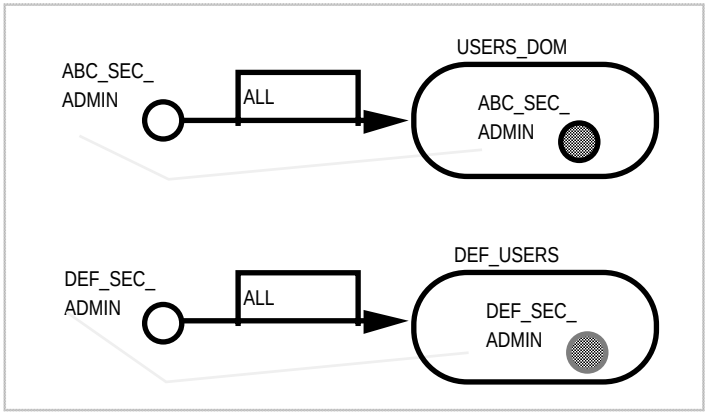

Figure V.8 ABC User Management Rules

Access Control Administration within ABC

The members of ABC_SEC_ADMIN have quite wide powers to create access rules for members of the company to perform operations on company objects, while having no power, except for user administration, to perform operations themselves. So one of the major aims of this model has been achieved. We have chosen to illustrate a single security administrator for the whole company, but clearly a separate administrator for each department could have been created. If we assume that the user and target scope of each Security Administrator is the users and files managed by that department, then there would be a problem of allowing a user in one department to access a file belonging to another. This could be achieved by cooperation between two Security Administrators; one creates an access rule with the required User_Scope but null target scope and the other alters it to introduce the required target scope. But it may be more practical generally to allow each of them to have wider scope.

V.2 Security Strategies for Customer Control of Network Services

We use this example to show that our model is applicable to other fields than simple computer systems. We concentrate on those features which have not already been illustrated by the previous example. Yu's paper [Yu 1989] on Security Strategies for Customer Control of Network Services describes an environment in which a network service vendor provides a set of services to customers. A database which is used for control of the services consists of service records which contain three classes of data:

- Administration and maintenance data to which only vendor employees have access
- Data, such as telephone directories, which is maintained by the vendor but to which customers have Read access
- Customer private data to which vendor staff must not have access.

Since this application has a specific requirement for separation between the service record data which is accessible to vendor staff and that which is accessible to customers, it is an good test of whether our model has enough expressive power.

Resources to be controlled

Yu's main example relates to the way in which Groups and Individual customers may be given authority to `Read` a 'White Pages' Telephone Directory. The resources to be controlled in this example are:

Object Type	**Exported Operations**
'White Pages' Telephone Directory	Read Update
other vendor data	maintenance operations
Customer Private Data	Create Update Read Destroy
Vendor Staff	vendor staff operations
Customer	Create Read Attributes Update Attributes
Group (=domain of customers)	

Table V.1 Resources to be Controlled

Management Structure

Yu contends that it is inappropriate to adopt traditional ownership as the source of authority over use of service objects. His idea of traditional ownership is that ownership stems from creation of an object. This is not our concept, and we use Owner as defined above.

We use the standard management structure as defined in section IV.1 with the following modifications. The policy is extended to allow restriction of the operations which a security administrator can authorise (section IV.7.2). Managers as well as Security Administrators have separate, possibly multiple, pairs of user and target scopes (section IV.7.4). The 'mutual exclusion' rule, preventing any one User from having more than one management role, is in force (section IV.7.6). We assume that there is a root object which is Owner of the whole system and has created the basic objects, access rules and role domains.

Basic Objects

The basic structure is one system domain, All_Dom, and the following domains within that (see figure V.9):

- Vendor_Dom, containing Vendor_Staff for vendor staff objects and Vendor_Data for database objects created by the vendor

- Cust_Dom, containing Cust_Staff for customer staff objects and Cust_Data for database objects created by the customer
- AR_Dom, for access rule and role domains (not illustrated).

The following vendor staff are in Vendor_Staff. A1, the System Manager , is a member of Sys_Dom. A2, the Security Administrator, is a member of SA_Dom, and A3, the Maintenance Engineer, is a member of Maint_Dom. The 'White Pages' Telephone Directory, TD, is a member of Vendor_Data.

There are two domains in Cust_Staff: Cust_Inds for individual customers and Groups for customer groups.

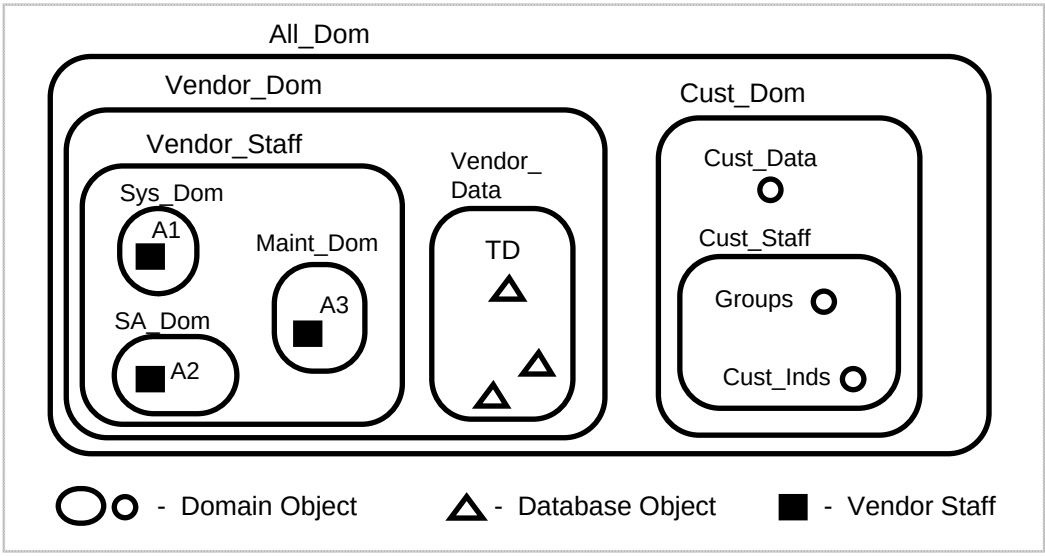

Figure V.9 Basic Domains & Objects

The following access rules and role domains define the authority of vendor staff. Sys_Dom, the domain which contains the System Manager, controls the access of Vendor_Staff to Vendor_Data. It has delegated to SA_Dom authority to create access rules for the access of Maint_Dom to Vendor_Data. The root object has delegated to SA_Dom authority to give Read access for individual customers (Cust_Inds) to Vendor_Data, which includes the Telephone Directory TD. A3, the Maintenance Engineer, has authority to perform maintenance operations on TD. See figures V.10 & V.11.

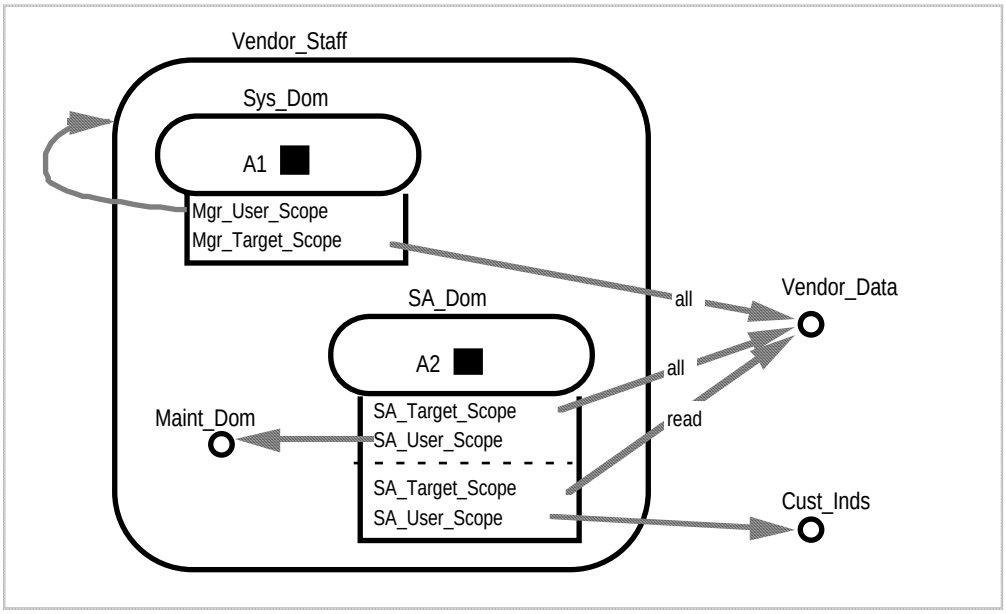

Figure V.10 Role Domains for Vendor Staff Authority

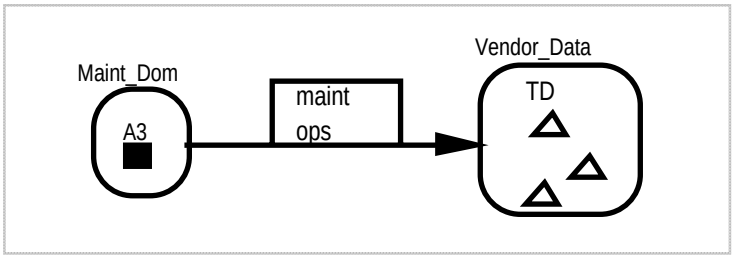

Figure V.11 Access Rule for Vendor Maintenance Staff

Customers and Groups

There are two kinds of customers: Individual and Group service subscribers. Members of SA_Dom, such as A2, act as Security Administrator for the Individual subscribers, as shown above. Groups have their own Security Administrators, who are able to grant services to individual customers in his group. The following access rules and role domains define the authority of customer staff in Group G1. GA1, in the G1_SA role domain, acts as Security Administrator for the G1_Membs domain, being able to give Read access to TD and any kind of access to G1_Data. He has so far only given Read access to TD for all of G1_Membs. See figure V.12.

Note that we have not illustrated the mechanism by which Group Security Administrators are set up. This can be done by making Sys_Dom the Manager of Cust_Dom; the 'mutual exclusion' rule in force on management roles ensures that he cannot give himself further powers in that area. Alternatively this can be done by the system automatically on introduction of a new Group, a Manager process creating the necessary security administrator's domain.

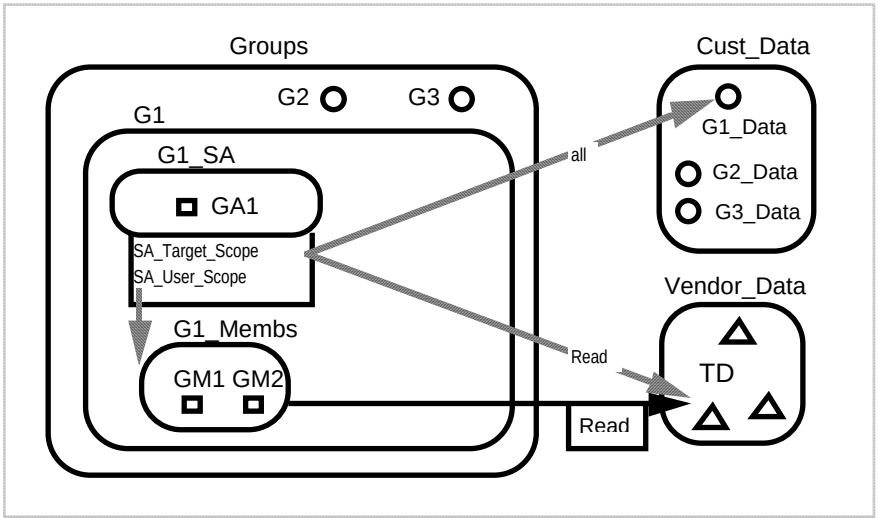

Figure V.12 Access Rule & Role Domain for Customer Staff

Summary of Authority in Network Services Example

Authority in this example can be summarised as follows:

- There is a root object which is Owner of the whole system.
- The System Manager A1 is Manager of Vendor_Staff and Vendor_Data.
- The Security Administrator A2 acts in two roles. He can make access rules: for maintenance staff (members of Maint_Dom) to perform any operation on objects in Vendor_Data; and for individual customers (members of Cust_Inds)to Read objects in Vendor_Data.
- Each customer group, e.g. G1, has its own Security Administrator who can make access rules for users in the group; to perform any operation on the group's private data, and to Read the White Pages directory TD
- Yu does not specify the policy to control the setting up of customer group Security Administrators. We do not model it here, as it would add nothing to the previous example. Possible policies, which could be modelled straightforwardly, are that the System Manager carries this out or that another manager, e.g. a Sales Manager does so.

VI IMPLEMENTATION ISSUES

This thesis does not report on a full implementation of a system using our concepts, as no implementation of domains yet exists. Preliminary work has been carried out [Twidle 1988] and at the time of writing an implementation of domains on Conic [Magee 1989] is under development. It has therefore been necessary to take other approaches to validation of the concepts.

One approach has been to write a system in Prolog which enables us to implement the concepts in a prototype system. This has had two aims: to give confidence in the Z specification; and to provide tools to enable the validation and monitoring of structures of domains, objects, access rules and role domains.

This system has been written and demonstrated. It is introduced and discussed in this chapter and is described in detail in Appendix B.

In addition, implementation issues have been considered and are discussed in this chapter. Three areas have been considered:

- General issues related to domains. Two in particular have arisen: the evaluation of domain membership in the light of the possibility of cyclic membership, and the evaluation of an object's ancestry.
- The practical representation of access rules. In addition to the well-known possibilities of Capabilities and Access Control Lists (ACLs), there is a third possibility, of an authorisation server. A further issue is the possibility of holding the representation of access rules either in terms of domain expressions or else 'flattened down' into their constituent objects. The implications of possible representations are discussed.
- The practical representation of management authority. Although we chose to represent it in terms of role domains in chapter IV, there are other possibilities. In particular, we consider how management authority can be represented in a system which does not use the concept of domains.

VI.1 Prolog Animation

A Prolog animation, described in detail in appendix B, was written for several reasons:

- As an aid to thinking. Each version of the ideas in this thesis has been accompanied by a Prolog animation, and the ability to see if the ideas work in an animated model was very important.

- To validate the Z specification. It has already been explained that Prolog by itself was not found to be a satisfactory specification language because of its lack of support for type-checking or modularity. However, there is no means of automatically animating Z, and it was essential to be able to ensure that syntactically correct Z made semantic sense. Translation of Z into Prolog was known to be straightforward [Stepney 1987], and has even been attempted automatically [Dick 1989] though without succeeding in obtaining reasonable performance.
- To provide a simple query and validation tool for domains and access rules, which could either be used directly as a prototype or as a validated specification for implementation in another language.

The program has two parts: a translation of the Z specification of objects and operations, and a representation, as a Prolog database, of the objects in the example which is described in English language in section V.1.

User Interface to the Prolog Program

We have relied upon the standard user interface to Prolog, without building any special purpose menus. All operations and queries are actioned as queries. Deterministic operations are carried out by entering the appropriate query goal in which all the variables are replaced by atoms. Queries are carried out by leaving some variables uninstantiated. This has been satisfactory for the purpose of building a prototype system, but any operational system would require purpose-built menus and output screens.

VI.1.1 Objects and Operations

The part which defines objects and operations corresponds quite closely to the Z specification, and where possible the comments in the program are expressed as cross-references to the Z.

Objects themselves are defined as Prolog facts, and all objects have a unique Identity (integer), a Name (treated as a mnemonic with no semantic significance), an alphabetic Type and a Domain Set (set of Identities enumerating the domains of which the object is an immediate member). Domains, access rules and role domains are then defined as objects of named type satisfying the appropriate predicates about their attributes. In addition two other object types, Users and Files are used, from which the database is constructed.

Operations are defined for creation, modification and destruction of generic objects, with further predicates added to define them for particular types.

The operation of a reference monitor is simulated by inserting into every operation a clause which requires the triple (user, target object, operation name) to be a permitted operation, where permission is evaluated by searching for an access rule object which applies to the triple. Similarly, evaluation of the appropriate authority for operations on access rule and role domain objects is ensured by inserting clauses requiring the existence of a role domain which gives the user that authority.

The evaluation of indirect domain membership is done whenever an object is created, modified or destroyed, and then stored as a field in the object. The set of all domains of which an object is an indirect member is calculated and stored at the same time. These two fields were added to objects in order to improve performance.

The evaluation of domain expressions is carried out when required. The Object Set of domains and the Domain Set of objects, to determine a domain's direct membership and the domains of which an object is a member, are determined by direct access to the object's attributes.

VI.1.2 Database Queries

We created a database which is an accurate representation of the first example in chapter V. It was constructed by starting from the minimal system shown in figure V.1, and building up the full database by means of operations to create and modify objects. These operations were subject to the full policy of authority and access control which we have been advocating, and so we have demonstrated that quite a complicated system can be created within these constraints.

The set of objects in the database are exactly those which can be seen in the diagrams in section V.1, plus additional access rule objects which were omitted from the diagrams for simplicity.

We have provided two kinds of query: domain structure validation queries; and queries about access and authority.

Domain Structure Validation Queries

In any database of domains and objects, there is the possibility of the structure becoming corrupt. We have identified four separate kinds of corruption. Any object may be a disowned_child or an orphan. A disowned_child has the identity of a domain in its Domain Set, but the domain does not have the object's identity in its Object Set. An orphan has no identity in its domain set, or the identity does not correspond to a domain object.

A domain may be a bereaved_parent or a lost_parent. A bereaved_parent has an identity on its Object Set which does not correspond to an existing object. This situation may occur legitimately within this model: if an object is a member of domains DomA and DomB, and an operation on DomA destroys it, then DomB becomes a bereaved_parent. A lost_parent has an identity in its Object Set corresponding to an existing object, but the object does not have the domain's identity in its Domain Set.

These four queries, run with uninstantiated variables, will search the database for these forms of structural corruption.

Queries about Access and Authority

The queries about access and authority are designed to give information about: the target objects which a user can access, and vice versa; and the management authority possessed by users.

access_rule_effect is a query which is run to find out the set of users, the set of operations and the set of target objects which are authorised by an access rule. user_can_access returns, for a particular user, all the sets of access rules, which authorise access for the user, together with the objects and operations allowed. object_can_be_accessed returns the corresponding information for a particular object.

Management authority can be queried by querying the contents of each role domain in the database. Alternatively, the user_has_auth query lists the role domains of which he is a member and the authority which each gives him.

Interface to the Prolog Queries from another Database

The queries need, of course, a Prolog database. Queries on a domain database in another format could be carried out by translating that database into the correct format. Since Prolog will accept ASCII source, any database which can translate its objects into the ASCII format in appendix B could in principle be queried by this means.

VI.1.3 Comments on our Use of Prolog

Writing the Prolog program, by hand-coding it from the Z specification, proved straightforward, although even on a Macintosh SE-30 or a 386 PC (it works on both) there were problems of performance. This was mainly related to the need to reevaluate domain membership whenever access rules were searched in order to validate an operation. We eventually included a field containing the full ancestry in every object and a field containing the full indirect domain membership in every domain. Every time an object's membership

was changed, the full set of domains of which it was a member was re-evaluated, and also every time a domain object was updated its full membership was re-evaluated. This had a further adverse effect on the speed of updating the database, because MacPROLOG updates the screen every time an object is altered. The resulting version is very slow in updating (up to a minute or two for a single database update operation) but reasonably efficient for querying and validating (10-15 seconds for most queries on our sample database). Our experience confirms the general view that while Prolog is useful as a tool for structuring and helping to reason about problems, it cannot be considered for implementations where there are more than a small number of objects in the system.

