Design and specification of role based access control policies

M.Hitchens and V.Varadharajan

Abstract: The authors describe a language based approach to the specification of authorisation policies that can be used to support the range of access control policies in commercial object systems. They discuss the issues involved in the design of a language for role based access control systems. The notion of roles is used as a primitive construct within the language. The basic constructs of the language are discussed and the language is used to specify several access control policies such as role based access control; static and dynamic separation of duty delegation and joint action based access policies. The language is flexible and is able to capture meta-level operations, and it is often these features which are significant when it comes to the applicability of an access control system to practical real situations.

1 Introduction

In a computing system, when a request for a certain service is received by one principal (an agent) from another, the receiving principal needs to address two questions. First, is the requesting principal the one it claims to be? Secondly, does the requesting principal have the appropriate privileges for the requested service? These two questions relate to the issues of authentication and access control (authorisation). Traditionally the work on access control classifies security models into two broad categories, namely discretionary and non-discretionary (mandatory) models. Typically, discretionary access control (DAC) models leave the specification of access control policies to individual users, and control the access of users to information on the basis of identity of users. In mandatory access control (MAC) models the standard approach is to have the access defined by a system administrator, and employ attributes such as classifications and clearances [1]. Recently, there has been extensive interest in role based access control (RBAC) [2] even though the idea of the use of roles for controlling access to entities and objects is as old as the traditional access control models. In the RBAC models, the attributes used in access control are the roles associated with the principals and the privileges associated with the roles.

An important aspect of an access control model is the type of policies that the model can support. The model must be flexible enough to support the variety of access control requirements needed in modern application environments. Many of the proposed models are often inflexible because they assume certain pre-defined access policies, and these policies have been built into the access control mechanisms. In fact, in some sense, DAC, MAC and RBAC are all mechanism oriented. They all fall into the trap of using access control mechanisms both for policy expression as well as in policy enforcement. By separating out the policy expression from the mechanisms used to implement and to enforce the policy, a number of advantages can be achieved in practice. On the one hand, a mechanism can be used to support a number of policies, while on the other hand a policy may be supported by multiple mechanisms. Of course, it may not be useful or even possible for any given policy specification to be implemented by *all* of the existing mechanisms. It is well known from language theory that any mechanism for expression has its limits. For instance, it has been shown in [3] that the multilevel MAC mechanism is not adequate for implementing some role based access control specifications.

It has been claimed elsewhere that role based access control better suits the needs of some real world organisations than MAC or DAC based approaches [4]. While we support this, view, we believe that a number of issues in the design and implementation of RBAC systems have not been adequately addressed in previous work. Much of the work on RBAC models and systems has not addressed the issue of how to express policies in a real world system. We strongly believe that a language-based approach to authorisation is required to support the range of access control policies required in commercial systems. While some language-based proposals have been presented, such as in [5] and [6], these tend to either lack expressiveness or be highly formal.

It has been recognised that RBAC systems need to deal with privileges which reflect the operations of the application (such as *credit* and *debit* in a financial system) rather than the more traditional approach of a fixed set of operations such as *read, write* and *execute* [7]. However the effects of this on the design of such systems is rarely addressed. This is especially important in choosing this level of granularity of access control, which has obvious parallels with the ideas of object-oriented design.

Several RBAC proposals tend to avoid the question of ownership. Indeed, it has been claimed that RBAC is simply a form of MAC [8]. Expecting the system manager to make all policy decisions in a distributed system is

© IEE, 2000

IEE Proceedings online no. 20000792

DOI: 10.1049/ip-sen:20000792

Paper received 27th January 2000

The authors are with the Distributed System and Network Security Research Unit, University of Western Sydney, Nepean, NSW 2150, Australia

impractical and probably unnecessary in most cases. Some proposals such as in [4] have addressed this issue by suggesting that the system manager delegate authority over various objects to other users. These delegated authorities are then exercising control equivalent to that of ownership in DAC systems. It has also been noted that, in practice, the users of RBAC systems still wish to maintain individual control over some objects. Given this, the concept of ownership in relation to RBAC deserves further consideration.

If a language based approach to RBAC is adopted, then we also need to consider other issues, such as how a history of actions can be maintained. Such information is often necessary for access policies such as separation of duty. Consideration also needs to be given to how users are represented and assigned to roles.

2 A language based approach

Roles are intended to reflect the real world job functions within an organisation. The permissions that are attached to the roles reflect the actual operations that may be carried out by members of the role. The policies that need to be expressed in an RBAC system can therefore be seen to have the potential to be both extensive and intricate. The inter-relationships between structures of an RBAC system, such as role, users and permissions, likewise have the potential to become complicated.

Languages, in various forms, have long been recognised in computing as ideal vehicles for dealing with the expression and structuring of complex and dynamic relationships. Therefore it seems sensible to at least attempt to employ a language based approach for expressing RBAC. If the language is well designed, this will deliver a degree of flexibility superior to other approaches. Flexibility is a necessary requirement if the RBAC system is to be capable of supporting a wide range of access policies used in commercial systems. Other work has also recognised the possibility of using a language to express access policies [9].

While in theory a general-purpose language could be used, a special purpose language allows for optimisations and domain specific structures which can improve efficiency. In particular, the notion of roles and other domain specific concepts should be available as primitive constructs within the language. Certainly, the use of roles as a construct would help to simplify the problem of management of large numbers of access control privileges by grouping them according to job functions and tasks. The permissions associated with the roles tend to change less often than the people who fill the job function that the role represents. Being able to express the relationships between permissions and roles within the structure of the language would make the administration of the access control system simpler and this is a major advantage when it comes to management of authorisation policies. In fact, key aspects of any authorisation model are the ease with which one can change, add and delete policy specifications and the specification of authorities that are able to perform such operations. A language that does not support the necessary constructs will not fulfill these requirements.