However, the trade-off we used may usefully be considered for a full implementation. It is probably impractical to maintain the full ancestry with every object and full membership with every domain, and does not have immediately identifiable benefits except for optimising queries. However, the full ancestry of users would be of immediate benefit in the evaluation of operation requests by means of access rules, as applicable access rules could immediately be identified. This is discussed later (section VI.3.4) when considering the internal representation of access rules.

The database of objects which we created, although orders of magnitude smaller in numbers than realistic systems, does have realistic complexity. There are only 82 objects in our sample system, but the maximum depth of user domains is 6, and of resource domains it is 5. Although this is less than would be seen in a realistic system - our guesstimate based on our knowledge of commercial organisations is a likely maximum of 10 - it is of the same order of magnitude. It was sufficient to be able to demonstrate all the major points in this thesis, such as segregation of responsibilities for security administrators and cooperation between independent managers.

Several minor errors in the Z specification were found, and corrected, as a result of this exercise.

The Prolog program as it stands could be used as a query and validation tool for a database of domains and access rules. The objects in the system can be input as ASCII character strings, and the performance for queries is such that it would be possible to run them on a database of several hundred objects.

VI.2 Domains

This section deals with the issues of the evaluation of domain membership in the light of the possibility of cyclic membership, and the evaluation of an object's ancestry.

VI.2.1 Domain Cycles

There are (at least) three sorts of domain structure, illustrated in figure VI.1:

a) Tree: Each object, except the root, is a member of exactly one domain.

b) Rooted acyclic directed graph: Some objects are members of more than one domain but there are no cycles.

c) Cyclic directed graph: Some domain objects are direct or indirect members of themselves.

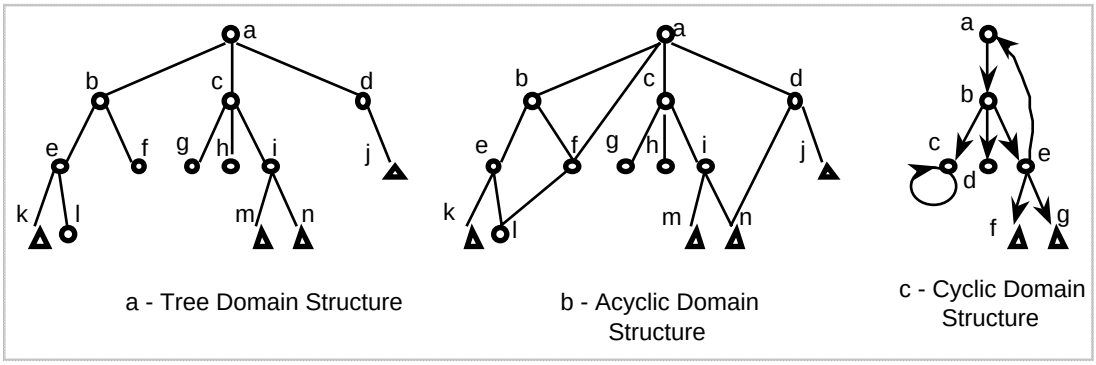

Figure VI.1 Sorts of Domain Structure

Intuitively, case a will never occur, as there will be overlapping domains in any realistic structure. Case b, in which some objects, both domain and nondomain, are members of overlapping domains, is the normal case. Case c, with cycles, may be permitted by some applications and forbidden by others.

Provided evaluation of the total membership of a domain can be done in all cases, then there is no reason in principle to forbid domain cycles. Individual applications can prevent the introduction of cycles by evaluating the potential new total membership when there is a request to include a domain object, and forbidding it if this would result in the introduction of a cycle.

Algorithm for Enumerating Members

Basic assumption: The objects in the system can be enumerated and their number is finite.

1. Define three sets of object identifiers:

- The Membership Set - the set of object identifiers known to be in the membership. This is initially empty.
- The set of Domains Tried Already (DTA)- the domains which have already been searched for members. This is initially empty.

- The set of Domains Still to Try (DST)- domain objects with potential new members. This consists initially of the original domain whose membership is to be enumerated.

2. The Evaluation Step.

2a. If DST is empty, exit from the algorithm, returning the Membership Set as the result.

2b. Select a member of DST, remove it from DST and add it to DTA. Enumerate its object set and add its members to the Membership Set . Add all those object set members, which identify domain objects, to DST, unless they are already in DTA. Repeat the evaluation step.

We need to prove two properties of this algorithm:

a) Does the algorithm always terminate?

Yes. The number of times which the evaluation step is invoked is equal to the number of object identities on the DTA set. This is no greater than the total number of domain objects in the system, which is finite.

b) Does the algorithm enumerate the total membership of the target domain?

Case A - Domain Objects.

Suppose that a domain object, Dom_1, is an indirect member of the target domain but is not found by the evaluation. Then by definition, either Dom_1 is a direct member of the target domain or it is a direct member of a domain which is an indirect member of the target domain.

i) If Dom_1 is a direct member of the target domain then its identity is found in the first invocation of the evaluation step. This alternative is therefore impossible.

ii) If Dom_1 is a direct member of a domain, say Dom_2, which is an indirect member of the target domain, then Dom_2 cannot have been found by the evaluation either, otherwise Dom_1 would have been added to DST.

iii) Further, for any n, if Dom_i has Dom_1 as a member at a distance of n and has not been found by the evaluation, then there must be a Dom_j with Dom_i as a member, and therefore with Dom_1 at a distance of n+1 from it, which has not been found by the evaluation. In other words, there is an infinite number of domain objects which are indirect members of the target domain, which is contrary to our basic assumption.

Case B - Nondomain Objects.

If there is a nondomain object which is an indirect member of the target domain but is not found by the algorithm, then by definition, either it is a direct member of the target domain or it is a direct member of a domain which is an indirect member of the target domain.

i) If it is a direct member of the target domain then its identity is found in the first invocation of the evaluation step. This alternative is therefore impossible.

ii) If it is a direct member of a domain, say Dom_i, which is an indirect member of the target domain, then Dom_i cannot have been found by the evaluation either. This has been proved impossible in Case A.

Therefore all direct and indirect members of the target domain are enumerated by the algorithm, provided the basic assumption of a finite population of the system is met. Note that there may be realistic conditions in which the assumption is false. Since we do not necessarily have global knowledge of the state of the system at any one moment, we have to rely upon communication between distributed nodes of the system with finite delay time and the state of the system may change during this time, so the objects which are members of the system at any one moment cannot be enumerated. There is a need to investigate algorithms for membership evaluation under relaxed assumptions.

VI.2.2 Object Ancestry

Evaluation of operation requests using access rules requires an answer to the question: is there an access rule which authorises the request? Since the access rule is normally expressed in relation to a domain which contains the object directly or indirectly (an *ancestor*), not to the object itself, we need to be able to enumerate the ancestors of an object. This is aided by the Domain Set of an object, which is the set of identities of domains which directly contain it. Similar considerations apply to those for membership enumeration.

Note that in each case a search could be terminated prematurely if the access rules do not permit Read access to the object sets of the relevant domains. It is necessary to ensure that the searching object has adequate authority. It is assumed that the reference monitor itself is not an object which is subject to access control.

VI.3 Representation of Access Rules

The task of access rule representation is to present a user interface consistent with our description in chapter III, above, while providing a system with reasonable performance. From a user point of view the underlying representation of the access rules that he sees is unimportant. They may indeed be access rules in a database managed by an authorisation server, or they may be flattened into capabilities or ACLs located close to the user or the object. However, they must either be held in a form which is recognisable to him as access rules, or separate source and object versions must be held, where the source is the access rules as seen by the user and the object is their internal representation.

The main implementation issues, which should be hidden from users, are:

- How to implement the reference monitor in a distributed system.
- How to represent access rules - whether they should be associated with the user objects, target objects or independent of both, and the relationship between the user view and the implementation view.

VI.3.1 Implementation Requirements

Validity

It is essential that the underlying implementation should make access control decisions which are consistent with the user view of domain-based access rules. When an access rule is created, modified or destroyed, the effect of the change must be reflected in access control decisions. It may be permissible in some circumstances for the effect to be delayed (e.g. until the following day, in the case of non-urgent changes), but this must be predictable. In addition when an object is included in or removed from a domain, or a new object is created, then the object's access authorisations (either as user or target) which derive from membership of the domain must change accordingly.

Performance

The overhead introduced for validation of authorised operation requests must be minimal if the system is not to degrade.

The overhead in updating access authorisations must be tolerable for domain membership changes. It is thought that the contents of user domains will change relatively infrequently compared to those of target domains which contain objects such as temporary files.

Changing access rules and obtaining status reports will be relatively infrequent and do not always have to take immediate effect, so higher overheads can be tolerated.

VI.3.2 Reference Monitor Selectivity

We have assumed in earlier chapters that the reference monitor will trap and validate all operation requests in the system. However, there are clearly two classes of operation for which explicit access control may be unnecessary in an implementation.

Firstly there are operations which are too trivial, from a control point of view, to require access control. Even if an implementation treated numbers as objects, we would not wish to carry out access control on arithmetic operations. There are no doubt other less obvious cases in any system which do not require access control.

Secondly, there are operations which have been implicitly validated by previous operations to which they are linked. If a Bind operation on a server restricts further operations by the user to reading a specified object, each subsequent Read operation may not require access control validation, provided there is sufficient confidence in the integrity of the system to prevent a user from reading without binding first.

Decisions about the operations which the reference monitor will select to validate are part of mandatory security policy to be decided during system design.

VI.3.3 Possible Reference Monitor Configurations

We make a basic assumption about the Access Control Enforcement Facility (AEF) (figure II.3). One AEF is assumed to act as the enforcement facility for all the objects in a suitable unit such as a processor node, in other words protection is provided at a system level, not done by the objects themselves. We discuss this in relation to reference monitors in the object area of trust (see figure II.5); similar considerations apply to the user area of trust.

We can distinguish two basic configurations:

- An unvalidated Operation Request is sent to the AEF, which refers the decision directly to the Access Control Decision Facility (ADF), i.e the ADF is associated with the target object. This is exactly the situation of figure III.2.
- The user asks the ADF, which must also be trusted by the object, for a Capability for the operation.

ADF Associated with Target Object

The ADF may be in the same trusted computer or system as the target object so that communication is secure. In that case the implementation may be as defined in [ISO 1989c]. Since communication is secure, it may be assumed that the message has not been interfered with between leaving the ADF and reaching the object. Many mainframe access control systems such as IBM's RACF [IBM 1985] follow this approach.

Possible representations for this configuration are ACLs or an access rules database (see VI.3.4).

ADF Generates Capabilities

The ADF may communicate with the target object via an insecure transmission medium which implies the message may have been interfered with between leaving the ADF and reaching the target object. It is therefore necessary to generate a secure capability, as is done in Kerberos [Steiner 1988], which is checked by the AEF close to the target object. The

Capability is included as part of the Operation Request and the AEF verifies its validity before allowing the operation to be invoked. See figure VI.2. The ADF has to do the substantial work of making the access decision, while the Capability Checker need only check that the capability is valid for the Operation Request which it accompanies.

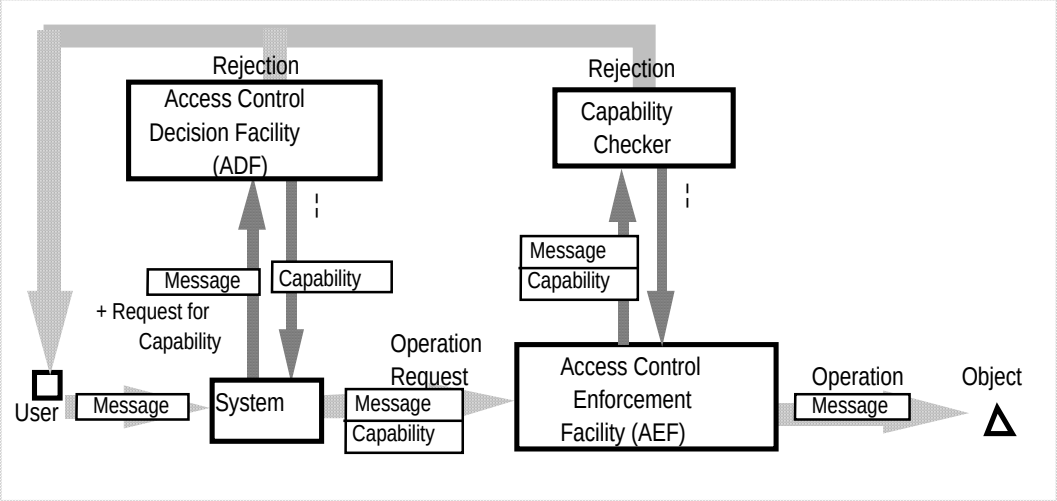

Figure VI.2 ADF Generates Capabilities

Capabilities can be generated, when a session is established between a user and a target domain or even when the user logs into the system. They can be cached at the user and reused for multiple operations, thus reducing the overheads of operation invocation to that of checking capability validity.

An advantage of the capability based approach appears when a server object is requested to carry out an operation on behalf of a user. In this case the Operation Request originated by the User is different from that sent to the target object. For instance the original operation may be 'Read ObjA' sent by UserX to the object ServerY (message 1), but it will result in 'Read' sent to ObjA (message 2) by ServerY.

Using the first approach, the only Operation Request which can be authorised by the reference monitor is the one which is actually sent. Message 2, which is is an operation on ObjA by ServerY, requires authorisation as if it were being requested by UserX, not by ServerY which is actually performing the request. The solution usually adopted is for the server to adopt the identity of the user in order to obtain user-specific authorisation for the operation. It has been pointed out, e.g. by [Vinter 1988], that this breaches the 'least privilege principle' as the server can now perform all the operations authorised for the user, and not only the one which was requested. On the other hand, if capabilities are used, the original authorisation which UserX obtains can be for 'Read' on ObjA and this can be passed to ServerY as a capability which authorises it to perform the requested operation and no other.

Possible representations for this configuration are access information held with the user, or an access rules database (see VI.3.4).

VI.3.4 Internal Representation of Access Rules

Location of Access Rules

There are three possible types of locations for access information:

- In an *access rules database*. An ADF served by a database can act as an Authorisation Server which either responds directly to an AEF or generates capabilities for the User for later validation by the AEF.
- Information held with the target objects and/or their domains (Access Control Lists)
- Information held with the user objects and/or their domains, to enable generation of capabilities.

Flattened Rules

In addition there is the option of *flattening* down the rules so that their information is held at the lowest level of domain, or with individual objects. The more the flattening, the less need to search the domain structure but the greater the task of updating. This distinction applies in two dimensions; whether or not (in the case of ACLs) the access permissions are held at every level of domain or only at the lowest level, they could be expressed at each level in terms either of individual users, lowest level of user domain or of the user domains specified in the access rules.

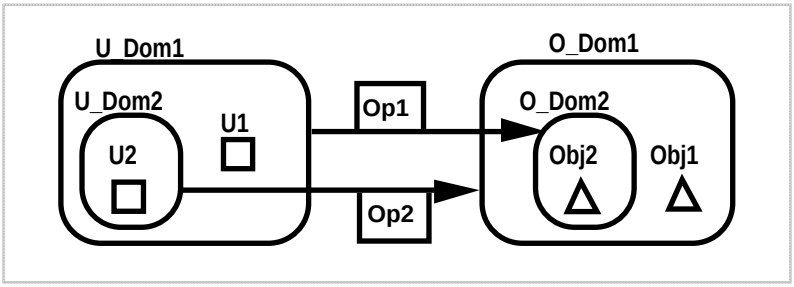

Figure VI.3 Flattening Example

Using the example access rules of figure VI.3 we can see the effects of the possible options:

Flattening	**Type**	**Held at**	**Data**
Unflattened	Database	Database	U_Dom1, O_Dom2, Op1 U_Dom2, O_Dom1, Op2
	User based	U_Dom1 U_Dom2	O_Dom2, Op1 O_Dom1, Op2
	ACL	O_Dom1 O_Dom2	U_Dom2, Op2 U_Dom1, Op1
Flattened User Domain	Database	Database	{U1, U2}, O_Dom2, Op1 U2, O_Dom1, Op2
	User based	U1 U2	O_Dom2, Op1 O_Dom2, Op1 O_Dom1, Op2
	ACL	O_Dom1 O_Dom2	U2, Op2 {U1, U2}, Op1
Flattened Target Domain	Database	Database	U_Dom1, Obj2, Op1 U_Dom2, {Obj1, Obj2}, Op2
	User based	U_Dom1 U_Dom2	Obj2, Op1 {Obj1, Obj2}, Op2
	ACL	Obj1 Obj2	U_Dom2, Op2 U_Dom1, Op1 U_Dom2, Op2
Both Domains Flattened	Database	Database	{U1, U2}, Obj2, Op1 U2, {Obj1, Obj2}, Op2
	User based	U1 U2	Obj2, Op1 {Obj1, Obj2}, Op2
	ACL	Obj1 Obj2	U2 Op2 {U1, U2}, Op1 U2, Op2

Table VI.1 Flattening Options for Access Rules

If the target (/user) domain is flattened, then: whenever an operation is requested there is no need to evaluate the target (/user) domain membership to identify the permissions which apply; but whenever an target (/user) moves into or out of a domain the effect of that movement on permissions has to be calculated, adding or removing permissions as appropriate. Since we assume that target domains are more volatile than user domains, there will be more advantage in flattening user domains than target domains.

If we do not flatten both ways, then a domain server is needed to evaluate domain contents. But if there is a server intervening every time there is an operation request, it is simpler to

store all permissions in a database and use an authorisation server also. So it is not efficient to flatten one way only.

Access Rules Database

An access rules database can be independent of both users and target objects as part of an authorisation service. It would in general be based on a distributed database and distributed authorisation servers. It would hold information about domains and their relationships and so would lead to a duplication of information in the system.

Access Rules Held with Users

Access rules can be held as attributes of user objects. The access rules (and ADFs) would thus be distributed to reflect the distribution of the users. The problem with holding access rules with the users is that they may exist in personal workstations which cannot be trusted, and their security is therefore difficult to enforce. In addition access rules are thought to be more likely to be generated by security administrators associated with target objects rather than with users.

Access Control Lists (ACLs) with Target Domains

An alternative approach is to hold access rules as ACLs which are attributes of the target objects and their containing domains. The security of the access rules can then be commensurate with the security of the objects they are protecting. Target objects are more likely to be based on servers, under the control of system managers rather than individual users, and their domains and access rules can be held on more secure systems. It appears practical for each user object to maintain its full ancestry, which will be relatively static, and pass it to the reference monitor with each operation request. The ACL's access information, in terms of user domains, could then be evaluated against the user's ancestry.

Relationship between the User and Implementation Views

Ideally the user view can be obtained directly from the implementation view, and then there is no need to maintain source and object versions simultaneously. The conditions to be met for this are:

- The implementation holds all relevant information about the structure of the access rule. This implies that (for ACLs) full information about the User Domain side of the access rule is held in each ACL entry.

  It also implies that some optimisations may not be available for use. If a security administrator has specified an unnecessarily complicated domain expression, it is not

permissible to simplify it in the stored version, unless the original can be reconstructed for the user.

- User comments and formatting in access rules may be lost.

On the other hand, if separate source and object versions are maintained, then it is necessary to ensure their continuing consistency, which is an additional task for the system.

VI.4 Representation of Authority Relations

We chose in section IV.3 to represent authority relations by means of role domains. This section briefly discusses other possibilities:

- Linked Authority Access Rules
- Access Rules with Constraints
- Authority Templates.

All of the above assume that the method of access control that is used is access rules. One other possibility should be mentioned, which makes no assumption about the method of access control used, except that it is identity-based:

- Authority as User Profile Attributes.

VI.4.1 Access Rule Based Authority

Other possible representations of authority, besides role domains, are:

- Linked Authority Access Rules - each authority relationship is represented by an access rule with a special kind of 'operation', e.g. Owner_Scope for an ownership relationship. In cases where authority relationships may be linked, e.g. SA_User_Scope and SA_Target_Scope, then the access rules must be linked. For example, policy may require a security administrator to have authority to give access for UsersA_Dom to TargetA_Dom and access for UsersB_Dom to TargetB_Dom, so his user scope and target scope need to be linked.
- Access Rules with Constraints - an access rule which permits the creation of access rule objects has constraints associated with it which represent the authority relationship, e.g. an access rule for a security administrator has two constraint attributes, SA_User_Scope and SA_Target_Scope, which define the scopes of members of the access rule's User Domain for these two kinds of authority. The Target Domain of the access rule represents the domain in which a new access rule object may be created.
- Authority Templates - a separate type of object, a Template, has a User Domain attribute which defines a set of users, and authority attributes which define the users' authority similarly to a role domain.

All of these representations are capable of expressing authority relationships in a similar manner, but note that, in all representations except access rule with constraints, there is also a need for an additional access rule to exist which allows the creation of an access rule object.

Discussion

The first point that strikes one with regard to authority is the number of plausible candidate representations that there are, in contrast to access relationships, where the access rule is clearly the best and simplest representation. It is clear that each of these representations is equivalent, in the sense that each is capable of representing the members of a domain expression being in an authority relationship with the members, or pairs of members, of other domain expressions. We therefore need to consider the criteria on which we should choose between them. The criteria include:

- How well does each conform to our intuitive (user level) view of authority?
- Economy of concepts at a user level
- Economy of concepts at an implementation level
- Ease of evaluating authority relationships
- Robustness to failures.

We have not included performance among these criteria because authority-giving activities can be assumed to take place rarely.

The chosen method for implementation does not appear to be a critical decision and is the subject of further study. However, we chose role domains for this thesis for the following reasons: it corresponds most closely to our intuitive view of authority, and it led to the simplest representation in diagrams. In addition, it appears to minimise additional complications in implementation and does not require the integrity of links to ensure consistency.

VI.4.2 Authority as User Profile Attributes

Since the concepts of delegation of authority make good sense in most systems, and not simply in those where domain-based access rules are used, we should consider how it could be implemented elsewhere.

This has in fact already been done in IBM mainframe security systems such as ACF2 [CA 1988], but not in such a general manner. The concepts which we have put forward can be implemented by means of fields in the User Profile record. Owner Scope and Manager Scope fields, and a pair of User Scope and Target Scope fields, or their equivalent, provide

representation of the Owner, Manager and Security Administrator authority relationships. The decision criteria are similar to those described above for role domains.

We have already remarked on the danger of attaching authority to individual users, not positions, and this is the main disadvantage of such an approach, as there is no concept of domains which users can move into and out of, gaining and losing the authority associated with the domain.

VI.5 Summary of Implementation Issues

This chapter has discussed implementation issues under four headings.

The Prolog animation validated the Z specification on which this thesis was based, using the example given in section V.1. It was limited by performance problems to a small example, although it would be capable of coping with higher volumes if used as a tool for queries.

There are several possibilities for representation of access rules in an implementation. They could be represented as Access Control Lists, capabilities or as a rules database. An additional implementation question is the extent to which the User and Target Domains should be flattened down to their constituent objects. If the access control information is only held at the domain objects which appear in the access rule, there is no task of updating when the domain contents change but interpretation of the access rule is slower.

Issues of Performance and Scaling

The major outstanding issues in implementing a system which uses access rules are mainly related to performance when the system is scaled up to realistic numbers. The use of domains helps with scaling up, because the structuring of objects into groups reduces the need for long serial searches of lists. Domains will typically contain tens or perhaps even hundreds of members, although the depth of the domain structure is not thought likely to be very much greater than that in our test example. Frequent searches of the entire domain structure will be impossibly expensive.

We have isolated the performance issue to one area: the evaluation of operation requests by the reference monitor, because changes to access rules and to managers' authority are occasional events for which a high performance cost can be tolerated. The problem can be eased further by reference monitor selectivity in the operation requests which it validates.

There two main factors which affect the speed of evaluation of operation requests: determination of the domains of which an object is a member - its ancestry - in order to determine whether a given access rule applies to the request; and the need to search through a

number of candidate access rules before rejecting the request. This suggests two separate, possibly combined, approaches to the problem.

Advance evaluation of each object's full ancestry immediately deals with the first factor. The cost of doing this at every update of the object's direct or indirect membership, as we did in our Prolog model, is almost certainly prohibitive, but it would be possible to do this for each user object at the time that the operation request is invoked. This is being considered for a future implementation.

Structuring of the access rules into capabilities or ACLs associated with the user or target objects deals with the problem of having to search through an indefinite number of access rules. As discussed above, there is a further trade-off to be made in deciding to hold the access rule information at the domain level or 'flatten' it by holding it at every object.

Further work is required before deciding on the most efficient representation of access rules, but we believe that the approach which we have identified promises to be practical for full-scale systems.

VII CONCLUSIONS

The thesis has addressed the issues of provision of discretionary access control in very large distributed systems: in particular how to specify access policy for large numbers of users and target objects; and how to delegate authority for this policy to managers and security administrators who may be members of independent organisations.

Our approach to access policy has been by grouping users and target objects into *domains*, and specifying policy by means of *access rules* which define the operations which members of user domains may perform on the objects in target domains. We have then extended the concept of domain to *role domains*. Mandatory policy determines the power which is associated with each management role. Membership of role domains defines the management roles occupied by users and the scope of objects over which the roles are exercised. This determines the hierarchical chain down which authority may be delegated.

The success of this approach has been shown by its ability to satisfy the major requirements of large-scale access control in distributed systems. Separation of authority for security administrators, so that they do not themselves have access to the resources which they administer, has been demonstrated. Also, we have shown how independent managers can cooperate by allowing members of other organisations strictly limited access to their resources.

In this chapter we review the thesis aims and approach given in chapter I to assess how each has been achieved. A critical evaluation of each aspect is provided. Finally, directions for further work are discussed.

VII.1 Critical Review of Objectives & Achievements

Domains

The basis from which we started was the work of Robinson and Sloman on management domains, extensively referenced in this thesis. We have subsequently refined this work. We are satisfied that domains are not only a powerful general concept but also the natural way of expressing our particular application requirements.

We have taken the definition of domains which was arrived at in mid-1989 as the basis for our work, as rebuilding one's tools in midstream is even more hazardous than changing one's horses. They have stood up very well to our use, but one aspect of them requires review; we needed a more powerful means of specifying the set of objects in an access rule or a

manager's scope than simply using the name of a domain . It is essential to be able to specify more complicated expressions, in particular the exclusion of domains using set difference, and also to specify individual objects (rather than domains containing a single object). We have had to remedy both of these deficiencies by devising *domain expressions*, which were not part of the original work on domains.

Domain expressions also highlight another problem, or possibly feature, of domains. Knowledge of a domain's Object Set does not unambiguously tell us what are its members. We can interpret a domain's membership in terms of its direct or its indirect membership and will obtain different results. Since access rules demand precise answers, the other function which a domain expression serves is to fix membership unambiguously, by defining conventions about whether membership is to be interpreted directly or indirectly.

Access Rules

The concept of access rules has proved to be a very successful means for managers and security administrators to specify access control policy. By introducing the notion of specifying the authority of sets of users to access sets of target objects it has dealt with the problems of scale which are not fully addressed by either capabilities or access control lists. Additionally it enables managers to approach specification in terms of the structure of their organisation and resources. In particular, the access privileges of users can be specified in terms of the roles they occupy rather than their individual identities.

There remains, however, a problem in the efficient implementation of access rules. A simple approach which attempted to implement them by means of a search of every access rule in the system for every operation request would be impossibly slow. Several approaches to performance improvements have been investigated theoretically. Access rules map naturally onto capabilities or access control lists for implementation, and we are investigating an ACL approach for this purpose. Other approaches include: reference monitor selectivity in the operation requests which it validates; *flattening* the information in access rules into its constituent domains and objects; and provision of the user's full ancestry with each operation request. A combination of these techniques is likely to lead to satisfactory performance.

We have so far only considered implementation for a single, universally trusted, reference monitor. We pointed out in section II.2.2 that the managers of a user domain may not trust a reference monitor associated with a target domain, and two separate reference monitors may be required. The implications of this require further study.

Delegation of Authority

We defined a example set of managerial roles: Owner, Manager and Security Administrator, and the authority associated with each role. These were then modelled by the use of role domains. A role domain has a defined membership, and scope attributes to delimit the members' authority in each kind of role; for instance the Owner_Scope attribute defines the objects, if any, of which the members are Owners.

The representation of managerial authority relations by role domains provides a good intuitive representation of management authority. We have shown how their use enables separation of authority for security administrators and the cooperation of independent managers. Many different policies can be specified using role domains, by varying the constraints placed upon the creation of each managerial role. We showed how the example set of managerial roles enabled hierarchical delegation of authority from an Owner through to users. We also gave a model of a more general set of roles which still satisfy the need for a hierarchical chain of authority.

We have, however, left unanswered several questions: what types of role domain may there be, what are the constraints on their attributes, and how suitable are they for the implementation of user objects?

Development Approach

Our development approach has been a combination of formal specification, English language, diagrams and Prolog programming.

One of the benefits of the combined approach has been that we have been able to use diagrams for communication of ideas while retaining the precision of formal specification. The representation of our concepts has used diagrams with conventions for the symbols in them, and we have shown how they correspond to Z specifications. This was not only true for the Petri net diagrams which we used for showing the transmission of authority, but also for the domain diagrams used throughout this document.

In relation to methods of representation, we need to comment on our use of a formal specification method, Z, as the basis for our work. There is no doubt that it is difficult to read without prior familiarity, but the use of a language of this kind was essential if we were to avoid the twin traps which otherwise faced us. English language alone is inadequate, because of the many possibilities for allowing ambiguity rather than sorting out one's thoughts precisely. Direct programming, too, has the major disadvantage that whatever is done will be shaped by the style of the language, and will not then transfer readily to other

systems. We started by attempting to write our specification directly in Prolog, and it became apparent that, even with such a high-level language, we were being influenced in the substance of what we did by the style of the language. The very primitive concepts used in Z allow the maximum concentration on the substance of one's work, while leaving the possibility of many styles in the implementation.

Writing in Prolog from a Z specification is very simple, and we believe that it has been a good way of achieving a satisfactory degree of validation of the work by animating the specification. Performance problems however place a severe limitation on the size of system which can be handled with Prolog.

VII.2 Suggestions for Future Work

Suggestions for further work flow immediately from our analysis of what we have done so far. An implementation of domains and access rules is already part of the Domino programme and should produce valuable results. Beyond that, investigation of the implementation of multiple reference monitors needs to be considered.

Several related issues have not been tackled at all by our work. The impact of our work to military-type mandatory access control needs investigation. We have made the simple assumption that all user objects can be assumed to be authenticated, but what are the implications of servers and other agent processes assuming a user's identity? We believe that our work is consistent with the Clark/Wilson integrity model but have not yet explored the relationship.

Another interesting area is the modelling of formal organisations. Most work on the modelling of organisations is either very qualitative, or uses a 'hydraulic' model of the organisation to model it as a system which pumps a fluid such as money or product. There seem promising possibilities for an analytic approach to organisational modelling by broadening the narrow base we have used, delegation of authority, to include other organisational transactions and relationships.

Lastly, a review of our domains work could bear fruit if we explore the compatibility of role domains and access rules. If we talk not about access rules but about access domains, then we gain a more consistent picture. We would then have two variations of one kind of authority, a user's authority to perform normal operations on objects (access domains) and a manager's authority to perform management operations (role domains), rather than two unrelated concepts. Not only is it more self-consistent, but it also provides a basis for meeting the needs expressed in [ISO 1989d] for management domains, and in [ODP 1989] for X-domains, in each case clearly asking for more than the rather austere domain structure

which we have so far created could provide. This has implications for the basic definition of domains. An access domain needs to be unambiguous in its membership, and to be more expressive than the domains we have used so far. The definition of a domain's Object Set would need to be a complete domain expression, yielding a set of objects, rather than a simple list of unevaluated domains and objects. How practical it would be to make this alteration remains to be seen.

VII.3 Final Remarks

We have produced and validated a model which enables the realistic management of discretionary access control in large distributed systems: realistic because it comes to terms with the fact that existing models, using capabilities or access control lists, leave the security administrator with an impossibly large task; realistic, too, because it models the need in real organisations for authority to be passed on in controlled way.

We can view our work from two points of view: from an academic point of view it has injected the needs and the techniques of a commercial world into an academic framework; and from a commercial point of view it may give a reasonable prospect that the next generation of systems will meet commercial needs for access control. We hope that this bridge between two worlds will be useful.

REFERENCES

[Anderson 1972] Anderson J.P., Computer Security Technology Planning Study, ESD-TR-73-51, vol 1 AD-758 206, ESD/AFSC Hanscom, AFB Bedford, Mass, Oct 1972.

[ANSI 1986] Financial Institution Message Authentication, (Wholesale), ANSI Draft Standard X9.9, American Bankers Association, Standards Dept, 1120 Connecticut Ave NW, Washington DC 20036, 1986.

[CA 1988] CA/ACF2 rel 5.1 General Information Manual, ABG0002-03, Computer Associates, Rosemont, Illinois, USA, Feb 1988.

[Clark 1987] Clark D.C. & Wilson D. R., A Comparison of Commercial and Military Computer, Security Policies, IEEE Security and Privacy Symposium 1987, pp 184-194.

[Clocksin 1981] Clocksin W.F. & Mellish C.S., Programming in Prolog, Springer-Verlag, 1981.

[Dick 1989] Dick A.J.J., Computer Aided Transformation of Prolog Specifications, Research Report 10-1702-01, 10 May 1989, Racal Research Ltd, Reading, Berks RG2 0SB.

[DoD 1985] Department of Defense (USA), Department of Defense Trusted Computer System, Evaluation Criteria, DOD 5200.78 - STD (Dec 1985).

[DoD 1987] Department of Defense (USA), Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria, Technical Guidelines Division, National Computer Security Center (USA), NCSC-TG-005 version 1, (July 1987).

[Estrin 1985] Estrin D., Non-Discretionary Controls for Inter-Organisation, Networks, Proc 1985 Symposium on Security & Privacy, pp 56-61, IEEE Computer Society,

[Estrin 1987] Estrin D., Controls for Interorganization Networks, IEEE Transactions on Software Engineering, vol SE-13 no 2, pp 249-261 (Feb 1987).

[Gomberg 1987] Gomberg D.A., A Model of Inter-Administration Network User, Authentication & Access Control, Mitre Corporation, Washington C3I Ops, 7525 Colshire, Drive, McLean VA 22102, MTR-87W00003, Dec 1987.

[Hayes 1987] Hayes I. (ed), Specification Case Studies, Prentice Hall 1987.

[IBM 1985] RACF General Information Manual, GC28-0722-9, IBM Corporation (1985).

[ISO 1988] ISO, Open Systems Interconnection: Security Architecture, ISO 7498/2 (1988).

[ISO 1989a] ISO, Security Framework I: Overview, ISO/IEC JTC1/SC21 N4210 (Draft, Nov 1989).

[ISO 1989b] ISO, Security Framework II: Authentication Framework, ISO/IEC JTC1/SC21 N4207 (Draft, Nov 1989).

[ISO 1989c] ISO, Security Framework III: Access Control Framework, ISO/IEC JTC1/SC21 N4206 (Draft, Nov 1989).

[ISO 1989d] ISO, OSI Systems Management Overview, DP10040, ISO/JTC1/SC21 N4066, Dec.89.

[Klerer 1988] Klerer S.M., The OSI Management Architecture: an Overview, IEEE Network, vol 2 no 2, pp 20-29 (March 1988)

[Lampson 1974] Lampson B.W., Protection, ACM Operating System Review, vol 8 no 1, pp 18-24 (Jan 1974).

[Landwehr 1981] Landwehr C.E., Formal Models for Computer Security, ACM Computing Surveys, vol 13 no 3, pp 279-339 (Sept 1981).

[Linden 1976] Linden T., Operating System Structures to Support Security and Reliable Software, ACM Computing Surveys, vol. 8 no. 4, Dec. 1976, pp. 409-445.

[LPA 1987] MacPROLOG version 2.7 Reference Manual, Logic Programming Associates, London (1987).

[LPA 1988] LPA Professional Prolog version 2.1 Reference Manual, Logic Programming Associates, London (1988).

[McLean 1990] McLean J., The Specification & Modeling of Computer Security, IEEE Computer, vol 23 no 1 (Jan 1990), pp 9-16.

[Magee 1989] Magee J., Kramer J. & Sloman M., Constructing Distributed Systems in Conic, IEEE Transactions on Software Engineering, vol 15 no 6, June 1989, pp 663-675.

[Moffett 1988]Moffett J.D. & Sloman M.S., The Source of Authority for Commercial Access Control, IEEE Computer, vol 21 no 2, pp 59-69 (Feb 1988).

[Mullender 1990] Mullender S.J. et al, Amoeba: A Distributed Operating System for the 1990s, IEEE Computer, vol 23 no 5, May 1990, pp 44-53.

[Needham 1978] Needham R.M. & Schroeder M., Using Encryption for Authentication in Large networks, of Computers, Comm. ACM, 21, 12 (Dec 1978) pp 993-998,

[Needham 1987] Needham R.M. & Schroeder M., Authentication Revisited, ACM SIGOPS, Vol 21 no 1 p 7 (Jan 1987).

[Neely 1985] Neely R.B. & Freeman J.W., Structuring Systems for Formal Verification, Proc 1985 Symposium on Security & Privacy, pp 2-13, IEEE Computer Society,

[Nessett 1988] Nessett D.M., The Inter-Authentication-Domain (IAD) Logon Protocol, (Preliminary Specification and Implementation Guide), Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550, 6 May 1988.

[ODP 1989] Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model, ISO/IEC JTC21/SC21 N4025, 1989

[Parker 1981] Parker D.B., Computer Security Management, Prentice-Hall, 1981

[Peterson 1981] Peterson J.L., Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

[Rangan 1988]Rangan P.V., An Axiomatic Basis of Trust in Distributed Systems, IEEE Symposium on Security & Privacy, April 1988, Oakland, CA, IEEE Computer Society Press, pp 204-211,

[Robinson 1988a] Robinson D.C., Domains: A Uniform Approach to Distributed System Management, PhD Thesis, Dept of Computing, Imperial College, London, March 1988.

[Robinson 1988b] Robinson D.C. & Sloman M.S., Domains: A New Approach to Distributed System Management, IEEE Distributed Computing Systems Workshop, Hong Kong Sept 1988.

[Satyanarayanan 1989] Satyanarayanan M., Integrating Security in a Large Distributed System, ACM Transactions on Computer Systems, vol 7 no 3 (Aug 1989), pp 247-280.

[Scruton 1983] Scruton R., A Dictionary of Political Thought, Pan Books, London, 1983.

[Sloman 1989]Sloman M.S. & Moffett J.D., Domain Management for Distributed Systems, in Meandzija & Westcott (eds), Proc of the IFIP Symposium on Integrated Network Management, Boston, USA, May 1989, North Holland, pp 505-516.

[Sloman 1990]Sloman M.S., Management for Open Distributed Processing, to appear in Proceeding of the Second Workshop on the Future Trends of Distributed Computer Systems in the 1990s, Cairo, Sept 1990. IEEE Computer Society.

[Snyder 1981] Snyder L., Formal Models of Capability-Based Protection Systems, IEEE Trans on Computers, vol 30 no 3, pp 172-181 (March 1981).

[Spivey 1988] The *fuzz* Manual, J.M. Spivey Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 1988

[Spivey 1989] Spivey, J.M., The Z Notation: A Reference Manual, Prentice Hall 1989.