In some sense, the question of who can modify the policy setting is what determines whether something is discretionary or mandatory. In general we feel that the traditional notions of discretionary and mandatory are not very helpful, in that a policy may be discretionary to some and mandatory to others. For instance, consider a manager of a project group and a number of group members. A policy can be mandatory to the group as it is set by the manager, but is discretionary to the group manager. Similarly at a higher level, a policy may be mandatory to the group manager but is discretionary to the laboratory manager above. This is typical of many organisations, and is often true in large distributed systems. It is often the flexibility and management of the operations of the access control system itself which are significant when it comes to the applicability of such a system to practical real situations. The use of a language based policy approach helps us to better structure such policies. Some recent work such as [6] and [10] have considered the use of a logic based language in the specification of authorisation policies. This paper proposes a language theory based approach, and is primarily targeted towards object based systems.

Of course, another reason for employing a special purpose language is that a general-purpose language will include many constructs not required for access control. This would impinge on the efficiency, safety and usability of such a system.

3 Role based access control language design issues

Typically, a role based access control model has the following three essential structures [7]:

- users: which correspond to real world users of the computing system
- permissions: a description of the access users can have to objects in the system
- roles: a description of the functions of users within an organisation

Permissions and users are mapped to roles, as shown in Fig. 1. The relationships are many to many.

3.1 Privileges and permissions

The 'classical' role based access model takes a simplistic view of the entities to which access is being controlled. When its application to object oriented systems is considered, it is to be noted that the *object*, the basic construct of such systems, is not directly represented within the classical model. In an object-oriented system, operations on objects are represented by the methods of the objects. In the classical model, permissions will therefore specify the object and the method to which access is allowed. If a user has access to a particular object, it is likely that the user will have access to more than one method of that object. While this will not always be true, it will generally hold. This could be handled by having a separate permission for each method. Given the number of methods to which users will have access, this is obviously cumbersome. More promising is the option of allowing a permission to control access to a number of methods. The language should be able to handle both these situations.

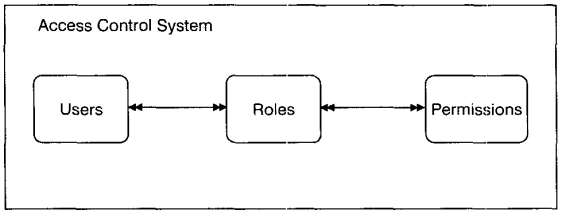

Fig. 1 *Classical role based access control model*

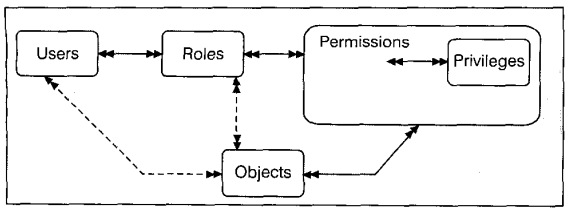

Fig. 2 *Object-oriented role based access control model*

Another important concept in object-oriented systems is that of a *class*. Users will often have access to a number of objects in a class, and may have identical access to such objects. Similar options to those for dealing with multiple methods should be possible. A separate permission for each object could be employed. Again this could lead to an unwieldy number of privileges. Alternatively, a permission could cover a number of objects as well as a number of methods. While this restricts the number of permissions, it gives them a complicated internal structure. Once again, the language needs to handle both situations involving access to a single object as well as access to a number of objects in a class. If these were handled within the permission, permissions would then be many to many mapping between objects and methods. Given that the mappings between entities in the classical model are many to many, it seems clear that two separate constructs are required.

The first construct, which we refer to as *permission*, specifies an object or objects to which a user has access. In this sense, it resembles a capability. The objects may be explicitly named [Note 1], be referenced by a named set, or a whole class could be referred to. Note that, as privileges generally refer to specific methods, the objects covered by a permission must belong to the same class (or a common super-class). Using named sets allows grouping of objects, and such groups may change dynamically.

Privileges, the second construct, are mapped to permissions in a many-to-many relationship, specifying the methods of those objects to which access is granted. Objects do not appear in privileges, which are concerned solely with methods and the conditions under which they can be accessed. The structures and the mappings between them are shown in Fig. 2. This model takes into account the objects, which are the chief structuring concept in an object-oriented system. The dashed lines in Fig. 2 represent ownership relations, which are discussed in section 3.4

3.2 Users and sessions

A basic structure in the language is that of the *user*. The elementary requirements for a RBAC language include the ability to specify a user (such as a name and a user identity), and the roles for which the user is authorised. Users can have access to multiple roles that they can assume as their authorised roles. Note that users may not always want one of their sessions to have access to all the user's objects and permissions. For example, a user may wish to run some untrusted code with only a small subset of their normal access.

For a particular process or login session [Note 2], a user may wish (or be required to have) the access to be governed by a subset of their authorised roles. We shall refer to this subset as the *active* roles. In a session, allowed accesses to objects are limited to those within the current active roles for that session.

If a user is *authorised* for a given role then a user may adopt that role as an *active* role. A role is not used for access checking until it becomes active. When a role becomes active it does so for a particular *session* of the user. Some systems allow only one role to be active at any one time. Others allow any number of active roles and, indeed, the user may have different roles active for simultaneously existing sessions.

There is obviously the need for a *session* structure that records the active roles, i.e. the ones actually used to determine whether accesses attempted in that session will be allowed or not.

3.3 Roles

Let us now consider the *role* structure in the language. Roles delimit the functions of users within organisations by prescribing the access to objects which users have. It must be possible within a role to specify a set of permissions. There are two other important considerations in determining what must exist within a role. RBAC models often need to include the concepts of role hierarchy and constraints. Role hierarchies allow roles to be constructed from other roles. Constraints place restrictions on how users are assigned to roles.