[Steiner 1988] Steiner J.G., Neuman B.C. & Schiller J.I., Kerberos: An Authentication Service for Open Network, Systems, Winter Usenix 1988, Dallas TX. Project Athena, MIT, Cambridge MA 02139, USA, 12 Jan 1988.

[Stepney 1987] Stepney S. & Lord S.P., A Formal Model of Access Control, Software - Practice & Experience, vol 17 no 9 (Sept 1987), pp 575-593.

[Twidle 1988] Twidle K. & Sloman M.S., Domain Based Configuration & Name Management, for Distributed Systems, IEEE Distributed Computing Systems Workshop, Hong Kong Sept 1988.

[Tygar 1987] Tygar, J.D. & Wing J.M., Visual Specification of Security Constraints, IEEE Workshop on Visual Languages, Linkoping, Sweden, Aug 1987, pp 288-301.

[Vinter 1988] Vinter S.T., Extended Discretionary Access Controls, IEEE Symposium on Security & Privacy, April 1988, Oakland, CA, IEEE Computer Society Press, pp 39-49.

[Voydock 1983] Voydock V.L. & Kent S.T., Security Mechanisms in High-Level Protocols, ACM Computing Surveys, vol 16 no 2, June 1983, pp 135- 171.

[Waldron 1978] Waldron R.S., Practical Auditing, HFL (Publishers) Ltd, London, 1978.

[Weber 1948] From Max Weber, ed Gerth H.H. & Mills C.W., Routledge & Kegan Paul, London 1948.

[Yu 1989] Yu, Che-Fn, Access Control and Authorization Plan for Customer Control of Network Services, Proceedings of IEEE GLOBECOM '89, Dallas, Texas November 1989, pp 862-869.

APPENDIX A - A FORMAL SPECIFICATION

This specification has been written in the formal language Z [Spivey 1989]. The original version was type-checked using the Fuzz type checker [Spivey 1988]. Two changes have been made for this thesis.

- Fuzz uses an idiosyncratic dialect of Latek, specified in ASCII. This has been transcribed into a Macintosh document, using a Z font kindly made available by Nigel Day of TopExpress and subsequently edited by us.
- Repetitions have been removed where necessary and there has been some reordering to improve readability. These are noted in the text.

A.1 Definition of Objects

We view objects intuitively as having (unique) identities, together with a set of attributes, which are visible state. They may also, of course, have invisible state, but this is not used or defined in this specification.

Identities are defined as a basic type.

[Obj_Ids]

A.1.1 Object Types

We take a simple view of object types. The fundamental object type is defined by the Object schema. An object type is defined by an axiomatic definition which refers to another definition and defines the additional attributes and predicates for it. The type has all the attributes, and must conform to all the predicates, of the type it is based on, and is referred to here as a subtype. Thus, in this specification Object has two subtypes, DOMAIN and ACCESS_RULE, and DOMAIN has a ROLE_DOMAIN subtype. We also define a dummy type of ANY_TYPE, for indicating when all types are acceptable. Applications will, of course, extend the set of types.

Type_Ids::= ANY_TYPE | DOMAIN | ROLE_DOMAIN | ACCESS_RULE

Types_Of: Type_Ids $\longrightarrow \mathbb{P}$ Type_Ids

Types_Of(DOMAIN) = { DOMAIN, ROLE_DOMAIN }

A.1.2 Operation Identities

Operations can affect the world of objects. We view an operation as having four components:

- The identity of the User object which issues a request for an operation by sending an operation message.
- The identity of the Target object to which the message is directed.
- The Operation Identity part of the message contents. If the request is to be successful this must be one of the operations exported by the Target object type. It is defined here as a function Type_Ops.
- A set of Operation parameters (possibly empty) as part of the message contents. In this specification they are modelled by declared variables in the operation schemas which end with a '?'.

The set of possible operation identities is defined as a free type. Operations may be associated with parameters, but these are described separately in the schemas which define

each operation. We also define a dummy operation of ANY_OP, for indicating when all types are acceptable.

```
Op_Ids::= ANY_OP | CREATE | DESTROY |
        DOM_INCLUDE_OBJECTS | DOM_REMOVE_OBJECTS |
        ALTER_DOMAIN_SET | DOM_READ_OBJECTS | RDOM_ALTER
```

For each object type there is a set of possible operations, defined by the function Type_Ops.

$$Type\_Ops: Type\_Ids \longrightarrow \mathbb{P}\ Op\_Ids$$

A.1.3 Attributes

Attributes are defined as a Free Type. Note that we have delayed the introduction of Dom_Expr, a domain expression, to section A.2.2.

```
Attrs::=
        Type_Id ⩤Type_Ids⩥|
        Domain_Set ⩤ℙ Obj_Ids⩥|
        Type_Set ⩤ℙ Type_Ids⩥|
        Object_Set ⩤ℙ Obj_Ids⩥|
        User_Domain ⩤Dom_Expr⩥|
        Target_Domain ⩤Dom_Expr⩥|
        Operation_Set ⩤ℙ Op_Ids⩥|
        Owner_Scope ⩤Dom_Expr⩥|
        Manager_Scope ⩤Dom_Expr⩥|
        SA_User_Scope ⩤Dom_Expr⩥|
        SA_Target_Scope ⩤Dom_Expr⩥
```

A.1.4 Objects

Objects are defined to have an identity, Obj_Id, and a set of attributes. All objects have two standard attributes: Type_Id (type identity) and Domain_Set (the set of identities of domains of which the object is a direct member). Additional attributes of subtypes of objects are declared in the axiomatic definitions defining the subtypes.

Object

$$Obj\_Id: Obj\_Ids$$

$$attr\_set: \mathbb{P}\ Attrs$$

$$\exists\ x: Type\_Ids; y: \mathbb{P}\ Obj\_Ids \bullet Type\_Id\ (x) \in attr\_set \land x \bullet ANY\_TYPE \land Domain\_Set\ (y) \in attr\_set$$

The world is the totality of objects (not of facts!). No two distinct objects in the world have the same identity.

World
Objects: ℙ Object
∀ Obj1, Obj2: Object |
Obj1 ∈ Objects ∧ Obj2 ∈ Objects •
Obj1.Obj_Id = Obj2.Obj_Id ⇒ Obj1 = Obj2

This assumption of uniqueness of object identities creates potential problems in a system in which there may not be global knowledge of objects. Uniqueness can be ensured by schemes which ensure that different parts of the system are allocated disjoint ranges of identifier from which to assign new object identities. We do not discuss this further in this specification. Note that this is only a problem for new objects; we specify below that an operation to update an existing object must always preserve its identity unchanged.

Note also that this specification deals entirely with identities, and not at all with the (possibly multiple) names by which an object is referred to through a user interface. Names are of course essential in a system and the schema describing an object would need to be expanded to include them. However, we can describe the properties of access control and authority without the use of object names.

The standard ΔWorld and ΞWorld schemas define the states of the world before and after operations.

ΔWorld
World
World'

ΞWorld
ΔWorld
ΘWorld = ΘWorld'

A.1.5 Attribute Functions

These functions are used to obtain the attribute value from a named attribute. A function is defined for every attribute which we have defined, but for brevity only the first is shown here. The remainder are defined similarly.

Type_Id_Of: Object ⟶ Type_Ids
∀ x: Object; y: Type_Ids • y = Type_Id_Of (x) ⟺
Type_Id (y) ∈ x.attr_set

We also can express that there can only be one attribute in an object for each attribute name. This too we only state for Type_Id, for brevity.

$\forall$ x: Object; y, z: Type_Ids •
Type_Id (y) $\in$ x.attr_set $\wedge$ Type_Id (z) $\in$ x.attr_set $\Rightarrow$ y = z

The relation Valid_Types determines whether a type is valid, taking into account ANY_TYPE, and the fact that a DOMAIN type is always valid in a domain.

Valid_Types: $\mathbb{P}$ Type_Ids $\leftrightarrow$ Type_Ids
$\forall$ x: $\mathbb{P}$ Type_Ids; y: Type_Ids •
(x, y) $\in$ Valid_Types $\Leftrightarrow$
ANY_TYPE $\in$ x $\vee$ y $\in$ x $\vee$ y $\in$ Types_Of (DOMAIN)

A.2 Domains

A.2.1 Domain Objects

A domain is an object with DOMAIN as its type identifier and two additional attributes: Type_Set is the set of object types (in addition to domain objects) permitted in its object set, and Object_Set is the set of identities of objects which are defined to be members of the domain. It is generally intended that all object identifiers in a domain's object set should identify existing objects of the permitted type. However, these conditions may be impossible to maintain in practice, and therefore are not included as predicates here, but in the schema Domain_Include_Objects for the operation to include objects. The Domain_World axiomatic definition applies to DOMAIN and its subtypes, while Plain_Domain_Objects defines the type identity of a simple domain.

Domain_World
World
Domain_Objects: $\mathbb{P}$ Object
$\forall$ obj: Object • obj $\in$ Domain_Objects $\Leftrightarrow$
Type_Id_Of (obj) $\in$ Types_Of (DOMAIN) $\wedge$
($\exists$ y: $\mathbb{P}$ Type_Ids; z: $\mathbb{P}$ Obj_Ids •
y = Type_Set_Of (obj) $\wedge$
z = Object_Set_Of (obj))

Domain_World
Plain_Domain_Objects: $\mathbb{P}$ Object
$\forall$ obj: Object • obj $\in$ Plain_Domain_Objects $\Leftrightarrow$
obj $\in$ Domain_Objects $\wedge$
Type_Id_Of (obj) = DOMAIN

The primary operations which can be performed upon a domain object are defined.

Type_Ops(DOMAIN) = { CREATE, DESTROY,
DOM_INCLUDE_OBJECTS, DOM_REMOVE_OBJECTS, DOM_READ_OBJECTS }

A.2.2 Domain Membership

We recognise three kinds of membership of a set of objects, each applicable for particular purposes.

Direct Domain Membership

Direct domain membership is the simplest. An object is a direct member of a domain if its object identity is in the domain's object set. We do not need to define a separate function for the purpose.

Indirect Domain Membership

An object X is an indirect member of a domain Y if it is a direct member of Y or an indirect member of a domain object Z which is itself a direct member of Y. Also, if X is itself a domain object, it is an indirect member of itself.

World

Memb_Ids_Of: Obj_Ids ⟶ ℙ Obj_Ids

∀ dom1: Object; y: Obj_Ids • y ∈ Memb_Ids_Of(dom1.Obj_Id) ⟺
y ∈ Object_Set_Of (dom1) ∨
(∃ dom2: Object | dom2 ∈ Objects •
y ∈ Memb_Ids_Of (dom2.Obj_Id) ∧
dom2.Obj_Id ∈ Object_Set_Of (dom1)) ∨
dom1 ∈ Domain_Objects ∧ y = dom1

Domain Expressions

A domain expression is the most powerful means we have of defining a set of objects. It may consist of:

- NULL, which returns an empty set.
- An object identity (De_Object), which returns the object identity itself.
- A domain identity (De_Domain), which returns all direct and indirect members of the domain.
- A domain identity (De_Dom_Restr), which returns only direct members.of the domain.
- A set of domain expressions (De_Set), which returns the identities of objects which are returned by any of the expressions.
- The difference between two domain expressions (De_Diff), which returns the objects which are returned by the first expression but not by the second.

The intersection of two domain expressions, whose membership is the objects which are members of both the first expression and the second can be expressed in terms of difference through the identity

$$A \cap B \equiv A \setminus (A \setminus B).$$

The expression itself is a Free Type, which is the means provided in Z of effectively allowing a multiplicity of types for a single identifier.

```
Dom_Expr::= NULL |
        De_Object ⩤Obj_Ids⩥ |
        De_Domain ⩤Obj_Ids⩥ |
        De_Dom_Rstr ⩤Obj_Ids⩥ |
        De_Set ⩤ℙ Dom_Expr⩥ |
        De_Diff ⩤Dom_Expr • Dom_Expr⩥
```

Domain Expression Evaluation: This axiomatic definition defines the set of object identities returned by a domain expression.

```
Dom_Expr_Membs: Dom_Expr ⟶ ℙ Obj_Ids
─────────────────────────────────────────────────────────────
Dom_Expr_Membs(NULL) = { }
∀ x: Dom_Expr; y: Obj_Ids • y ∈ Dom_Expr_Membs(x) ⟺
─────────────────────────────────────────────────────────────
    (∃ obj_id: Obj_Ids •
        x = De_Object(obj_id) ∧ y = obj_id) ∨
    (∃ obj_id: Obj_Ids •
        x = De_Domain(obj_id) ∧ y ∈ Memb_Ids_Of(obj_id))) ∨
    (∃ obj_id: Obj_Ids; obj: Object •
        x = De_Dom_Rstr(obj_id) ∧ obj.Obj_Id = obj_id ∧
─────────────────────────────────────────────────────────────
        y ∈ Object_Set_Of(obj))) ∨
    (∃ expr_set: ℙ Dom_Expr • x = De_Set (expr_set) ∧
        (∃ dom_expr: Dom_Expr •
─────────────────────────────────────────────────────────────
            dom_expr ∈ expr_set ∧ y ∈ (Dom_Expr_Membs(dom_expr))) ∨
    (∃ expr1, expr2: Dom_Expr • x = De_Diff (expr1, expr2) ∧
─────────────────────────────────────────────────────────────
        y ∈ (Dom_Expr_Membs(expr1) \ Dom_Expr_Membs(expr2)))
```

A.3 Operations

A.3.1 Operation Requests

An operation request, at a minimum, defines the identities of the user requesting it, the target object to which it is directed, and the name of the operation which is requested.

```
Operation_Request
─────────────────────────────────────────────────────────────
user_id?, target_id?: Obj_Ids
requested_op?: Op_Ids
─────────────────────────────────────────────────────────────
```

A.3.2 Operation Semantics

The world may be changed by carrying out operations, by means of a successful operation request. To be feasible it must meet at least the following pre-conditions: it is authorised, the invoking user and the target object exist, and the operation is possible for the type of target object. The target object continues to exist after the operation, with an unchanged identity. This is consistent with our view that all objects, except root objects, exist within domains, and all operations to create and destroy objects are viewed as operations directed to domains which contain the objects.

Feasible_Operation

```
ΔWorld
Operation_Request
user, target, target': Object
user.Obj_Id = user_id?
user ∈ Objects
target ∈ Objects
target' ∈ Objects'
target.Obj_Id = target_id?
target'.Obj_Id = target.Obj_Id
requested_op? ∈ Type_Ops ( Type_Id_Of ( target ) )
```

A.3.3 General Properties of Operations on Objects

We define operations to create and destroy objects.

Create or Destroy an Object

The target object, in a Create or Destroy operation, is an existing domain, and the only effect on its attributes is to alter its Object_Set.

Object_Cr_Destr

```
Feasible_Operation
target ∈ Domain_Objects
target ∈ Objects
∃ x, y: ℙ Obj_Ids •
    Object_Set (x) ∈ target.attr_set ∧
    Object_Set (y) ∈ target'.attr_set ∧
    target.attr_set \ { Object_Set (x) } =
        target'.attr_set \ { Object_Set (y) }
```

Create an Object

The following schema expresses what is true of any operation to create an object. The target object of the operation is the domain which will contain the object. Parameters supplied must include an object identity new_obj_id? and a type identity type_id?.

No object must already exist which already has that identity, and the object's type must be one of the permitted types for the target domain. The target domain's object set is updated to include the new object's identity, and the domain set of the new object includes the target domain's identity. After the operation the world contains the new object.

Object_Create

```
Object_Cr_Destr
new_object: Object
new_obj_id?: Obj_Ids
type_id?: Type_Ids
requested_op? = CREATE
type_id? • ANY_TYPE
∃ x: ℙ Type_Ids •
Type_Set_Of (target, type_id?) ∈ Valid_Types
¬(∃ obj: Object | obj ∈ Objects • obj.Obj_Id = new_obj_id? )
Object_Set_Of (target') =
      Object_Set_Of (target) ∪ { new_obj_id? }
new_object.Obj_Id = new_obj_id?
Type_Id _Of(new_object) = (type_id? )
Domain_Set_Of(new_object) = { target_id? }
Objects' = (Objects \ { target }) ∪ { target', new_object }
```

Destroy an Object

The following schema expresses what is true of any operation to destroy an object. The target object of the operation is a domain which contains the object. The object identity destr_obj_id? must be supplied as a parameter. The target domain's object set is updated to remove the destroyed object's identity, and the domain set of the new object includes the target domain's identity. After the operation the world no longer contains the destroyed object.

Note that we define no constraints here which prevent destruction of an object which is still contained in another domain. This may be very undesirable in many systems, and application-specific constraints may be required to prevent it.

Object_Destroy

Object_Cr_Destr
destr_object: Object
destr_obj_id?: Obj_Ids
requested_op? = DESTROY destr_object ∈ Objects
destr_object.Obj_Id = destr_obj_id?
Object_Set_Of (target') = Object_Set_Of (target) \ {destr_obj_id?} Objects' = (Objects \ {target, destr_object}) ∪ {target'}

A.3.4 Operations on Domains

CREATE and DESTROY are in any case operations on domains, and we define here the semantics of creation and destruction of domain objects by an operation on the parent domain. We also define operations to include and remove object identities in domains, and to read a domain's object set.

Create a Domain

A new domain can be created with specified object types. It is initially created with an empty object set. The schema is based on the general Object_Create schema, replacing the declaration of new_object with one which ensures that the predicates for domain objects are expressed.

Domain_Create

Object_Create
type_set?: ℙ Type_Ids
type_id? ∈ Types_Of(DOMAIN) ⇒ Object_Set ({ }) ∈ new_object.attr_set ∧ Type_Set (type_set?) ∈ new_object.attr_set

Destroy a Domain

We have not specified a constraint that a domain can only be destroyed when its object set is empty. If this constraint is not met, then there is the risk that objects which are not members of another domain will effectively be destroyed, but we regard this as an application level decision. No specific schema is needed.

Changing a Domain's Object Set

Inclusion and removal of objects into/from a domain are similar operations. This schema defines their most important common characteristic, that the only attribute of the domain which is altered is the object set.

ΔDomain_Object_Set

Feasible_Operation

member, member': Object

member_id?: Obj_Ids

before_obj_set, after_obj_set: ℙ Obj_Ids

target ∈ Domain_Objects
member.Obj_Id = member_id?

member'.Obj_Id = member.Obj_Id

Object_Set_Of (target) = before_obj_set

Object_Set_Of (target') = after_obj_set

target.attr_set \ { Object_Set (before_obj_set) } =
target'.attr_set \ { Object_Set (after_obj_set) }

Include Object in a Domain

When an object is included in a domain, its membership of other domains is not affected; it is included in the object set of this domain, provided it exists and its type is valid. The domain's identity is included in the object's domain set. The remainder of object's attributes are unaltered. We do not insist in this specification that none of the included identities should previously have been in the domain's object set.

Domain_Include_Object

ΔDomain_Object_Set

requested_op? = DOM_INCLUDE_OBJECTS
member ∈ Objects

(Type_Set_Of (target), Type_Id_Of (member)) ∈ Valid_Types

after_obj_set = before_obj_set ∪ { member_id? }

Domain_Set_Of (member') = Domain_Set_Of (member) ∪ { target.Obj_Id }

(∃ x, y: ℙ Obj_Ids • member.attr_set \ { Domain_Set (x) } =
member'.attr_set \ { Domain_Set (y) })

Objects' = (Objects \ {target, member }) ∪ {target', member' }

Remove Object from a Domain

Removal of an object from a domain is carried out by removing its name from the object set. If it is not still a member of another domain's object set it will then effectively be destroyed. There could be an application level constraint to prevent destruction of a domain before its object set is empty. If the removed object exists its domain set is updated by removal of the target domain's identity. We do not insist in this specification that all the removed identities should previously have been in the domain's object set.