Regarding roles as simply sets of permissions is not sufficiently powerful, as the relationships between job descriptions within real world organisations can be quite complex. If a RBAC system is to accurately model real world systems then it must support some form of role inheritance. This follows from the observation that one real world job description is often an extension of another. For similar reasons, it must be possible to specify constraints on users taking on roles (for example, users may not be allowed to take on one role if another role they already have specifically disallows it).

3.3.1 Role inheritance: Role inheritance can be modelled by allowing roles to be formed, in part, from other roles. The one possible consequence of building roles from permissions and other roles, which themselves consist of yet other permissions and roles (i.e. inheritance) is the issue of conflicts between the permissions (and their privileges) of the various roles. Consider the simple example of a role *branch_manager* that inherits from both the role *accounts_manager* and the role *loans_manager*. Both *accounts_manager* and *loans_manager* inherit from the role *teller*.

There are two points that require discussion

- what is the effect of the *branch_manager* having the permissions of the *teller* twice (once through *accounts_manager* and once through *loans_manager*)?
- what if the permissions unique to *accounts_manager* and *loans_manager* are in conflict at some point?

The answer to the first question should be that there is no effect. The operations allowed by an active role, its privileges and permissions should not depend upon how they were retrieved internally by the RBAC system during access checking. This is the classic answer to this question when multiple inheritance is allowed.

The second question revolves around the forms of conflicts that are possible in the language. If the language allows negative permissions (i.e. operations a user is

Note 1: Permission may not actually include the name (as in directory path) of the object, but could hold a pointer to the object or other system dependent construct.

Note 2: As systems vary significantly in their treatment of processes and login sessions, we shall simply refer to sessions and leave the details to implementations.

specifically barred from carrying out) then a conflict resolution mechanism is required. If the language allows only positive permissions, then a decision must be made as to whether the most restrictive or most generous permission will apply; and this is a policy issue.

For example, the accounts manager may only be allowed to examine loan balances during business hours, but the loans manager may be able to examine them on Saturdays as well. If the branch manager is attempting to check a loan balance on a Saturday, the privilege inherited from the accounts manager's role may be checked first. Although this permission does not allow the action, the search continues and eventually the appropriate permission, inherited from the loans manager's role, is checked. As the branch manager is inheriting all the functions of the loans and accounts managers, it follows that the least restrictive form should apply. In this case, the second option of most generous permission applies. Similar arguments apply regardless of whether the conditions conflict in external factors (such as different limits to time or place of access) or internal factors (such as status of the object or access control system).

The other requirement for role inheritance is that it should be possible for the inheritance to be partial. That is, a role inherits some (but not all) of the permissions of another role. It has been noted that in practice [11] the real world organisations that roles are intended to model do not always display total inheritance. The language must be able to support this requirement. It must be possible to specify that one role inherits all the permissions of another, less certain stated permissions. The converse (explicitly stating those privileges to be inherited) does not really require an ancestor role at all, although it may be useful for modelling purposes. This partial inheritance provides the ability to limit the depth of inheritance. In some sense, this captures the effects of overriding inheritance and being able to define certain private roles. This feature is important to enable specification of principles such as the separation of duty and delegation. For instance, this feature helps to address some of the issues raised in [12] concerning role inheritance and enforcement of control principles. Real world job descriptions may be related by some overlap of job functions, rather then the strict subset relationship implied by simple inheritance. It may be useful, for modelling purposes, to create roles which contain the common functions of job descriptions, and have the roles which correspond to the actual job functions inherit from those roles. As the ancestor role does not correspond to any actual real world job, it can be nominated as a virtual role and no user may then have it as an authorised role.

3.3.2 Role constraints: Constraints on roles limit the possible roles that may be active and/or authorised for a user. Constraints between roles must be checked at two points. First, when the role is authorised for the user; secondly, whenever an attempt is made to make the role active for a particular session. While a user may be authorised for two roles, it should be possible to prevent the user being simultaneously active in both roles. This decision on whether a role can be made active for a session may be based solely on the roles already active for that particular session, or on the roles active for all current sessions of the user.

In general, constraint decisions can be based on a number of factors, such as which roles are already authorised or active for the user or session, or how many users are already authorised or active in that role. It should be possible to stop a role being authorised or active based on other role assignments of the user or session (mutual exclusion). It should also be possible to request that another role is authorised or active for the user before a role assignment is made (prerequisite roles).

3.4 Object ownership

In most role based access control systems, there appears to be an implicit assumption that objects are explicitly referred to in the permissions. In general, it could be a named set whose membership can change dynamically. While this is often what is required, it can lead to problems. Specifically, what happens when a new object is created? If objects are explicitly named within permissions, then permissions have to be updated or created every time a new object is created. This is an unwanted and unnecessary overhead. It also violates the general requirement that the access control system interfere as little as possible with the rest of the system. However, if objects are not explicitly named within permissions, how can one determine to which objects a permission refers. While the possibility of specifying a class, rather than a group of objects, within a permission, partially solves this problem, there are many occasions when users should not have access to all objects of a given class.

The concept of ownership, found in other access control systems, can be used to provide a solution to this problem. This would allow permissions to be written so that access to objects was granted only to the owners of the objects. Such permissions could be written in advance of the objects coming into existence, and would not need to refer to the objects by name. Ownership can be checked within the privileges and permissions, allowing dynamic creation of objects without each such creation requiring update of the privileges and permissions. While the information for a newly created object (who or what owns it) must be entered into the access control system, this is no more information required than in any other approach to access control.

The immediate question that arises is where to vest the ownership - in users or in roles? The obvious answer, for a role based system, may be in the roles. However, consider the following example. A system may have a basic role, *user*, which includes a permission giving access to the factory object for word processing documents. If ownership of word processing objects (documents) was vested in the role, then all members of the role *user* (i.e. probably every user of the system) could access the word processing objects. This situation would (at least in general) be undesirable. More sensible would be that ownership of a new word processing object would be vested in the user that requested its creation.

However, restricting ownership to users only is also not entirely satisfactory. For example, consider the role *budget_working_group*. It is likely that access to spreadsheet documents created by members of this role should be automatically granted to all other members of the role, not just the particular member who created the document. The granting of such access should be on creation of the document and should not require manual input for every new document. It would not be feasible to say that all spreadsheets are owned by the role through which access was granted to their creating object, as this would preclude private ownership of such objects. As a further example, consider that *user* and *budget_working_group* may exist on the same system and that the *budget_working_group* may

produce word processing documents as well as spreadsheets.

Another possibility is to use the concept of role inheritance to guide ownership, if ownership is to be vested solely in roles. Ownership is then granted to the role that is actually the active role of the user, not any intermediate roles from which the active role has inherited. Therefore, if *budget_working_group* was the active role of the user [Note 3], then ownership of any objects created would be vested in the *budget_working_group*. However, allowing 'private' ownership of objects would then require the creation of 'private' roles for each individual user, each of which inherits basic permission from the role *user*. Each user could retain ownership of private objects by having this private role as their active role. The result would be that no user would ever have *user* as their active role – which, at the very least, does not follow the spirit of role based access control.

It thus seems clear that it must be possible to vest ownership of an object in one or more users, one or more roles, or some combination thereof. A user owning an object has a clear meaning. With role ownership the choice is between requiring the user attempting the access requiring ownership to have that role as one of their currently active roles, or to have that role as one of their possibly active, but not currently, active roles. The latter case restricts the possibility of sand-boxing operations and hence we advocate the former. The final argument in favour of allowing user-based ownership is that practical experience with role based systems has found that it is impractical to do away with them. Such abilities should be encompassed within the RBAC system, as otherwise a separate access control system would be required to handle them. This would then produce obvious problems in coordinating and understanding the two access control systems.

Whether the information regarding ownership is stored with the object or elsewhere (and accessible to the access control system) or by the access control system (and if so exactly how) is implementation dependent.

3.5 History and variables

Access control decisions will often need to be made based on the previous actions of the user attempting access (or of other users with the object in question or even other objects). This is as true for RBAC systems as for any other, and various proposals for dealing with this have been made, e.g. in [5, 13]. Most of these are fairly restricted mechanisms, recording simply the past occurrence of actions (or transactions).

In taking a language-based approach we have the option of a more flexible mechanism. One of the important features of most programming languages is the manner in which they store information. We propose that a language for RBAC should allow the declaration and use of variables to hold information useful for such purposes as tracking the history of user actions. Access control decisions can be based upon the values of such variables; we refer to such variables as meta-variables, to separate them from the variables of the programming languages used to write the objects being protected [Note 4]. The values of meta-variables can be checked in the condition clauses of the various structures of the language, such as privileges and permissions. Changes to the values can also be included in these structures. This will allow the state of the access control system to automatically change in response to user actions; hence such variables are particularly useful when it comes to modelling dynamic access control. For instance, while classical role based access control is able to model static separation of duties, modelling dynamic separation of duties is often less straightforward [14, 15]. While the need for dynamic separation of duties is commonly acknowledged, the details of how it is to be achieved in practical implementations are often omitted. A common example of dynamic separation of duties is the initiation and authorisation of a payment. While members of a given role, such as *accountant*, may be allowed to initiate and authorise payments, a given member may not be allowed to both initiate and authorise a particular payment. Assume that the payment objects do not record the identity of the initiator and do not enforce the separation of duties. It is then the responsibility of the access control system to record this information and enforce the separation. When the payment is to be authorised, the identity of the authorising entity can be checked against that of the initiator. This is achieved using meta-variables. In fact, we will see later that meta-variables are also useful when it is necessary to limit the number of times a permission can be used, and to alter the state of roles and collections and record parameters to actions. Recording parameters with actions has been suggested in the definition of role templates for content-based access control [16].

3.6 Attributes

Roles are intended to reflect the real world job functions within an organisation. In practice, job functions often encompass attributes such as physical ones (e.g. location) and job classification (e.g. probationary period). Attributes can also be used in handling MAC and Lattice based controls (such as a Chinese Wall policy [17]).

For example, consider a wholesale company with a number of warehouses. There may be a role within the company's computer system *stock_inventory_clerk*. Users in this role have the responsibility for entering movement of goods into and out of the warehouses. However, there is no need (and possibly good arguments exist against) for such clerks being able to alter the values at warehouses other than the specific ones at which they work. This could be handled by having a separate role for each location and including within each such role privileges for the objects at each location; however, this simply results in a large number of virtually identical roles. This can be avoided by giving each user and object a location attribute and checking (in the conditions of privileges and permissions) whether the values of the attributes match at the time of access.

It may be thought that the attributes that specify the fine details of a job description are unnecessary. For example, whether a clerk is in a probationary period or not could be handled by having two separate roles. However, the number of such attributes and their interaction, in practice, can again lead to an explosion in the number of roles. A quoted example [18] is for 3125 separate sub-categories of bank teller arising only from different attributes. It is obviously far easier to handle these using attributes than by using multiple roles.

We believe that it is impossible to precisely define in advance all the possible attributes that could be of interest

Note 3: For purposes of the example in this paragraph we assume that each user can have only one active role at a time. This is unnecessarily restrictive, but allows a clearer presentation of the example.

Note 4: The term 'meta' in this paper is used to refer to operations and values within the access control mechanism itself, as opposed to those of the entities to which access is being controlled.

to the designers of security systems. However we believe that it is possible to define a useful set of attribute types. This is inline with programming language design, where language designers do not attempt to pre-set the variables available to programmers. Therefore an RBAC language should allow attributes of various names and types to be associated with users and objects. The designer of a security system can then decide which situations require separate roles and which can be handled by a single role with differing attribute values.