Domain_Remove_Object

ΔDomain_Object_Set
requested_op? = DOM_REMOVE_OBJECTS after_obj_set = before_obj_set \ { member_id? }
member ∈ Objects ⇒
Domain_Set_Of (member') =
Domain_Set_Of (member) \ { target.Obj_Id } ∧
(∃ x, y: ℙ Obj_Ids •
member.attr_set \ { Domain_Set (x) } =
member'.attr_set \ { Domain_Set (y) }) ∧
Objects' = (Objects \ { target, member }) ∪ { target', member' }
member ∉ Objects ⇒
Objects' = (Objects \ {target}) ∪ {target'}

Reading a Domain's Object Set

This operation returns, in the object_set! parameter, the identities in the domain's object set.

Read_Domain_Object_Set

Feasible_Operation
ΞWorld
object_set!: ℙ Obj_Ids
target ∈ Domain_Objects target ∈ Objects
requested_op? = DOM_READ_OBJECTS Object_Set (object_set!) ∈ target.attr_set

Blank Page

A.4 Access Rules

A.4.1 Access Rule Objects

An access rule object, whose type is ACCESS_RULE, has three additional attributes: User_Domain and Target_Domain specify sets of object by means of domain expressions, and Operation_Set specifies a set of operation identities.

```
Access_Rule_Objects: ℙ Object
∀ obj: Object • obj ∈ Access_Rule_Objects ⟺
    Type_Id_Of (obj) = ACCESS_RULE ∧
    (∃ x, y: Dom_Expr; z: ℙ Op_Ids •
        x = User_Domain_Of (obj) ∧
        y = Target_Domain_Of (obj) ∧
        z = Operation_Set_Of (obj) )
```

A.4.2 Authorised Operations

We assume that the reference monitor intercepts all operation requests, and decides whether to allow the request on the basis of access rules. We take a straightforward view of access rules, viewing them directly in terms of operation identities. It would be possible to extend the specification to define the authority identities contained in access rules differently from operation identities. One reason for this is that we may wish to have a policy that a single authority specification gives authority for more than one operation, e.g. that authority for a Write operation on a particular attribute should also give authority to perform a Read operation on it. We would then have a mapping between authorities and operations. We do not include this here, for simplicity.

A user who issues an operation request is authorised for a (target, operation identity) combination if there is an access rule where the user is in the identities defined by the User Domain expression, the target object is in the identities defined by the Target Domain expression, and the operation identity is in the Operation Set of the access rule.

```
Authorised_Op_Request: Obj_Ids ⟷ (Obj_Ids • Op_Ids)
∀ user, target: Obj_Ids; op_id: Op_Ids •
    (user, (target, op_id)) ∈ Authorised_Op_Request ⟺
        (∃ ar: Object | ar ∈ Objects •
            user ∈ Dom_Expr_Membs (User_Domain_Of(ar)) ∧
            target ∈ Dom_Expr_Membs (Target_Domain_Of (ar)) ∧
            (op_id ∈ Operation_Set_Of (ar) ∨
            ANY_OP ∈ Operation_Set_Of (ar) ))
```

We extend our view of the world to take this into account.

Authorised_Operation
Feasible_Operation
(user_id?, (target_id?, requested_op?)) ∈ Authorised_Op_Request

Not all operations are issued directly by users. A consequence for the reference monitor is that it must recognise that certain user requests will cause further operation requests to be issued, and validate the original request for them also. For example, a user request to perform DOMAIN_INCLUDE_OBJECTS will result in ALTER_DOMAIN_SET operation requests, to update the Domain Set of each member object. This requires authority.

ΔDomain_Object_Set_Authorised
ΔDomain_Object_Set
∀ member_id?: Obj_Ids | member_id? ∈ member_ids? •
(user_id?, (member_id?, ALTER_DOMAIN_SET)) ∈
Authorised_Op_Request

A.4.3 Operations on Access Rules

For simplicity we define all operations on access rules as CREATE and DESTROY operations on parent domains. Operations for alteration of access rules could also be defined, but it may be found adequate to rely on their creation and deletion for this purpose.

Security Administrator Authorisation

Security Administrator authorisation to create and destroy access rules is required. The user must be in the Object_Set of a ROLE_DOMAIN which has:

- A SA_User_Scope which is a superset of all the objects in the User_Domain of the access rule and
- A SA_Target_Scope which is a superset of all the objects in the Target Domain of the access rule.

SA_Rdom_Auth: Obj_Ids ⟷ (Dom_Expr • Dom_Expr)
∀ sa: Obj_Ids; userdom, targetdom: Dom_Expr •
(sa, (userdom, targetdom)) ∈ SA_Rdom_Auth ⟺
(∃ rdom: Object | rdom ∈ Objects •
sa ∈ Object_Set_Of(rdom) ∧
Dom_Expr_Membs (userdom) ⊆
Dom_Expr_Membs (SA_User_Scope_Of (rdom)) ∧
Dom_Expr_Membs (targetdom) ⊆
Dom_Expr_Membs (SA_Target_Scope_Of (rdom)))

Create an Access Rule

Access_Rule_Create has user domain, target domain and operation set parameters for the new object.

Access_Rule_Create

Object_Create
user_domain?: Dom_Expr
target_domain?: Dom_Expr
operation_set?: ℙ Op_Ids
(user_id?, (user_domain?, target_domain?)) ∈ SA_Rdom_Auth type_id? = ACCESS_RULE
User_Domain_Of(new_object) = user_domain?
Target_Domain_Of (new_object) = target_domain? Operation_Set_Of (new_object) = operation_set?

Destroy an Access Rule

There are no parameters to this operation.

Access_Rule_Destroy

Object_Destroy
(user_id?, (User_Domain_Of(destr_object), Target_Domain_Of(destr_object))) ∈ SA_Rdom_Auth Type_Id_Of(destr_object) = ACCESS_RULE

A.5 Role Domains

A.5.1 Role Domain Objects

A role domain is a subtype of a domain with ROLE_DOMAIN as its type identifier and four additional attributes in its attribute set: Owner_Scope, Manager_Scope, SA_User_Scope and SA_Target_Scope.

Role_Domain_Objects: ℙ Object
∀ obj: Object • obj ∈ Role_Domain_Objects ⟺ obj ∈ Domain_Objects ∧
Type_Id_Of (obj) = ROLE_DOMAIN ∧
(∃ w, x, y, z: Dom_Expr •
x = Owner_Scope_Of (obj) ∧
x = Manager_Scope_Of (obj) ∧
x = SA_User_Scope_Of (obj) ∧ x = SA_Target_Scope_Of (obj))

A.5.2 Operations on Role Domains

We take the simplest approach to operations on role domains. We need to have separate operations to update each kind of scope attribute because of their separate authority

requirements. We therefore simplify the create and destroy operations by insisting that they are carried out on role domains with null scope attributes.

Type_Ops(ROLE_DOMAIN) • { RDOM_ALTER }

Create a Role Domain

Role domains are created with empty scope fields, without any special authority required.

Rdom_Create

Domain_Create
type_id? = ROLE_DOMAIN Owner_Scope_Of (new_object) = NULL
Manager_Scope_Of (new_object) = NULL
SA_User_Scope_Of (new_object) = NULL SA_Target_Scope_Of (new_object) = NULL

Destroy a Role Domain

Role domains are destroyed with empty scope fields, without any special authority required.

Rdom_Destroy

Object_Destroy
Type_Id_Of(destr_object) = ROLE_DOMAIN Owner_Scope_Of (destr_object) = NULL
Manager_Scope_Of (destr_object) = NULL
SA_User_Scope_Of (destr_object) = NULL SA_Target_Scope_Of (destr_object) = NULL

Owner & Manager Authority

A user has owner (or manager) authority over a domain expression, such as x_Scope, if it object identity is in the object set of a role domain with an Owner_Scope (or Manager_Scope) which is a superset of all the objects in the domain expression.

Owner_Rdom_Auth: Obj_Ids ⟷ Dom_Expr
∀ owner: Obj_Ids; de: Dom_Expr • (owner, de) ∈ Owner_Rdom_Auth ⟺ (∃ rdom: Object | rdom ∈ Objects • owner ∈ Object_Set_Of(rdom) ∧ Dom_Expr_Membs (de) ⊆ Dom_Expr_Membs (Owner_Scope_Of (rdom)))

Manager_Rdom_Auth: Obj_Ids ⟷ Dom_Expr
∀ manager: Obj_Ids; de: Dom_Expr • (manager, de) ∈ Manager_Rdom_Auth ⟺ (∃ rdom: Object | rdom ∈ Objects • manager ∈ Object_Set_Of(rdom) ∧ Dom_Expr_Membs (de) ⊆ Dom_Expr_Membs (Manager_Scope_Of (rdom)))

Alter Role Domain Scope Attributes

If a scope attribute in a role domain is to be altered, both the current value of it, and the new value, must be within the authority of the alterer. 'Owner' is required for all objects in Owner_Scope or Manager_Scope, and 'manager' is required for all objects in SA_User_Scope or SA_Target_Scope.

Rdom_Alter_Owner_Scope

Feasible_Operation
owner_scope?: Dom_Expr
requested_op? = RDOM_ALTER Type_Id_Of(target) = ROLE_DOMAIN
(user_id?, Owner_Scope_Of (target)) ∈ Owner_Rdom_Auth
(user_id?, owner_scope?) ∈ Owner_Rdom_Auth
Owner_Scope_Of(target') = owner_scope?
∃ x, y: Dom_Expr •
target.attr_set \ { Owner_Scope (x) } =
target'.attr_set \ { Owner_Scope (y) }
Objects' = (Objects \ { target }) ∪ { target' }

Rdom_Alter_Manager_Scope

Feasible_Operation
manager_scope?: Dom_Expr
requested_op? = RDOM_ALTER Type_Id_Of(target) = ROLE_DOMAIN
(user_id?, Manager_Scope_Of (target)) ∈ Owner_Rdom_Auth
(user_id?, manager_scope?) ∈ Owner_Rdom_Auth
Manager_Scope_Of(target') = manager_scope?
∃ x, y: Dom_Expr •
target.attr_set \ { Manager_Scope (x) } =
target'.attr_set \ { Manager_Scope (y) }
Objects' = (Objects \ { target }) ∪ { target' }

Rdom_Alter_SA_User_Scope

Feasible_Operation
sa_user_scope?: Dom_Expr
requested_op? = RDOM_ALTER Type_Id_Of(target) = ROLE_DOMAIN
(user_id?, SA_User_Scope_Of (target)) ∈ Manager_Rdom_Auth
(user_id?, sa_user_scope?) ∈ Manager_Rdom_Auth
SA_User_Scope_Of(target') = sa_user_scope?
∃ x, y: Dom_Expr •
target.attr_set \ { SA_User_Scope (x) } =
target'.attr_set \ { SA_User_Scope (y) }
Objects' = (Objects \ { target }) ∪ { target' }

Rdom_Alter_SA_Target_Scope

Feasible_Operation

sa_target_scope?: Dom_Expr

requested_op? = RDOM_ALTER
Type_Id_Of(target) = ROLE_DOMAIN

(target_id?, SA_Target_Scope_Of (target)) ∈ Manager_Rdom_Auth

(target_id?, sa_target_scope?) ∈ Manager_Rdom_Auth

SA_Target_Scope_Of(target') = sa_target_scope?

∃ x, y: Dom_Expr •
    target.attr_set \ { SA_Target_Scope (x) } =
        target'.attr_set \ { SA_Target_Scope (y) }
Objects' = (Objects \ { target }) ∪ { target' }

A.6 Authority & Access Relations

These relations, and the transitions between them, express the abstract conditions which must be met for the delegation of authority. We can demonstrate their security properties more simply than when dealing with complex objects such as role domains and access rules. In the next section we will show how the complex objects map onto these simple relations, and that therefore they retain their security properties.

A.6.1 Authority & Access Relations

There are four kinds of authority: Owner, Manager, SA_User and SA_Target.

Auth_Ids::= Owner_Auth | Manager_Auth | SA_User_Auth | SA_Target_Auth

The authority relation expresses the authority which a user object has over another user or target object. It is also defined in a decorated version to enable us to express its alteration.

Auth, Auth': Obj_Ids • Obj_Ids • Auth_Ids

An access relation expresses the relationship between a user object identity and a target object together with an operation id.

Access_Auth, Access_Auth': Obj_Ids ⟷ (Obj_Ids • Op_Ids)

A.6.2 Giving Authority & Giving Access

Giving authority is represented by a transition as a result of which there is a new authority relation. The predicates satisfied by these abstract transitions must be satisfied by any concrete representation, e.g. role domains. The predicates in them are the specific policy expressed in this specification, and could be different. They are:

- An Owner of an object can give Owner authority over it to another user.
- An Owner of an object can give Manager authority over it to another user.

- A Manager of an object can give SA_User authority over it to another user.
- A Manager of an object can give SA_Target authority over it to another user.

Give_Auth

source_user?, target_user?, target_object?: Obj_Ids

auth?: Auth_Ids

((source_user?, target_object?, Owner_Auth) ∈ Auth ∧
(auth? = Owner_Auth ∨ auth? = Manager_Auth)) ∨
((source_user?, target_object?, Manager_Auth) ∈ Auth ∧
(auth? = SA_User_Auth ∨ auth? = SA_Target_Auth))

Auth' = Auth ∪ { (target_user?, target_object?, auth?) }

Giving access is represented by a transition as a result of which there is a new access relation. A user can give another user authority to perform any operation on an object only if he has SA_User authority over that user and SA_Target authority over the object. The predicates satisfied by this abstract transition must be satisfied by any concrete representation.

Give_Access_Auth

source_user?, target_user?, target_object?: Obj_Ids

op?: Op_Ids

(source_user?, target_user?, SA_User_Auth) ∈ Auth
(source_user?, target_object?, SA_Target_Auth) ∈ Auth

Access_Auth' = Access_Auth ∪
{ (target_user?, (target_object?, op?)) }

It can immediately be verified by inspection that Give_Auth and Give_Access_Auth between them meet the requirement of *hierarchical authority, continuous chain of authority* and *giving authority for an operation* which are defined in section IV.7.5. We cannot of course guarantee that any given system will meet the remaining requirement, *initial authority*. Removal of authority and access are dealt with similarly.

A.7 Mapping from Role Domains and Authority/Access Relations

A.7.1 Mappings

The authority relation maps onto role domains as follows. (user, target object, Owner_Auth) is in the Owner_Auth relation if there is a role domain object such that the user is in its Object_Set and the target object is in its Owner_Scope. The other mappings are similar.

∀ x, y: Obj_Ids • (x, y, Owner_Auth) ∈ Auth ⇔
(∃ rdom: Object | rdom ∈ Objects •
x ∈ Object_Set_Of(rdom) ∧
y ∈ Dom_Expr_Membs (Owner_Scope_Of (rdom)))

$\forall$ x, y: Obj_Ids • (x, y, Manager_Auth) $\in$ Auth $\Longleftrightarrow$
($\exists$ rdom: Object | rdom $\in$ Objects •
x $\in$ Object_Set_Of(rdom) $\wedge$
y $\in$ Dom_Expr_Membs (Manager_Scope_Of (rdom)))

$\forall$ x, y: Obj_Ids • (x, y, SA_User_Auth) $\in$ Auth $\Longleftrightarrow$
($\exists$ rdom: Object | rdom $\in$ Objects •
x $\in$ Object_Set_Of(rdom) $\wedge$
y $\in$ Dom_Expr_Membs (SA_User_Scope_Of (rdom)))

$\forall$ x, y: Obj_Ids • (x, y, SA_Target_Auth) $\in$ Auth $\Longleftrightarrow$
($\exists$ rdom: Object | rdom $\in$ Objects •
x $\in$ Object_Set_Of(rdom) $\wedge$
y $\in$ Dom_Expr_Membs (SA_Target_Scope_Of (rdom)))

The access relation maps onto access rules as follows. A (user, target object, operation) is in the Access_Auth relation if there is an access rule object such that the user is in its User_Domain, the target object is in its Target_Domain and the operation is in its Operation_Set.

$\forall$ x, y: Obj_Ids; z: Op_Ids • (x, y, z) $\in$ Access_Auth $\Longleftrightarrow$
($\exists$ ar: Object | ar $\in$ Objects •
x $\in$ User_Domain_Of(ar) $\wedge$
y $\in$ Target_Domain_Of(ar))
z $\in$ Operation_Set_Of(ar))

A.7.2 Proof Summaries

We here summarise the proofs that role domains, as defined, conform to the requirements of *hierarchical authority* and *continuous chain of authority* which are defined in section IV.7.5. We show here the proof outline for giving Owner authority using Rdom_Alter_Owner_Scope. The other cases, for Rdom_Alter_Manager_Scope, Rdom_Alter_SA_User_Scope, and Rdom_Alter_SA_Target_Scope, are similar.

a) If (userA, dom_expr) is in the Owner_Rdom_Auth relation, then (userA, objB, Owner_Auth) is in the Auth relation where objB is any member of dom_expr.

   This follows straightforwardly from the definitions. For if (userA, dom_expr) is in Owner_Rdom_Auth then userA is a member of some role domain whose Owner_Scope is a superset of dom_expr, which contains objB. It follows immediately from the definition of the mapping from Owner_Rdom_Auth to Auth that (userA, objB, Owner_Auth) is a member of Auth.

b) If the operation Rdom_Alter_Owner_Scope is authorised for userA, then the predicates for the Give_Auth transition to add a new Owner_Auth member to the Auth relation are satisfied.

   This also follows straightforwardly from the definitions. The operation requires that for both the current and new values of Owner_Scope that (userA, dom_expr) must be in Owner_Rdom_Auth. We have shown that for every object objB in the domain expression defined by Owner_Scope then (userA, objB, Owner_Auth) is a member of Auth. So the predicates required by the Give_Auth transition when auth? = Owner_Auth are satisfied for every objB.

c) We have already shown that the Give_Auth transition satisfies the requirements. Therefore the role domains do so also.

The proofs that Access_Rule_Create and Access_Rule_Destroy conform to the predicates of Give_Access_Auth are similar, and follow from the definition of SA_Rdom_Auth and the mapping of SA_User_Scope and SA_Target_Scope onto SA_User_Auth and SA_Target_Auth. It then follows that they conform to the requirements of *Giving Authority for an Operation* defined in section IV.7.5.

A.8 Formal Interpretation of Thesis Diagrams

A.8.1 Domain Diagrams

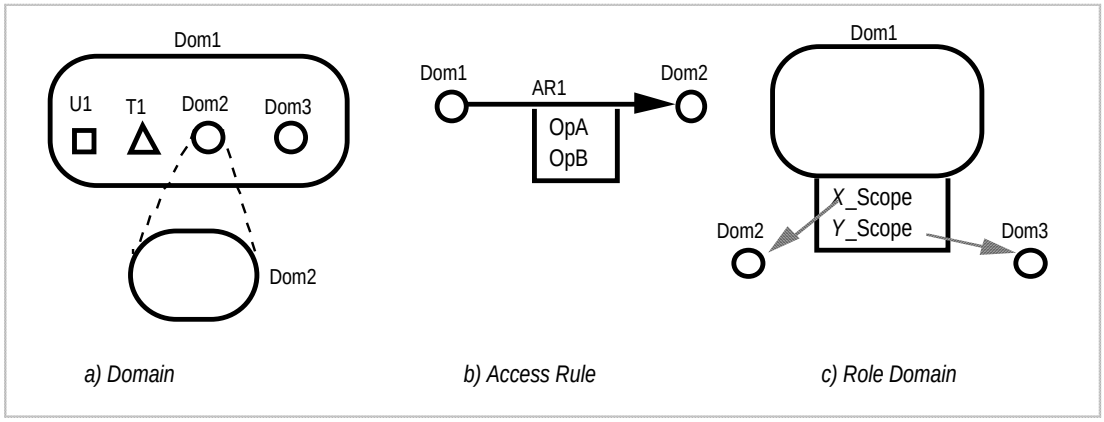

Figure A.1 Domains and Access Rules

Figure A.1 should be interpreted as follows.

a) Domain

Dom1 ∈ Domain_Objects ∧
Dom2 ∈ Domain_Objects ∧
Dom3 ∈ Domain_Objects ∧
U1 ∈ User_Objects ∧
T1 ∈ Target_Objects ∧
Object_Set_Of(Dom1) = { Dom2, Dom3, U1, T1}

We assume that the expressions User_Objects and Target_Objects have been defined appropriately in the application.

b) Access Rule

AR1 ∈ Access_Rule_Objects ∧
User_Domain_Of(AR1) = De_Domain(Dom1) ∧
Target_Domain_Of(AR1) = De_Domain(Dom2) ∧
Operation_Set_Of(AR1) = {OpA, OpB}

c) Role Domain

Dom1 ∈ Domain_Objects ∧
Dom2 ∈ Domain_Objects ∧
Dom3 ∈ Role_Domain_Objects ∧
*X*_Scope_Of(Dom1) = De_Domain(Object_Set_Of(Dom2)) ∧
*Y*_Scope_Of(Dom1) = De_Domain(Object_Set_Of(Dom3))

We assume that the *X*_Scope and *Y*_Scope attributes have been defined appropriately in the application.