Ownership, as discussed in Section 3.4, can be thought of as a special attribute. It can be tested in a privilege, as can any other attribute. This allows users to delegate access to other users on the basis of the objects they own. For example, a user could delegate access to a secretary on the basis of the objects s/he owns. Of course, this is a powerful form of access and the user would need to set lifetimes on the privileges. Lifetimes could be set by simply checking the current time within the privilege and ensuring that it is not beyond a cut-off point.

3.7 Access control algorithm

Based on the above discussion, the general structure of a permission comprises

- a set of **methods** which are governed by the permission
- a **condition** which determines whether access is allowed to the methods
- an **action** (or alteration to the value of meta-variables) which is carried out when permission for access is requested or granted

Requests to the access control system are of the following form:

```
access (s,o,m,p)
```

where

s is a session identifier, which can be used to identify a user
o identifies an object being accessed
m identifies the method being invoked
p is the set of parameters for the invocation

The access control system will report whether the method invocation is to be allowed or disallowed, according to the following algorithm:

```
for each active role for the session
 for each permission in the role
  if the permission applies to the object
    for each privilege in the permission
      if the method being invoked is one of those listed
      for the privilege and
        the privilege's condition evaluates to true
        then execute the action of the privilege
             allow the access
```

Note that the ordering of roles, permission and privileges is significant in this algorithm; the ordering determines which action section will be executed if more than one is possible.

4 Access control language: Tower

In this section, we briefly describe major features of a language called Tower that is currently being developed to specify role based access control in distributed systems.

The most important structures in Tower are the definitions of users, roles, permissions and privileges. Each structure is declared and is given a name. The name is used to identify the structure throughout the access control system. Therefore each of these names must be unique and these can be block structured for scoping. A new structure instance may be created and assigned to a structure variable. The closure of a structure includes any variables declared in the same scope. The structures are immediately available upon creation for evaluating access requests. They may also have their values modified in code which is subsequently executed. In this paper we do not specify the management interface of the access control system. We envisage that both users and administrators can enter policies (in the form of Tower expressions) into the system. Whether this is in a form similar to the Adage VPB [9] or by some other means is not relevant to the design of the language itself.

4.1 Block structure and variables

For an access control system to function it will require some capacity for storing information about the objects it manages and the access policies to be enforced. The Tower language allows the specification of information internal to the access control system in the form of variables. There are two distinct categories of variable in Tower, which differ in the type of information stored, their scope and use. These categories are

- Simple variables (henceforth referred to as variables)
- Structure variables

The types of (simple) variables supported are the standard ones such as integer, real, Boolean, string, userid (user identity) and sets.

From the point of view of such variables, Tower is a block structured language. A block consists of the definition of either a role or permission or statements between matching *begin* and *end* statements. Within a block, variables are declared before any roles, permissions or interior blocks. A variable is in scope within the block in which it is declared, within any structures declared within that block and within any interior blocks (except for further declarations using the same variable names) and any constructs defined within them. Variables declared within permissions or roles are only in scope within those constructs. Variable declarations have the following syntax:

var_name[= *value*], var_name[= *value*], . . . : var_type

except for set variables, which are declared as

```
setvar_name[= value], setvar_name[= value], :
set of element_type
```

A set variable of any set type may be empty.

The optional section after each variable name allows the value of variables to be initialised when declared. The value of variables can be altered in subsequent code, especially in the action sections of privileges (see below). The values of variables may be tested within condition expressions and constraints. Any attempt to access a value of a variable before it is initialised results in an error.

Each variable name may be followed by a '*' or a '&' (or both). These control the actual number of instances of the named variable and their effect within the current scope. If neither symbol follows the variable name in the declaration then only a single variable is created. If the variable name in the declaration is followed by either or both of these symbols then more than one variable, each with the same name, is (potentially) created within this scope. If a variable's name is followed by a '*', then a separate such variable is created for each object covered by the permission(s) within the scope of the declaration. If a variable's name is followed by a '&', then a separate such variable is

created for each user whose access requests involve this scope. If both symbols occur, then a separate variable is created for each user/object pair. As it cannot always be known in advance which users and objects will be involved, these variables are created dynamically as required. As accesses to variables only happen when a request to a specific object by a specific user occurs, it is straightforward for the system to determine which variable is to be used in any particular case.

Structure variables cover the following constructs within Tower: permissions, privileges, roles, users, ownership and blocks. Such structures are declared using the syntax given above for variables; as with the variables, their values must be initialised before use (the exception to these provisions is blocks). The details of how values for these structures are created are covered in the following sections. Apart from the obvious differences between structures and variables in terms of syntax and value, the chief difference between them is the scope of structures. Unlike variables, which are only in scope within the block or structure within which they are declared, structure variables can be in scope within the entire access control system. The decision on scoping must be made when the structure variable is declared. Global scope is the default; if a structure's scope is not to be global then its name must be followed by a '@' character in the declaration. The unique user identification of the user who created the structure can be pre-pended to its declared name to ensure uniqueness.

The exception to the above is blocks defined by *begin* and *end* keywords. Any such block is considered to be global if it is not defined within another block. Blocks do not need declaration but can be given a name, as in the following:

```
my_block := begin
  ...
end
```

The name of a block can be used to add additional structures or variables to the scope, it represents. That is, Tower is not a statically scoped language but to some extent it is dynamically scoped. This is related to database schema evolution.

Many constructs within Tower are based upon sets. The language provides a number of operations upon sets for all of these constructs. The following standard set operations are provided:

- union, e.g. Set1: = Set2 + {element1,element2},
- difference, e.g. Set1: = Set2-Set3,
- intersection, e.g. Set1: = Set3/Set4,
- test for inclusion, e.g. element1 **in** Set1,
- cardinality, e.g. size(Set1),
- equality, e.g. Set1 = Set2
- subset test, e.g. Sett1 < Set2.

The operators are type-sensitive, i.e. the types of all the sets involved must match and the types of the elements must match the declared element type of the sets.

4.2 Ownership of objects and structures

As described in Section 3.4, the concept of ownership can simplify the expression of access control policies. Many systems limit ownership to a single user. This does not match many real world situations, where ownership is often equally shared between many people. For example, all members of the committee may jointly own a document produced by a committee. Vesting ownership in more than a single entity leads to the question of how many of these entities must co-operate for successful performance of actions restricted to an owner. In Tower, we employ a relatively simple answer to this question: for each object, the number (or fraction) of the joint owners who must agree before an action can be performed is stored along with the ownership information.

Each object (and class specification) stored in the system has a corresponding access control structure. These structures record the owner(s) of the corresponding object and other related information. While the creation of the ownership structures is automatic on the creation of the corresponding object, they have a conceptual Tower syntax. This allows for updating of the ownership information within the scope of the language.

```
name := object
  owners
  {uuid, uuid, ...}|{role, role, ...}|{uuid, uuid, ...}{role, role, ...}
  quorum positive integer | real between 0 and 1 |all
  creation {uuid, uuid, ...}|{role, role, ...}|{uuid, uuid, ...}{role,
role, ...}
                                              [variable declarations]
end_object
```

The name of the structure is the system dependent unique object identifier. The first clause specifies the owner of the object, as one or more specified users and/or the members of named roles. The second option allows for a dynamic concept of ownership, as it grants joint ownership to all users who currently have at least one of the named roles as an active role [Note 5].

The second clause specifies how many of the owners must agree if any operation requiring owner approval is to be carried out. For an object there are only three such operations:

- changing any of the information stored in the ownership structure (including the specification of the owner),
- allowing the object (or class specification) to be referenced from within a permission, and
- revoking the ability of the object (or class specification) to be referenced from within a permission.

The second operation prevents users from including objects within a permission when they do not own that object. The third operation allows for revocation of access.

The third *creation* clause specifies the owner of any object created as a direct result (i.e. without subsequent accesses to other objects) of access to this object. For example, while the owner of a text editor may be the system manager, any files created using the text editor can be specified as belonging to the user who accessed it.

The same principles of ownership can be applied to structures of the access control system (roles, permissions, privileges, users). The syntax is the same as that given above, except that the keyword *object* is replaced with *structure*. The name of the ownership structure is that of the structure to which it applies, followed by the special character '^'. This allows us to control access to the access control system itself in a conceptually efficient manner. Each structure in Tower has an associated ownership structure. The ownership information in such an ownership structure also applies to itself, avoiding infinite recursion. Thus it is possible to specify who owns each structure and can therefore modify it. This also allows us to restrict the use of access control structures; they can only be altered or used (included in the values of other structures) either by

Note 5: While we could have simply allowed the role to be an authorised role, insisting that it must be an active role helps protect untrusted code running using limited privileges.

their owner, or with their owner's permission. In the case of removing one structure from another (such as removing a role from a user's list of authorised roles), the permission of the owner of either structure is sufficient.

4.3 Privileges

In an object-oriented system, it is reasonable to base the lowest level construct of the access control system at the method level. In Tower, a privilege is a triple, consisting of the set of names of the methods to which it gives access, the condition under which access is granted and any action to be taken within the access control system if access is granted. A new privilege is created as follows:

privilege_name := **privilege**
  [condition_expression]
  [action_statement,action_statement, ...]
  {*method_clause, method_clause*, ...}
end_permission

The condition expression and the set of action statements are optional. The condition expression is a Boolean expression (of arbitrary complexity) which must evaluate to true if any of the methods is to be invoked under the authority of this privilege. A condition expression can test the state of the object being invoked (by itself invoking methods of the object) and the values of parameters passed and access control system variables in scope. An action statement can only be executed if the invocation of any of the methods is allowed under the authority of the privilege (the default) or *whenever* the condition expression is tested (via the use of the keyword **always**). When an action statement is executed, the state of the access control system is altered. A *method_clause* is either a method name or a set of method names.

Note that there is no specification within a privilege as to the objects to which it applies. This is handled at the permission level. While users will probably have access to multiple methods of each object, they will not necessarily be able to access those methods under the same condition. We therefore associate conditions and methods in a privilege and group privileges together with a specification of which objects they apply to within permissions. Those methods of an object to which the same conditions apply may be grouped together in the method set of a privilege.

4.4 Permissions

Permissions encapsulate the access to objects of a single class. A permission consists of a specification of the objects to which it gives access and how those objects can be accessed. The latter is specified as a set of privileges. A permission will give access to some subset of the objects of the class. Normally the subset will be a proper subset, and not all the objects of the class. This restriction reflects the observation that normally a user will not have access to all the objects of a class (unless they are the only user who can access objects of that class). It would be an unusual situation where, for example, a single user would have access to all spreadsheets or all text documents in a multi-user system. However, it is usually impossible to specify in advance the names (or other identifiers) of all the objects of a given class to which a user will have access. A permission can specify that it allows access to objects of a class owned by a given set of users. This allows access ||control to be specified for objects which have not yet come into existence. The syntax for creating a new permission is as follows:

permission_name := **permission**
  class_name
  [**owner**]
  [**users** *user_set*]
  [**roles** *role_set*]
  [**objects** *object_set*]
  [variable_declarations]
  privileges {*privilege_clause, privilege_clause*, ...}
end_permission

The *class_name* gives the name of the class of the object to which this permission grants access. After that are clauses specifying the objects covered by the permission. The objects to which the permission will grant access may be specified in terms of their ownership. If the keyword **owner** is employed then the permission can grant access to objects of the named class owned (singly or jointly) by the user attempting to gain access. The permission may grant access to objects of the named class owned by any of the listed entities. This may be a set of explicitly named users, or users which currently have the named role as an active role. The permission may be defined to give access to a set of existing objects by explicitly naming them. The permission can then be used to access those objects and no others. Finally the object set may be a named object set, which can be dynamically updated without directly accessing the permission.