A.8.2 Domain Expression

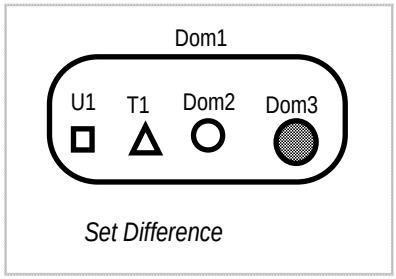

Figure A.2 Domain Expression

We have not illustrated all possible domain expressions in this thesis. Apart from straightforward domain membership, the only one we have used is set difference, illustrated in figure A.2. It means 'all the members of Dom1 less the members of Dom3'. Note that the domain expression variable has not been shown in the diagram, so we have existentially quantified it. Also, that in this example Dom3 is a member of the object set of Dom1; the domain expression would be meaningful even if this were not so.

∃ DE: Dom_Expr •
Dom1 ∈ Domain_Objects ∧
Dom2 ∈ Object_Set_Of(Dom1) ∧
Dom3 ∈ Object_Set_Of(Dom1) ∧
U1 ∈ Object_Set_Of(Dom1) ∧
T1 ∈ Object_Set_Of(Dom1) ∧
Dom_Expr_Membs(DE) =
Dom_Expr_Membs(De_Domain(Object_Set_Of(Dom1))) \
Dom_Expr_Membs(De_Domain(Object_Set_Of(Dom3)))

A.8.3 Petri Net Diagrams

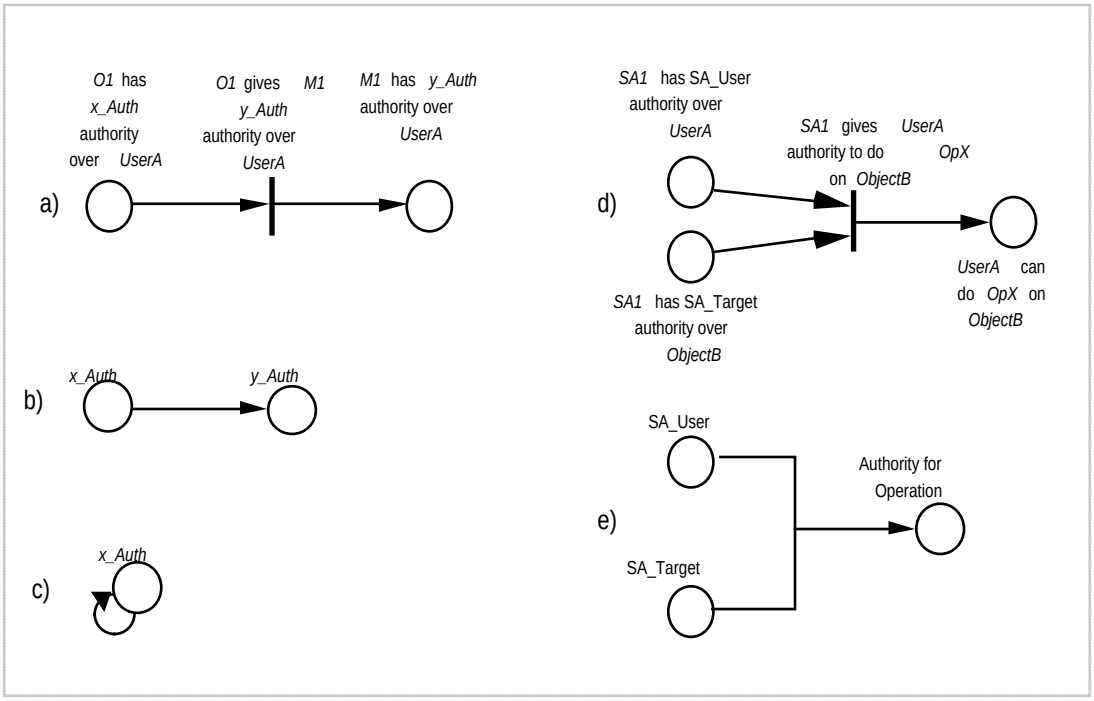

Figure A.3 Petri Net Diagrams

Figure A.3 should be interpreted as follows.

a) Delegation of Authority (Full Diagram)

(O1, UserA, x_Auth) ∈ Auth ∧
(∃ ga: Give_Auth • ga.auth? = y_Auth ∧
ga.source_user? = O1 ∧ ga.target_user? = M1 ∧
ga.target_object? = UserA) ∧
(M1, UserA, y_Auth) ∈ Auth

b) Delegation of Authority (Abbreviated)

The abbreviated diagram has implied existential quantification in it, but is otherwise identical.

∃ O1, UserA: Object; x_Auth: Auth_Ids •
((O1, UserA, x_Auth) ∈ Auth ∧
(∃ ga: Give_Auth • ga.auth? = y_Auth ∧
ga.source_user? = O1 ∧ ga.target_user? = M1 ∧
ga.target_object? = UserA) ∧
(M1, UserA, y_Auth) ∈ Auth)

c) Delegation of Authority (Abbreviated and Sharing Authority)

This abbreviated diagram is identical to b) except that authority is not being delegated downwards but is being shared, i.e. y_Auth = x_Auth, for example sharing of ownership. Posession of x_Auth is the precondition for source_user? and the postcondition for target_user? The interpretation below makes it clear that this is not a reflexive operation, as source_user? and target_user? are in general different.

∃ O1, UserA: Object; x_Auth: Auth_Ids •
((O1, UserA, x_Auth) ∈ Auth ∧
(∃ ga: Give_Auth • ga.auth? = x_Auth ∧
ga.source_user? = O1 ∧ ga.target_user? = M1 ∧
ga.target_object? = UserA) ∧
(M1, UserA, x_Auth) ∈ Auth)

d) Giving Access (Full Diagram)

(SA1, UserA, SA_User_Auth) ∈ Auth ∧
(SA1, ObjectB, SA_Target_Auth) ∈ Auth ∧
(∃ ga: Give_Access_Auth •
ga.source_user? = SA1 ∧ ga.target_user? = UserA ∧
ga.target_object? = ObjectB ∧ ga.op? = OpX) ∧
(UserA, ObjectB, OpX) ∈ Access_Auth

e) Giving Access (Abbreviated)

The abbreviated diagram has implied existential quantification in it, but is otherwise identical.

∃ SA1, UserA, ObjectB: Object; OpX: Op_Ids •
(SA1, UserA, SA_User_Auth) ∈ Auth ∧
(SA1, ObjectB, SA_Target_Auth) ∈ Auth ∧
(∃ ga: Give_Access_Auth •
ga.source_user? = SA1 ∧ ga.target_user? = UserA ∧
ga.target_object? = ObjectB ∧ ga.op? = OpX) ∧
(UserA, ObjectB, OpX) ∈ Access_Auth

APPENDIX B - PROLOG

This appendix provides a Prolog listing, with some comments, which animates the specification in Appendix A. It is derived directly form the Z description by hand-coding. The possibility of using a tool for automatic translation from Z to Prolog [Dick 1989] was investigated, but the resulting model would have run too slowly to be practical.

It is written in a conservative subset of Edinburgh syntax Prolog [Clocksin 1981] and has been run on both LPA MacPROLOG version 2.7 for the Macintosh [LPA 1987] and LPA Professional Prolog version 2.1 for IBM PC compatibles [LPA 1988] - although these both use Edinburgh syntax, their semantics differ.

B.1 General Comments

The Z specification translated very easily into Prolog, almost on a one-for-one basis. We used lists for sets, the setof construct for creating them and on for detecting membership. We elected not to use a typing field in our entities, for performance reasons, so all automatic type-checking was thereby discarded. Coming from a strongly typed specification, we feel that his economy of effort was justified.

We met problems on dynamic checking of domain membership during queries, as the evaluation space was consistently exceeded for complicated structures. There was no straightforward way of either increasing the evaluation space or of making the queries tail-recursive for efficient reuse of the space. The solution we adopted was to add two extra fields to each object, containing the complete direct and indirect membership of the object_set and domain_set of the object. These were updated whenever there was a change of domain membership; consequently it was also necessary to update the object sets of all the ancestors of the domains concerned. This has resulted in very slow performance of most operations on objects, but since we have been able to make the evaluations tail-recursive, evaluation space has not been exceeded. Time alone will tell whether these performance problems are a foretaste of ones to come in a Conic domains implementation.

B.2 The Model

Where possible we provide provide cross-references to Appendix A, rather than detailed comments.

B.2.1 Objects

Facts about Object Types

See A.1.1

```
type_id('DOMAIN').
type_id('ROLE_DOMAIN').
type_id('ACCESS_RULE').
type_id('USER').
type_id('FILE').
type_id('TARGET2').

types_of('DOMAIN', ['DOMAIN', 'ROLE_DOMAIN']).
```

Facts about Operations

See A.1.2

```
op_id('CREATE').
op_id('DESTROY').
op_id('DOM_INCLUDE_OBJECT').
op_id('DOM_REMOVE_OBJECT').
op_id('ALTER_DOMAIN_SET').
op_id('DOM_READ_OBJECTS').
op_id('RDOM_ALTER').
```

See A.2.1

```
type_ops('DOMAIN',
    ['CREATE', 'DESTROY',
    'DOM_INCLUDE_OBJECT', 'DOM_REMOVE_OBJECT', 'DOM_READ_OBJECTS']).
```

See A.5.1

```
type_ops('ROLE_DOMAIN',
    ['CREATE', 'DESTROY',
     'DOM_INCLUDE_OBJECT', 'DOM_REMOVE_OBJECT', 'DOM_READ_OBJECTS',
     'RDOM_ALTER']).
```

See A.4.3

```
type_ops('ACCESS_RULE', []).

type_ops('FILE', ['READ', 'WRITE']).
type_ops('TARGET2', ['READ', 'WRITE']).
type_ops('USER', ['READ', 'WRITE']).
```

Objects

Objects themselves are defined as facts, and their attributes are defined implicitly. No attempt is made in this program to force strong typing on the attributes, except by ensuring that the operations which manipulate objects are correct. The format of an object is shown here.

```
/*   object(_obj_id, _name, _type_id, _domain_set, _attributes, _all_objs, _all_doms)   */
```

For efficency, the attributes which are common to all objects, i.e. _name, _type_id and _domain_set, are separated out in the object relation. The _attributes field is a list of all the other attributes. _all_objs and _all_doms are fields which are included solely for convenience in the implementation. They contain all direct and indirect _object_set and _domain_set members for the object, and therfore enable queries to be answered without having to evaluate indirect members of the sets on every occasion. They are updated at the same time as the _object_set and _domain_set, and every time an _object_set is updated, _all_objs is revaluated for all the object's ancestors.

The World (A.1.4) is represented by the Prolog database, and a new state of the world is created by assert and retract operations on the database. No attempt is made to evaluate the database continuously for uniqueness of identities or uniqueness of attribute names within objects, but these are constraints on individual operations on objects.

B.2.2 Domains

See A.2.1

```
domain(_obj_id, _name, _type_id, _domain_set,
        _type_set, _object_set, _attributes, _all_objs, _all_doms) :-
    object(_obj_id, _name, _type_id, _domain_set, _attributes, _all_objs, _all_doms),
    types_of('DOMAIN', _dom_types), on(_type_id, _dom_types),
    on(['Type_Set', _type_set], _attributes),
    on(['Object_Set', _object_set], _attributes).

plain_domain(_obj_id, _name, _type_id, _domain_set,
        _type_set, _object_set, _attributes, _all_objs, _all_doms) :-
    domain(_obj_id, _name, _type_id, _domain_set,
        _type_set, _object_set, _attributes, _all_objs, _all_doms),
    _type_id == 'DOMAIN'.
```

Domain Expression Evaluation.

See A.2.2

```
dom_expr_membs(n, []).

dom_expr_membs([ d | _dom_ids], _membs):-
    setof(_obj_id,
        _dom_id^_all_membs^_a^_b^_c^_d^_e^_f^_g^(on(_dom_id, _dom_ids),
        (domain(_dom_id, _a, _b, _c, _d, _e, _f, _all_membs, _g),
        on(_obj_id, _all_membs));
        _obj_id = _dom_id)),
        _membs).

dom_expr_membs([ r | _obj_ids], _membs):-
    setof(_obj_id1,
        _dom_id^_object_set^_a^_b^_c^_d^_e^_f^_g^(on(_dom_id, _obj_ids),
        domain(_dom_id, _a, _b, _c, _d, _object_set, _e, _f, _g),
        on(_obj_id1, _object_set)),
        _membs).

dom_expr_membs([o | _obj_ids], _obj_ids).

dom_expr_membs([_dom_expr1, u, _dom_expr2], _membs):-
    dom_expr_membs(_dom_expr1, _membs1),
    dom_expr_membs(_dom_expr2, _membs2),
    setof(_memb, (on(_memb, _membs1); on(_memb, _membs2)), _membs).

dom_expr_membs([_dom_expr1, df, _dom_expr2], _membs):-
    dom_expr_membs(_dom_expr1, _membs1),
    dom_expr_membs(_dom_expr2, _membs2),
    setof(_memb, (on(_memb, _membs1), not on(_memb, _membs2)), _membs).

dom_expr_membs([_dom_expr1, i, _dom_expr2], _membs):-
    dom_expr_membs(_dom_expr1, _membs1),
    dom_expr_membs(_dom_expr2, _membs2),
    setof(_memb, (on(_memb, _membs1), on(_memb, _membs2)), _membs).

dom_expr_memb(_dom_expr, _memb):-
    dom_expr_membs(_dom_expr, _membs),
    on(_memb, _membs).
```

Recalculation of Domain Membership

This is a set of utilitities used whenever a domain's membership is altered to recalculate all itsdirect and indirect members.

```
recalc_all_objs(_obj_id_list):-
    forall(on(_obj_id, _obj_id_list),
        (more_membs_of([], [], [_obj_id], _new_all_objs),
        retract(object(_obj_id, _name, _type_id, _domain_set,
            [['Type_Set', _type_set], ['Object_Set', _object_set]], _all_objs, _all_doms)),
        assert(object(_obj_id, _name, _type_id, _domain_set,
            [ ['Type_Set', _type_set], ['Object_Set', _object_set]],
                _new_all_objs, _all_doms)))).
```

more_membs_of builds a list in its first parameter. Its second parameter contains a list of domains which have been tried already and the third is the untried list. It terminates when the untried list is empty and the list is returned in the fourth parameter. candidate_domain tests each domains for possible addition to the untried list.

```
more_membs_of(_membs, _tried_already1, [], _all_membs):-
    _all_membs = _membs.

more_membs_of(_membs1, _tried_already1, [_obj_id | _doms_list1], _all_membs):-
    domain(_obj_id, _, _, _, _, _object_set, _, _, _),
    _tried_already2 = [_obj_id | _tried_already1],
    (setof(_memb, (on(_memb, _object_set); on(_memb, _membs1)), _membs2);
        _membs2 = []),
    (setof(_memb,
        candidate_domain(_memb, _doms_list1, _object_set, _tried_already2),
        _doms_list2);
    _doms_list2 = []),
    more_membs_of(_membs2, _tried_already2, _doms_list2, _all_membs).

candidate_domain(_dom_id, _doms_list, _object_set, _tried_already):-
    (on(_dom_id, _doms_list);
    (on(_dom_id, _object_set),
    domain(_dom_id, _, _, _, _, _, _, _, _))),
    not on(_dom_id, _tried_already).
```

B.2.3 Operations

Feasible Operations

See A.3.1

```
feasible_operation(_user_id, _target_id, _op_id):-
    object(_user_id, _, 'USER', _, _, _, _),!,
    object(_target_id, _,  _type_id, _, _, _, _),!,
    type_ops(_type_id, _op_ids),
    on(_op_id, _op_ids).
```

Object Creation

See A.3.3

```
object_types_create(_user_id, _target_id, _new_obj_id, _name, _type_id, _new_attrs):-
    permitted_operation(_user_id, _target_id, 'CREATE'), !,
    domain(_target_id, _, _, _, _type_set, _object_set, _, _, _all_doms),
    valid_type_id(_type_id, _type_set),
    (not object(_new_obj_id, _, _, _, _, _, _)),
    domain_add_to_object_set(_target_id, _object_set, [_new_obj_id]),
    assert(object(_new_obj_id, _name, _type_id, [_target_id], _new_attrs,
            [], [_target_id | _all_doms])).

object_create(_user_id, _target_id, _new_obj_id, _name, _type_id, []):-
    types_of('DOMAIN', _dom_types), not on(_type_id, _dom_types),
    not _type_id == 'ACCESS_RULE',
    object_types_create(_user_id, _target_id, _new_obj_id, _name, _type_id, [[]]).
```

Object Destruction

See A.3.3

```
object_types_destroy(_user_id, _target_id, _destr_obj_id):-
    permitted_operation(_user_id, _target_id, 'DESTROY'), !,
    domain(_target_id, _, _, _, _, _object_set, _, _, _),
    object(_destr_obj_id, _, _, _, _, _, _),
    domain_take_from_object_set(_target_id, _object_set, [_destr_obj_id]),
    retract(object(_destr_obj_id, _, _, _, _, _, _)).

object_destroy(_user_id, _target_id, _destr_obj_id):-
    object(_destr_obj_id, _, _type_id, _, _, _, _),
    types_of('DOMAIN', _dom_types), not on(_type_id, _dom_types),
    not _type_id == 'ACCESS_RULE',
    object_types_destroy(_user_id, _target_id, _destr_obj_id).
```

Domain Creation

See A.3.4

```
domain_types_create(_user_id, _target_id, _new_obj_id, _name, _type_id,
        _type_set, _attrs):-
    types_of('DOMAIN', _dom_types),
    on(_type_id, _dom_types),
    object_types_create(_user_id, _target_id, _new_obj_id, _name, _type_id,
        [['Type_Set', _type_set], ['Object_Set', []] | _attrs]).

object_create(_user_id, _target_id, _new_obj_id, _name, 'DOMAIN', _type_set):-
    domain_types_create(_user_id, _target_id, _new_obj_id, _name,
        'DOMAIN', _type_set, []).
```

Domain Destruction

See A.3.4

```
object_destroy(_user_id, _target_id, _destr_obj_id):-
    plain_domain(_destr_obj_id, _, _, _, _, _, _, _, _),
    object_types_destroy(_user_id, _target_id, _destr_obj_id).
```