If an access is attempted to an object which is not to one of the named objects then this permission will not grant access. Of course, even if the object which is being accessed is the one covered by the permission access may still be denied according to the privileges included within the permission. Tests for ownership may also occur in the condition sections of privileges, but such tests are additional (not an alternative) to the permission level tests.

Then any variables which are in scope within the permission are declared. Finally, there is a set of privileges which define the exact access allowed by the permission. A *privilege_clause* is either a privilege, a privilege set or a *privilege_expression*. A *privilege_expression* is an expression specifying changes to a privilege (such as adding or subtracting methods, conditions or actions).

The following gives an example of the initialisation of a permission and the effects of ownership. A user *a* wishes to access the objects of class *text_object* owned by user *b*. *a* enters the following code:

b_text := **permission**
  text_object
  users *b*
  {*privilege, privilege*, ...}
end_permission

The permission will be created *if* both the owner of the class definition for *text_objects* and user *b* give their permission. The method by which they would do this relates to the management interface and is outside the scope of this paper.

4.5 Roles

The syntax for creating a new role value is as follows:

role_name := **role**
  [variable_declarations]
  [**authorised** constraint_expression
  [constraint_action]]
  [**active** constraint_expression
  [constraint_action]]
  [**session** constraint_expression
  [constraint_action]]
  [**roles** {*role_clause, role_clause*, ...}]
  [**permissions** {*permission_clause, permission_clause*, ...}]
end_role

Role constraints may be expressed to affect the roles at three different levels:

- the roles that a user may be authorised to have as active
- the roles that a user has active across concurrent sessions
- the roles that a user has authorised within a particular session

These are in increasing level of refinement - if a role specifies that no user can have both it and another role as authorised roles, then obviously the user cannot have both those roles as active roles (either in the same session or in another one).

Constraints in a role may be used to impose restrictions upon whether a user may have this role added to his/her set of roles, or whether a user may add another role while possessing this one. Such a constraint is specified as a Boolean function which must evaluate to true if the role is to be added. A shorthand is provided for the common case of exclusion, which is that possession of the current role is mutually exclusive with the roles in the role set.

exclude *role_set*

This set can be explicitly listed in the constraint expression or represented by a set variable, allowing easier dynamic update.

The constraint action allows for updating of any variables relevant to the constraint. The role and permission sections define the access allowed by the newly created role. The definitions of *role_clause* and *permission_clause* are analogous to that of *privilege_clause* in Section 4.4. Role inheritance is modelled by allowing roles to be formed, in part, from other roles. These roles may already exist, and are referred to by name, or are defined within the new role.

4.6 Users and sessions

The syntax for creating a new user structure is as follows:

user_name := **user**
  name
  uuid
  [{*role*, *role*, ...}]
end_user

Note that the roles are those which the user *may* take on (known as the authorised roles of that user). When a new user is created this set may often be empty. In addition to explicitly naming roles, one or more role sets could also be given.

For each login session of a user, it is also necessary to record the actual roles that are currently active. It is the active roles that are used to check whether any attempted method invocation should be allowed.

The syntax for a session is:

session_name := **session**
  user_name
  uuid
  [{*role*, *role*, ...}]
end_session

Note that in some sense this a conceptual syntax; as such structures would be implicitly created whenever a new user session is commenced. However, they have an actual existence and are used in checking role constraints as well as actual method invocation.

4.7 System evolution : Alterations to structure values

We have described how the various structures of the language are given their initial values. As the system evolves, any of these structures may need to have their values updated. Set operations may be applied to each of these structures, for example

$$P1 := P1 + \{Pr1\ Pr2\}$$

Permission P1 now has privileges Pr1 and Pr2 added to its set of privileges. The type of Pr1 and Pr2 means that the update must be to the privilege set of the permission. Therefore we can simply use the permission name without further qualification. This applies to all the components of structures that can be unambiguously identified. Where a structure consists of two or more sets of the same element type, such as the record of the owners of an object and the owners of any new objects, further qualification, and updates occur as follows:

$$object1.owners := object1.owners + \{michael\}$$

$$object1.creation := object1.creation + \{vijay\}$$

The first statement adds the user *michael* to the set of users who own object *object1*. The second statement adds the user *vijay* to the set of users who will own any objects created using *object1*.

From the above, the set operations applied to a privilege alter the contents of its set of method names (as the only set contained in a permission is the method set). Similarly, the roles and permissions which make up a role can be altered, as in the following examples:

$$R := \{P1, P2\}$$

The permissions in R are now P1 and P2.

$$R1 := R1 + \{R2, R3\}$$

R1 has R2 and R3 added to its roles

$$R1 := R1 - \{R4\}$$

R4 is no longer one of R1's roles

The system can determine if the roles or permissions of a role are being updated by resolving the names on the righthand side of the assignment statements.

The other information held in a structure may also be updated within assignment statements. For example, the condition within a privilege may be added to. For example,

$$Pr1 := Pr1 + \text{condition expression}$$

The new condition expression for the privilege is formed by joining the previous expression and that in the assignment statement with the *and* conjunction.

5 Access policy examples

The basic constructs and structures of the Tower language can be used to specify a range of access control polices. This section describes some commonly used access control policies using Tower. In the interests of space, the examples do not include all the necessary preliminary declarations and initialisations. Nevertheless we hope they convey the required information.