Including and Removing Objects

See A.3.4

```
domain_include_objects(_user_id, _target_id, _member_ids):-
    permitted_operation(_user_id, _target_id, 'DOM_INCLUDE_OBJECT'), !,
    forall(on(_member_id, _member_ids),
        authorised_op_request(_user_id, _member_id, 'ALTER_DOMAIN_SET')), !,
    domain(_target_id, _, _, _, _type_set, _object_set, _, _, _),
    forall(on(_obj_id,_member_ids),
        (object(_obj_id, _, _type_id, _, _, _, _),
        valid_type_id(_type_id, _type_set))),
    forall(on(_obj_id,_member_ids),
        (object(_obj_id, _, _, _domain_set, _, _, _),
        object_add_to_domain_set(_obj_id, _domain_set, _target_id))),
    domain_add_to_object_set(_target_id, _object_set, _member_ids).
```

```
domain_remove_objects(_user_id, _target_id, _member_ids):-
    permitted_operation(_user_id, _target_id, 'DOM_REMOVE_OBJECT'), !,
    forall(on(_member_id, _member_ids),
        authorised_op_request(_user_id, _member_id, 'ALTER_DOMAIN_SET')), !,
    domain(_target_id, _, _, _, _type_set, _object_set, _, _, _),
    domain_take_from_object_set(_target_id, _object_set, _member_ids),
    forall(on(_obj_id, _member_ids),
        (object(_obj_id, _, _type_id, _domain_set, _, _, _) ->
        object_take_from_domain_set(_obj_id, _domain_set, _target_id))).
```

Read a Domain's Object Set

See A.3.4

```
domain_read_objects(_user_id, _target_id, _object_set):-
    permitted_operation(_user_id, _target_id, 'DOM_READ_OBJECTS'), !,
    domain(_target_id, _, _, _, _type_set, _object_set, _, _, _).
```

Manipulating Object Sets

These are common routines used by the domain operations to manipulate object sets. `domain_add_to_object_set` and `domain_take_from_object_set` both use `domain_alter_object_set` for their common function.

```
domain_alter_object_set(_target_id, _new_object_set):-
    object(_target_id, _, _, _, _, _all_objs, _),!,
    (_new_object_set == [] -> _new_all_objs = [];
        setof(_obj_id,
            (on(_obj_id, _all_objs); on(_obj_id, _new_object_set)),
            _new_all_objs)),!,
    retract(object(_target_id, _name, _type_id, _domain_set,
        [_type_set_attr, ['Object_Set', _object_set] | _attrs],
        _all_objs, _all_doms)),!,
    assert(object(_target_id, _name, _type_id, _domain_set,
        [_type_set_attr, ['Object_Set', _new_object_set] | _attrs],
        _new_all_objs, _all_doms)),!,
    recalc_all_objs(_all_doms).

domain_add_to_object_set(_target_id, _object_set, _obj_ids):-
    setof(_memb,
        (on(_memb, _object_set); on(_memb, _obj_ids)),
        _new_object_set),
    domain_alter_object_set(_target_id, _new_object_set).
```

```
domain_take_from_object_set(_target_id, _object_set, _obj_ids):-
    (setof(_memb,
        (on(_memb, _object_set), not on(_memb, _obj_ids)),
        _new_object_set);
    _new_object_set = []),
    domain_alter_object_set(_target_id, _new_object_set).
```

Manipulating Domain Sets

These are common routines used by the domain operations to manipulate domain sets. object_add_to_domain_set and object_take_from_domain_set both use object_alter_domain_set for their common function.

```
object_alter_domain_set(_target_id, _new_domain_set):-
    setof(_obj_id,
        (on(_obj_id2, _new_domain_set),
        object(_obj_id2, _, _, _, _,_, _all_doms2),
            on(_obj_id, _all_doms2)),
        _all_doms3),!,
    setof(_obj_id3,
        (on(_obj_id3, _all_doms3);
         on(_obj_id3, _new_domain_set)),
        _new_all_doms),!,
    retract(object(_target_id, _name, _type_id, _domain_set, _attrs,
        _all_objs, _all_doms)),!,
     assert(object(_target_id, _name, _type_id, _new_domain_set, _attrs,
        _all_objs, _new_all_doms)).

object_add_to_domain_set(_target_id, _domain_set, _obj_id):-
    setof(_memb,
        (on(_memb, _domain_set); _memb = _obj_id),
        _new_domain_set),
    object_alter_domain_set(_target_id, _new_domain_set).

object_take_from_domain_set(_target_id, _domain_set, _obj_id):-
    (setof(_memb,
        (on(_memb, _domain_set), not _memb = _obj_id),
        _new_domain_set);
    _new_domain_set = []),
    object_alter_domain_set(_target_id, _new_domain_set).
```

Utilities

This utility validates an object's type_id for membership of a domain. It is valid if the domain's type_set contains ALL or the object's type_id, or if the type_id is DOMAIN.

```
valid_type_id(_type_id, _type_set):-
    on('ALL', _type_set).
valid_type_id(_type_id, _type_set):-
    on(_type_id, _type_set).
valid_type_id(_type_id, _type_set):-
    types_of('DOMAIN', _dom_types),
    on(_type_id, _dom_types).
```

This utility validates an op_id against the valid operations for a type. It is also valid if it equals ALL.

```
valid_op_id('ALL', _type_id).
valid_op_id(_op_id, _type_id):-
    type_ops(_type_id, _op_ids), on(_op_id, _op_ids).
```

The program does not attempt to control the order in which objects are placed in the database. The sort routine works its way down the (numeric) object identities, inserting each one in turn at the beginning od the database. It was written for simplicity, not speed.

```
sort_objects(0):-
    retract(object), asserta(object).

sort_objects(_x):-
    (object(_x, _, _, _, _, _, _) ->
        (retract(object(_x, _a, _b, _c, _d, _e, _f)),
        asserta(object(_x, _a, _b, _c, _d, _e, _f)));
        true),
    _y is (_x - 1),
    sort_objects(_y).
```

B.2.4 Access Rules

Access_Rule Objects

See A.4.1

```
access_rule(_obj_id, _name, _type_id, _domain_set,
        _user_domain, _target_domain, _operation_set, _attributes, _all_objs, _all_doms) :-
    object(_obj_id, _name, _type_id, _domain_set, _attributes, _all_objs, _all_doms),
    _type_id == 'ACCESS_RULE',
    on(['User_Domain', _user_domain], _attributes),
    on(['Target_Domain', _target_domain], _attributes),
    on(['Operation_Set', _operation_set], _attributes).
```

Authorised Operation

See A.4.2

```
authorised_op_request(_user_id, _target_id, _op_id):-
    access_rule(_, _, _, _, _user_domain, _target_domain, _operation_set,
        _, _, _),
    dom_expr_memb(_user_domain, _user_id),
    dom_expr_memb(_target_domain, _target_id),
    (on(_op_id, _operation_set); on('ALL', _operation_set)).

permitted_operation(_user_id, _target_id, _op_id):-
    feasible_operation(_user_id, _target_id, _op_id),!,
    authorised_op_request(_user_id, _target_id, _op_id),!.
```

Access Rule Creation

See A.4.3

```
object_create(_user_id, _target_id, _new_obj_id, _name, 'ACCESS_RULE',
        [_user_domain, _target_domain, _operation_set]):-
    sa_auth(_user_id, _user_domain, _target_domain),
    object_types_create(_user_id, _target_id, _new_obj_id, _name,
        'ACCESS_RULE',
        [['User_Domain', _user_domain],
        ['Target_Domain', _target_domain],
        ['Operation_Set', _operation_set]]).
```

Access Rule Destruction

See A.4.3

```
object_destroy(_user_id, _target_id, _destr_obj_id):-
    access_rule(_destr_obj_id, _, _, _, _user_domain, _target_domain, _, _, _, _),
    sa_auth(_user_id, _user_domain, _target_domain),
    object_types_destroy(_user_id, _target_id, _destr_obj_id).
```

B.2.5 Role Domain Objects

Role Domains

See A.5.1

```
role_domain(_obj_id, _name, _type_id, _domain_set, _type_set, _object_set,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope,
        _all_objs, _all_doms) :-
    domain(_obj_id, _name, _type_id, _domain_set,
        _type_set, _object_set, _attributes, _all_objs, _all_doms),
    _type_id == 'ROLE_DOMAIN',
    on(['Owner_Scope', _owner_scope], _attributes),
    on(['Manager_Scope', _manager_scope], _attributes),
    on(['SA_User_Scope', _sa_user_scope], _attributes),
    on(['SA_Target_Scope', _sa_target_scope], _attributes).
```

Role Domain Creation

See A.5.2

```
object_create(_user_id, _target_id, _new_obj_id, _name, 'ROLE_DOMAIN', _type_set):-
    domain_types_create(_user_id, _target_id, _new_obj_id, _name,
        'ROLE_DOMAIN', _type_set,
        [['Owner_Scope', n], ['Manager_Scope', n],
        ['SA_User_Scope', n], ['SA_Target_Scope', n]]).
```

Role Domain Destruction

See A.5.2

```
object_destroy(_user_id, _target_id, _destr_obj_id):-
    role_domain(_destr_obj_id, _, _, _, _, _,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope, _, _),
    _owner_scope == n,
    _manager_scope == n,
    _sa_user_scope == n,
    _sa_target_scope == n,
    object_types_destroy(_user_id, _target_id, _destr_obj_id).
```

Alter Role Domain Scope Attributes

See A.5.2

```
rdom_alter_owner_scope(_user_id, _target_id, _new_owner_scope):-
    permitted_operation(_user_id, _target_id, 'RDOM_ALTER'), !,
    role_domain(_target_id, _name, _type_id, _domain_set, _type_set, _object_set,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope,
        _all_objs, _all_doms),!,
    owner_auth(_user_id, _owner_scope),!,
    owner_auth(_user_id, _new_owner_scope),!,
    retract(object(_target_id, _, _, _, _, _, _)),
    assert(object(_target_id, _name, _type_id, _domain_set,
        ['Type_Set', _type_set], ['Object_Set', _object_set],
        ['Owner_Scope', _new_owner_scope], ['Manager_Scope', _manager_scope],
        ['SA_User_Scope', _sa_user_scope],
        ['SA_Target_Scope', _sa_target_scope]], _all_objs, _all_doms)).

rdom_alter_manager_scope(_user_id, _target_id, _new_manager_scope):-
    permitted_operation(_user_id, _target_id, 'RDOM_ALTER'), !,
    role_domain(_target_id, _name, _type_id, _domain_set, _type_set, _object_set,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope,
        _all_objs, _all_doms),!,
    owner_auth(_user_id, _manager_scope),!,
    owner_auth(_user_id, _new_manager_scope),!,
    retract(object(_target_id, _, _, _, _, _, _)),
    assert(object(_target_id, _name, _type_id, _domain_set,
        ['Type_Set', _type_set], ['Object_Set', _object_set],
        ['Owner_Scope', _owner_scope], ['Manager_Scope', _new_manager_scope],
        ['SA_User_Scope', _sa_user_scope],
        ['SA_Target_Scope', _sa_target_scope]], _all_objs, _all_doms)).
```

```
rdom_alter_sa_user_scope(_user_id, _target_id, _new_sa_user_scope):-
    permitted_operation(_user_id, _target_id, 'RDOM_ALTER'), !,
    role_domain(_target_id, _name, _type_id, _domain_set, _type_set, _object_set,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope,
        _all_objs, _all_doms),!,
    manager_auth(_user_id, _sa_user_scope),!,
    manager_auth(_user_id, _new_sa_user_scope),!,
    retract(object(_target_id, _, _, _, _, _, _)),
    assert(object(_target_id, _name, _type_id, _domain_set,
        [['Type_Set', _type_set], ['Object_Set', _object_set],
        ['Owner_Scope', _owner_scope], ['Manager_Scope', _manager_scope],
        ['SA_User_Scope', _new_sa_user_scope],
        ['SA_Target_Scope', _sa_target_scope]], _all_objs, _all_doms)).

rdom_alter_sa_target_scope(_user_id, _target_id, _new_sa_target_scope):-
    permitted_operation(_user_id, _target_id, 'RDOM_ALTER'), !,
    role_domain(_target_id, _name, _type_id, _domain_set, _type_set, _object_set,
        _owner_scope, _manager_scope, _sa_user_scope, _sa_target_scope,
        _all_objs, _all_doms),!,
    manager_auth(_user_id, _sa_target_scope),!,
    manager_auth(_user_id, _new_sa_target_scope),!,
    retract(object(_target_id, _, _, _, _, _, _)),
    assert(object(_target_id, _name, _type_id, _domain_set,
        [['Type_Set', _type_set], ['Object_Set', _object_set],
        ['Owner_Scope', _owner_scope], ['Manager_Scope', _manager_scope],
        ['SA_User_Scope', _sa_user_scope],
        ['SA_Target_Scope', _new_sa_target_scope]], _all_objs, _all_doms)).
```

Security Administator Authority

See A.4.3

```
sa_auth(_user_id, _user_domain, _target_domain):-
    dom_expr_membs(_user_domain, _user_dom_membs),!,
    dom_expr_membs(_target_domain, _target_dom_membs),!,
    role_domain(_rdom_id, _, _, _, _, _object_set, _, _,
        _sa_user_scope, _sa_target_scope, _all_objs, _),
    on(_user_id, _all_objs),
    dom_expr_membs(_sa_user_scope, _sa_user_scope_membs),
    forall(on(_memb, _user_dom_membs), on(_memb, _sa_user_scope_membs)),
    dom_expr_membs(_sa_target_scope, _sa_target_scope_membs),
    forall(on(_memb, _target_dom_membs), on(_memb, _sa_target_scope_membs)).
```

Owner & Manager Authority

See A.5.1

```
owner_auth(_user_id, _owner_domain):-
    dom_expr_membs(_owner_domain, _owner_dom_membs),!,
    role_domain(_rdom_id, _, _, _, _, _object_set, _owner_scope, _, _, _, _all_objs, _),
    on(_user_id, _all_objs),
    dom_expr_membs(_owner_scope, _owner_scope_membs),
    forall(on(_memb, _owner_dom_membs), on(_memb, _owner_scope_membs)).

manager_auth(_user_id, _manager_domain):-
    dom_expr_membs(_manager_domain, _manager_dom_membs),!,
    role_domain(_rdom_id, _, _, _, _, _object_set, _, _manager_scope, _, _, _all_objs, _),
    on(_user_id, _all_objs),
    dom_expr_membs(_manager_scope, _manager_scope_membs),
    forall(on(_memb, _manager_dom_membs), on(_memb, _manager_scope_membs)).
```

B.3 Standard Queries

See the discussion in section VI.1.

```
disowned_child(_obj_id, _dom_id) :-
    object(_obj_id, _name, _type_id, _domain_set, _, _, _),
    on(_dom_id, _domain_set),
    domain(_dom_id, _, _, _, _, _object_set, _, _, _),
    not on(_obj_id, _object_set)

orphan(_obj_id, _dom_id) :-
    object(_obj_id, _name, _type_id, _domain_set, _, _, _),
    (_domain_set = [];
    (on(_dom_id, _domain_set),
    not domain(_dom_id, _, _, _, _, _, _, _, _))).

bereaved_parent(_dom_id, _obj_id) :-
    domain(_dom_id, _, _, _, _, _object_set, _, _, _),
    on(_obj_id, _object_set),
    not object(_obj_id, _, _, _, _, _, _).

lost_parent(_dom_id, _obj_id) :-
    domain(_dom_id, _, _, _, _, _object_set, _, _, _),
    on(_obj_id, _object_set),
    object(_obj_id, _, _, _domain_set, _, _, _),
    not on(_dom_id, _domain_set).

access_rule_effect(_ar_id, _user_ids, _target_ids, _operation_set):-
```

```
    access_rule(_ar_id, _, _, _, _user_domain, _target_domain, _operation_set, _, _, _),
    dom_expr_membs(_user_domain, _obj_ids),
    one find_user_ids(_obj_ids, _user_ids),
    dom_expr_membs(_target_domain, _target_ids).

find_user_ids(_obj_ids, _user_ids):-
    setof(_user_id,
        _a^_b^_c^_d^_e^(on(_user_id, _obj_ids),
            object(_user_id,  _a, 'USER', _b, _c, _d, _e)),
        _user_ids).

find_user_ids(_obj_ids, _user_ids):-
        _user_ids = [].

user_can_access(_user_id, _list):-
    one (setof(_item, user_in_access_rule(_user_id, _item), _list); _list = []).

user_in_access_rule(_user_id, [_ar_id, _target_ids, _operation_set]):-
    access_rule_effect(_ar_id, _user_ids, _target_ids, _operation_set),
    on(_user_id, _user_ids).

object_can_be_accessed(_obj_id, _list):-
    object(_obj_id, _, _type_id, _domain_set, _, _, _),
    one (setof(_item, object_in_access_rule(_obj_id, _type_id, _item), _list); _list = []).

object_in_access_rule(_target_id, _type_id, [_ar_id, _user_ids, _op_ids]):-
    access_rule_effect(_ar_id, _user_ids, _target_ids, _operation_set),
    setof(_op_id,  (on(_op_id, _operation_set), valid_op_id(_op_id, _type_id)), _op_ids),
    on(_target_id, _target_ids).

user_has_auth(_user_id, _list):-
    one (setof(_item, user_has_scopes(_user_id, _item), _list); _list = []).

user_has_scopes(_user_id, [_rdom_id, _ow, _ma, _sa_u, _sa_t]):-
    role_domain(_rdom_id, _, _, _, _, _,
        _ow_ex, _ma_ex, _sa_u_ex, _sa_t_ex, _all_objs, _),
    on(_user_id, _all_objs),
    dom_expr_membs(_ow_ex, _ow),
    dom_expr_membs(_ma_ex, _ma),
    dom_expr_membs(_sa_u_ex, _sa_u),
    dom_expr_membs(_sa_t_ex, _sa_t).
```

B.4 Example

These objects correspond to the objects in the example in section V.1

B.4.1 Building the Example

Base Set of Objects on Which to Start

```
object.
object(2, 'ROOT_DOM',  'DOMAIN', [],  [['Type_Set', ['ALL']], ['Object_Set', [3,4]]], [], []).
object(3, 'OWNER_DOM', 'ROLE_DOMAIN', [2], [['Type_Set', ['USER']], ['Object_Set', [5]],
    ['Owner_Scope', [d, 2]], ['Manager_Scope', [d, 2]],
    ['SA_User_Scope', [d, 2]], ['SA_Target_Scope', [d, 2]]], [], []).
object(4,  'OWNER_AR', 'ACCESS_RULE', [2],
    [['User_Domain', [d, 3]], ['Target_Domain', [d, 2]], ['Operation_Set', ['ALL']]], [], []).
object(5, 'THE_OWNER', 'USER', [4], [[]], [], []).
```

Operations to Build up the Example

These were the operations used to build up the example.

```
start_from_scratch:-
    object_create(5, 2, 10, 'AR_DOM', 'DOMAIN', ['ACCESS_RULE']),!,
    domain_include_objects(5, 10, [4]),!,
    domain_remove_objects(5, 2, [4]),!,
    object_create(5, 2, 11, 'RESOURCES_DOM', 'DOMAIN', ['FILE', 'TARGET2']),!,
    object_create(5, 2, 12, 'USERS_DOM', 'DOMAIN', ['USER']),!,
    object_create(5, 12, 13, 'ABC_USERS', 'DOMAIN', ['USER']),!,
    object_create(5, 12, 14, 'DEF_USERS', 'DOMAIN', ['USER']),!,
    object_create(5, 13, 15, 'MAN_DIR', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 13, 16, 'ADMIN_DIR', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 13, 17, 'FINANCE_DIR', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 13, 18, 'RESEARCH_DIR', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 13, 19, 'ADMIN_DEPT', 'DOMAIN', ['USER']),!,
    object_create(5, 13, 20, 'FINANCE_DEPT', 'DOMAIN', ['USER']),!,
    object_create(5, 13, 21, 'RESEARCH_DEPT', 'DOMAIN', ['USER']),!,
    object_create(5, 19, 22, 'ABC_SEC_ADMIN', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 19, 23, 'PERSONNEL', 'DOMAIN', ['USER']),!,
    object_create(5, 21, 24, 'RESEARCH_GEN', 'DOMAIN', ['USER']),!,
    object_create(5, 21, 25, 'RESEARCH_ABCDEF', 'DOMAIN', ['USER']),!,
    object_create(5, 25, 26, 'ABCDEF_PM', 'DOMAIN', ['USER']),!,
    object_create(5, 25, 27, 'ABCDEF_SC', 'DOMAIN', ['USER']),!,
    object_create(5, 25, 28, 'ABCDEF_PS', 'DOMAIN', ['USER']),!,
    object_create(5, 14, 29, 'DEF_SEC_ADMIN', 'ROLE_DOMAIN', ['USER']),!,
    object_create(5, 14, 30, 'DEFABC_JV', 'DOMAIN', ['USER']).

start_from_scratch2:-
    object_create(5, 15, 100, 'USER_A', 'USER', []),!,
    object_create(5, 16, 101, 'USER_B', 'USER', []),!,
    object_create(5, 17, 102, 'USER_C', 'USER', []),!,
    object_create(5, 18, 103, 'USER_D', 'USER', []),!,
    object_create(5, 22, 104, 'USER_E', 'USER', []),!,
    object_create(5, 26, 105, 'USER_F', 'USER', []),!,
    object_create(5, 27, 106, 'USER_G', 'USER', []),!,
    object_create(5, 27, 107, 'USER_H', 'USER', []),!,
    object_create(5, 27, 108, 'USER_I', 'USER', []),!,
    object_create(5, 28, 109, 'USER_J', 'USER', []),!,
```

```
    object_create(5, 29, 110, 'USER_K', 'USER', []),!,
    object_create(5, 30, 111, 'USER_L', 'USER', []),!,
    object_create(5, 30, 112, 'USER_M', 'USER', []).

start_from_scratch3:-
    object_create(5, 11, 50, 'FILES_DOM', 'DOMAIN', ['FILE']),!,
    object_create(5, 50, 51, 'ADMIN_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 50, 52, 'FINANCE_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 50, 53, 'RESEARCH_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 51, 54, 'DPA_DOM', 'DOMAIN', ['FILE']),!,
    object_create(5, 51, 55, 'PERSONNEL_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 52, 56, 'SUPPLIERS_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 53, 57, 'ABCDEF_PROJ_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 57, 58, 'ABCDEF_PRIV_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 57, 59, 'ABCDEF_SHRD_FILES', 'DOMAIN', ['FILE']),!,
    object_create(5, 53, 60, 'RES_FILES_X', 'DOMAIN', ['FILE']),!,
    object_create(5, 53, 61, 'RES_FILES_Y', 'DOMAIN', ['FILE']).

 start_from_scratch4:-
    rdom_alter_manager_scope(5, 15, [d, 11,12]),!,
    rdom_alter_manager_scope(5, 16, [d, 14, 19, 51]),!,
    rdom_alter_manager_scope(5, 17, [d, 20, 52]),!,
    rdom_alter_manager_scope(5, 18, [d, 21, 53]),!,
    object_create(5, 10, 71, 'AR1', 'ACCESS_RULE', [[d, 15], [d, 11, 12],
    ['RDOM_ALTER']]),!,
    object_create(5, 10, 72, 'AR2', 'ACCESS_RULE',
    [[d, 16], [d, 14, 19, 51], ['RDOM_ALTER']]),!,
    object_create(5, 10, 73, 'AR3', 'ACCESS_RULE',
    [[d, 17], [d, 20, 52], ['RDOM_ALTER']]),!,
    object_create(5, 10, 74, 'AR4', 'ACCESS_RULE',
    [[d, 18], [d, 21, 29], ['RDOM_ALTER']]).

start_from_scratch5:-
    rdom_alter_sa_user_scope(5, 15, [d, 12]),!,
    rdom_alter_sa_target_scope(5, 15, [d, 10, 11, 12]),!,
    rdom_alter_sa_user_scope(100, 22, [[d, 12], df, [d, 22]]),!,
    rdom_alter_sa_target_scope(100, 22, [d, 50]),!,
    rdom_alter_sa_user_scope(5, 16, [d, 14, 19]),!,
    rdom_alter_sa_target_scope(5, 16, [d, 10,14, 51]),!,
    rdom_alter_sa_user_scope(101, 29, [[d, 14], df, [d, 29]]),!,
    rdom_alter_sa_target_scope(103, 29,  [d, 59]),!,
    object_create(5, 10, 75, 'AR5', 'ACCESS_RULE', [[d, 15], [d, 10], ['ALL']]),!,
    object_create(5, 10, 76, 'AR6', 'ACCESS_RULE', [[d, 16], [d, 10], ['ALL']]),!,
    object_create(100, 10, 77, 'AR7', 'ACCESS_RULE', [[d, 22], [d, 10], ['ALL']]),!,
    object_create(101, 10, 78, 'AR8', 'ACCESS_RULE', [[d, 29], [d, 10], ['ALL']]),!,
    object_create(100, 10, 79, 'AR9', 'ACCESS_RULE', [[d, 22], [d, 12], ['ALL']]),!,
    object_create(101, 10, 80, 'AR10', 'ACCESS_RULE', [[d, 29], [d, 14], ['ALL']]).

start_from_scratch6:-
    object_create(5, 51, 201, 'AF1', 'FILE', []),!,
    object_create(5, 55, 211, 'PF1', 'FILE', []),!,
    object_create(5, 52, 221, 'FF1', 'FILE', []),!,
    object_create(5, 56, 231, 'SF1', 'FILE', []),!,
    object_create(5, 58, 251, 'APF1', 'FILE', []),!,
    object_create(5, 59, 261, 'ASF1', 'FILE', []),!,
    object_create(5, 60, 271, 'RXF1', 'FILE', []),!,
    object_create(5, 61, 281, 'RYF1', 'FILE', []),!,
    object_create(5, 51, 202, 'AF2', 'FILE', []),!,
```

```
    object_create(5, 55, 212, 'PF2', 'FILE', []),!,
    object_create(5, 52, 222, 'FF2', 'FILE', []),!,
    object_create(5, 56, 232, 'SF2', 'FILE', []),!,
    object_create(5, 58, 252, 'APF2', 'FILE', []),!,
    object_create(5, 59, 262, 'ASF2', 'FILE', []),!,
    object_create(5, 60, 272, 'RXF2', 'FILE', []),!,
    object_create(5, 61, 282, 'RYF2', 'FILE', []).

start_from_scratch7:-
    object_create(104, 10, 90, 'AR20', 'ACCESS_RULE',
        [[[d, 19], df, [d, 22]], [[d, 51], df, [d, 55]], ['ALL']]),!,
    object_create(104, 10, 91, 'AR21', 'ACCESS_RULE',
        [[d, 23], [d, 55], ['ALL']]),!,
    object_create(104, 10, 92, 'AR22', 'ACCESS_RULE',
        [[d, 20], [d, 52], ['ALL']]),!,
    object_create(104, 10, 93, 'AR23', 'ACCESS_RULE',
        [[d, 25], [d, 57], ['ALL']]),!,
    object_create(104, 10, 94, 'AR24', 'ACCESS_RULE',
        [[d, 21], [d, 60, 61], ['ALL']]),!,
    object_create(110, 10, 95, 'AR25', 'ACCESS_RULE',
        [[d, 30], [d, 59], ['ALL']]).
```

B.4.2 Final Result

This is the database set of objects described in the example, section V.1.

```
object.
object(2, 'ROOT_DOM', 'DOMAIN', [], [['Type_Set', ['ALL']], ['Object_Set', [3, 10, 11, 12]]], [], []).
object(3, 'OWNER_DOM', 'ROLE_DOMAIN', [2], [['Type_Set', ['USER']], ['Object_Set', [5]],
['Owner_Scope', [d, 2]], ['Manager_Scope', [d, 2]], ['SA_User_Scope', [d, 2]], ['SA_Target_Scope', [d,
2]]], [], []).
object(4, 'OWNER_AR', 'ACCESS_RULE', [10], [['User_Domain', [d, 3]], ['Target_Domain', [d, 2]],
['Operation_Set', ['ALL']]], [], []).
object(5, 'THE_OWNER', 'USER', [3], [[]], [], []).
object(10, 'AR_DOM', 'DOMAIN', [2], [['Type_Set', ['ACCESS_RULE']], ['Object_Set', [4, 71, 72, 73,
74, 75, 76, 77, 78, 79, 80, 90, 91, 92, 93, 94, 95]]], [], []).
object(11, 'RESOURCES_DOM', 'DOMAIN', [2], [['Type_Set', ['FILE', 'TARGET2']], ['Object_Set',
[50]]], [], []).
object(12, 'USERS_DOM', 'DOMAIN', [2], [['Type_Set', ['USER']], ['Object_Set', [13, 14]]], [], []).
object(13, 'ABC_USERS', 'DOMAIN', [12], [['Type_Set', ['USER']], ['Object_Set', [15, 16, 17, 18, 19,
20, 21]]], [], []).
object(14, 'DEF_USERS', 'DOMAIN', [12], [['Type_Set', ['USER']], ['Object_Set', [29, 30]]], [], []).
object(15, 'MAN_DIR', 'ROLE_DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [100]],
['Owner_Scope', n], ['Manager_Scope', [d, 11, 12]], ['SA_User_Scope', [d, 12]], ['SA_Target_Scope',
[d, 10, 11, 12]]], [], []).
object(16, 'ADMIN_DIR', 'ROLE_DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [101]],
['Owner_Scope', n], ['Manager_Scope', [d, 14, 19, 51]], ['SA_User_Scope', [d, 14, 19]],
['SA_Target_Scope', [d, 10, 14, 51, d]]], [], []).
object(17, 'FINANCE_DIR', 'ROLE_DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [102]],
['Owner_Scope', n], ['Manager_Scope', [d, 20, 52]], ['SA_User_Scope', n], ['SA_Target_Scope', n]], [],
[1]).
object(18, 'RESEARCH_DIR', 'ROLE_DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [103]],
['Owner_Scope', n], ['Manager_Scope', [d, 21, 53, d]], ['SA_User_Scope', n], ['SA_Target_Scope', n]],
[], []).
object(19, 'ADMIN_DEPT', 'DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [22, 23]]], [], []).
object(20, 'FINANCE_DEPT', 'DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', []]], [], []).
object(21, 'RESEARCH_DEPT', 'DOMAIN', [13], [['Type_Set', ['USER']], ['Object_Set', [24, 25]]], [], []).
```

```
object(22, 'ABC_SEC_ADMIN', 'ROLE_DOMAIN', [19], [['Type_Set', ['USER']], ['Object_Set', [104]],
['Owner_Scope', n], ['Manager_Scope', n], ['SA_User_Scope', [[d, 12], df, [d, 22]]],
['SA_Target_Scope', [d, 50]]], [], []).
object(23, 'PERSONNEL', 'DOMAIN', [19], [['Type_Set', ['USER']], ['Object_Set', []]], [], []).
object(24, 'RESEARCH_GEN', 'DOMAIN', [21], [['Type_Set', ['USER']], ['Object_Set', []]], [], []).
object(25, 'RESEARCH_ABCDEF', 'DOMAIN', [21], [['Type_Set', ['USER']], ['Object_Set', [26, 27, 28]]],
[], []).
object(26, 'ABCDEF_PM', 'DOMAIN', [25], [['Type_Set', ['USER']], ['Object_Set', [105]]], [], []).
object(27, 'ABCDEF_SC', 'DOMAIN', [25], [['Type_Set', ['USER']], ['Object_Set', [106, 107, 108]]], [],
[]).
object(28, 'ABCDEF_PS', 'DOMAIN', [25], [['Type_Set', ['USER']], ['Object_Set', [109]]], [], []).
object(29, 'DEF_SEC_ADMIN', 'ROLE_DOMAIN', [14], [['Type_Set', ['USER']], ['Object_Set', [110]],
['Owner_Scope', n], ['Manager_Scope', n], ['SA_User_Scope', [[d, 14], df, [d, 29]]],
['SA_Target_Scope', [d, 59]]], [], []).
object(30, 'DEFABC_JV', 'DOMAIN', [14], [['Type_Set', ['USER']], ['Object_Set', [111, 112]]], [], []).
object(50, 'FILES_DOM', 'DOMAIN', [11], [['Type_Set', ['FILE']], ['Object_Set', [51, 52, 53]]], [], []).
object(51, 'ADMIN_FILES', 'DOMAIN', [50], [['Type_Set', ['FILE']], ['Object_Set', [54, 55, 201, 202]]], [],
[]).
object(52, 'FINANCE_FILES', 'DOMAIN', [50], [['Type_Set', ['FILE']], ['Object_Set', [56, 221, 222]]], [],
[]).
object(53, 'RESEARCH_FILES', 'DOMAIN', [50], [['Type_Set', ['FILE']], ['Object_Set', [57, 60, 61]]], [],
[]).
object(54, 'DPA_DOM', 'DOMAIN', [51], [['Type_Set', ['FILE']], ['Object_Set', [55, 56]]], [], []).
object(55, 'PERSONNEL_FILES', 'DOMAIN', [51, 54], [['Type_Set', ['FILE']], ['Object_Set', [211, 212]]],
[], []).
object(56, 'SUPPLIERS_FILES', 'DOMAIN', [52, 54], [['Type_Set', ['FILE']], ['Object_Set', [231, 232]]],
[], []).
object(57, 'ABCDEF_PROJ_FILES', 'DOMAIN', [53], [['Type_Set', ['FILE']], ['Object_Set', [58, 59]]], [],
[]).
object(58, 'ABCDEF_PRIV_FILES', 'DOMAIN', [57], [['Type_Set', ['FILE']], ['Object_Set', [251, 252]]], [],
[]).
object(59, 'ABCDEF_SHRD_FILES', 'DOMAIN', [57], [['Type_Set', ['FILE']], ['Object_Set', [261, 262]]],
[], []).
object(60, 'RES_FILES_X', 'DOMAIN', [53], [['Type_Set', ['FILE']], ['Object_Set', [271, 272]]], [], []).
object(61, 'RES_FILES_Y', 'DOMAIN', [53], [['Type_Set', ['FILE']], ['Object_Set', [281, 282]]], [], []).
object(71, 'AR1', 'ACCESS_RULE', [10], [['User_Domain', [d, 15]], ['Target_Domain', [d, 11, 12]],
['Operation_Set', ['RDOM_ALTER']]], [], [1]).
object(72, 'AR2', 'ACCESS_RULE', [10], [['User_Domain', [d, 16]], ['Target_Domain', [d, 14, 19, 51]],
['Operation_Set', ['RDOM_ALTER']]], [], []).
object(73, 'AR3', 'ACCESS_RULE', [10], [['User_Domain', [d, 17]], ['Target_Domain', [d, 20, 52]],
['Operation_Set', ['RDOM_ALTER']]], [], []).
object(74, 'AR4', 'ACCESS_RULE', [10], [['User_Domain', [d, 18]], ['Target_Domain', [d, 21, 29]],
['Operation_Set', ['RDOM_ALTER']]], [], []).
object(75, 'AR5', 'ACCESS_RULE', [10], [['User_Domain', [d, 15]], ['Target_Domain', [d, 10]],
['Operation_Set', ['ALL']]], [], []).
object(76, 'AR6', 'ACCESS_RULE', [10], [['User_Domain', [d, 16]], ['Target_Domain', [d, 10]],
['Operation_Set', ['ALL']]], [], []).
object(77, 'AR7', 'ACCESS_RULE', [10], [['User_Domain', [d, 22]], ['Target_Domain', [d, 10]],
['Operation_Set', ['ALL']]], [], []).
object(78, 'AR8', 'ACCESS_RULE', [10], [['User_Domain', [d, 29]], ['Target_Domain', [d, 10]],
['Operation_Set', ['ALL']]], [], []).
object(79, 'AR9', 'ACCESS_RULE', [10], [['User_Domain', [d, 22]], ['Target_Domain', [d, 12]],
['Operation_Set', ['ALL']]], [], []).
object(80, 'AR10', 'ACCESS_RULE', [10], [['User_Domain', [d, 29]], ['Target_Domain', [d, 14]],
['Operation_Set', ['ALL']]], [], []).
object(90, 'AR20', 'ACCESS_RULE', [10], [['User_Domain', [[d, 19], df, [d, 22]]], ['Target_Domain', [[d,
51], df, [d, 55, d]]], ['Operation_Set', ['ALL']]], [], []).
```

```
object(91, 'AR21', 'ACCESS_RULE', [10], [['User_Domain', [d, 23]], ['Target_Domain', [d, 55]],
['Operation_Set', ['ALL']]], [], [1]).
object(92, 'AR22', 'ACCESS_RULE', [10], [['User_Domain', [d, 20]], ['Target_Domain', [d, 52]],
['Operation_Set', ['ALL']]], [], []).
object(93, 'AR23', 'ACCESS_RULE', [10], [['User_Domain', [d, 25]], ['Target_Domain', [d, 57]],
['Operation_Set', ['ALL']]], [], []).
object(94, 'AR24', 'ACCESS_RULE', [10], [['User_Domain', [d, 21]], ['Target_Domain', [d, 60, 61]],
['Operation_Set', ['ALL']]], [], []).
object(95, 'AR25', 'ACCESS_RULE', [10], [['User_Domain', [d, 30]], ['Target_Domain', [59]],
['Operation_Set', ['ALL']]], [], []).
object(100, 'USER_A', 'USER', [15], [[]], [], []).
object(101, 'USER_B', 'USER', [16], [[]], [], []).
object(102, 'USER_C', 'USER', [17], [[]], [], []).
object(103, 'USER_D', 'USER', [18], [[]], [], []).
object(104, 'USER_E', 'USER', [22], [[]], [], []).
object(105, 'USER_F', 'USER', [26], [[]], [], []).
object(106, 'USER_G', 'USER', [27], [[]], [], []).
object(107, 'USER_H', 'USER', [27], [[]], [], []).
object(108, 'USER_I', 'USER', [27], [[]], [], []).
object(109, 'USER_J', 'USER', [28], [[]], [], []).
object(110, 'USER_K', 'USER', [29], [[]], [], []).
object(111, 'USER_L', 'USER', [30], [[]], [], []).
object(112, 'USER_M', 'USER', [30], [[]], [], []).
object(201, 'AF1', 'FILE', [51], [[]], [], []).
object(202, 'AF2', 'FILE', [51], [[]], [], []).
object(211, 'PF1', 'FILE', [55], [[]], [], []).
object(212, 'PF2', 'FILE', [55], [[]], [], []).
object(221, 'FF1', 'FILE', [52], [[]], [], []).
object(222, 'FF2', 'FILE', [52], [[]], [], []).
object(231, 'SF1', 'FILE', [56], [[]], [], []).
object(232, 'SF2', 'FILE', [56], [[]], [], []).
object(251, 'APF1', 'FILE', [58], [[]], [], []).
object(252, 'APF2', 'FILE', [58], [[]], [], []).
object(261, 'ASF1', 'FILE', [59], [[]], [], []).
object(262, 'ASF2', 'FILE', [59], [[]], [], []).
object(271, 'RXF1', 'FILE', [60], [[]], [], []).
object(272, 'RXF2', 'FILE', [60], [[]], [], []).
object(281, 'RYF1', 'FILE', [61], [[]], [], []).
object(282, 'RYF2', 'FILE', [61], [[]], [], []
```