5.1 Role hierarchy

One of the most important advantages claimed for the role based approach is that it can model organisation structures. A simple (possibly simplistic) view of such structures is a

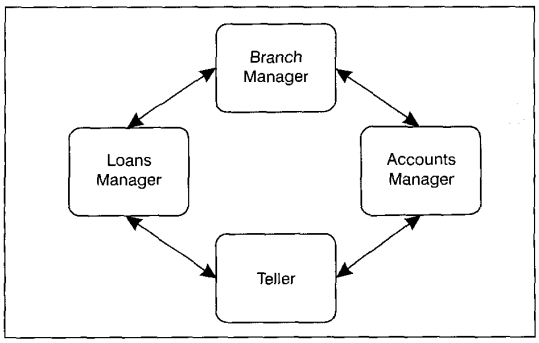

Fig. 3 *Example of role inheritance*

hierarchical ordering of responsibilities, with more senior positions encompassing all the privileges of the more junior positions, plus some extra privileges. For example, consider a hypothetical structure for a branch of a bank, as shown in Fig. 3.

Let *teller_role* be a role containing all the appropriate permissions and privileges needed for a teller to carry out his or her functions. Let P1 and P2 be permissions which contain the extra privileges required for an accounts manager. Let P3 and P4 be the permissions which contain the extra privileges required for a loans manager. Let permissions P5 and P6 contain the extra permissions required for a branch manager.

The roles for accounts manager and loans manager can be created as follows (we assume appropriate variable declarations):

```
accounts_manager_role := role
   roles teller_role
   permission {P1,P2}
end_role
loans_manager_role := role
   roles teller_role
   permissions {P3,P4}
end_role
```

Note that both these roles inherit all the privileges of the teller role. The role for the branch manager can be created as follows:

```
branch_manager_role := role
   roles {accounts_manager_role, loans_manager_role}
   permissions {P5,P6}
end_role
```

Note that the privileges associated with the teller role are indirectly inherited by the branch manager role. In fact, it is also possible to inherit only some of the privileges. For instance, the bank manager's role could inherit all the privileges of accounts manager except say x1, and all the privileges of loans manager except x2. This would be specified as:

```
branch_manager_role := role
   roles {accounts_manager_role - x1, loans_manager_role - x2}
   permissions {P5,P6}
end_role
```

While this may be considered a simplistic example, it does demonstrate role inheritance in Tower. Other examples involving partial overlap rather then strict inheritance can be modelled using virtual roles, as discussed in Section 3.3.1.

5.2 Role hierarchy with private roles

In the previous section, it was implicitly assumed that *all* actions that can be carried out by other staff can also be done by the branch manager. For example, there are no private files for correspondence and record keeping. While this may well be the policy for a bank, in practice it would also allow some privacy to its employees. We would therefore require that not all privileges be inherited. Privileges may need to be shared amongst all holders of a position, but not inherited or we may require privileges to be private to individual users.

For example, for privileges that are to be shared by all loans managers but not inherited by branch managers, each loan manager has their user structure defined as:

```
user_structure_name := user
   (name of loans manager)
   (uuid of loans manager)
   {loans_manager_role,private_loans_manager_role}
end_user
```

where *private_loans_manager_role* contains those privileges not to be given to branch managers.

A user can be allowed their own private privileges by creating a role for which they will be the only authorised user. For example, for a teller *john*, a role called *john_private_role* could be created and their user structure would be

```
john_user_structure := user
   john
   (uuid of john)
   {teller_role, john_private_role}
end_user
```

5.3 Separation of duties

5.3.1 Static separation of duty: Consider a class where one group of users is allowed to add items to an object, and another is allowed to remove items from the object; for example, items are produced by one group of users and submitted for certification by another. Between creation and certification, the items are held in a container object. This situation is a simple example of static separation of duties and can be represented in Tower as follows:

```
create_privilege := privilege
   {create}
end_privilege
certify_privilege := privilege
   {certify}
end_privilege
create_permission := permission
   container_class
   objects {container_object}
   privileges {create_privilege}
end_permission
certify_permission := permission
   container_class
   objects {container_object}
   privileges {certify_privilege}
end_permission
creator_role := role
   authorised exclude certifier_role
   permissions {create_permission}
end_role
certifier_role := role
   authorised exclude creator_role
   permissions {certify_permission}
end_role
```

Only one constraint expression is actually necessary. For completeness, a constraint expression is included in both.

5.3.2 Dynamic separation of duty

Tower can also handle dynamic separations of duty. Consider a class of cheque objects, which may be accessed by members of the role *accountant*. However, the same user may not both issue and authorise the same cheque.

```
begin
  issuing_user* : userid
  issue_privilege := privilege
    issuing_user := user
    {issue}
  end_privilege
  authorise_privilege := privilege
    (issuing_user <> user)
    {authorise}
  end_privilege
  cheque_permission := permission
    cheque_class
    privilege {issue_privilege, authorise_privilege}
  end_permission
  accountant := role
      permissions {cheque_permission}
  end_role
end
```

Note that one copy of the variable *issuing_user* is created for each object covered by the *cheque_permission* and its privileges. The value of the variable is set in the action part of the *issue_privilege* and checked in the condition part of the *authorise_privilege*. The role *accountant* does not need to be declared within the block, but placing it within the block aids readability.

5.4 Chinese Wall policy

The Chinese Wall policy [17] can be viewed as a special form of dynamic separation of duty. In this policy, objects are grouped together into different sets which reflect conflicts of interests. If a user has accessed an object in a set, then the user is not allowed to access any other object within that conflict of interest set. For example, if company A and company B are in the same conflict of interest set and if a user is acting as a consultant to company A, then s/he is not allowed to act as a consultant to company B.

The operations for each company are placed in a separate role. A conflict of interest set is represented by the set of each of these roles. The constraint expression for each role must reflect, on a per user basis, the actual role which has been accessed.

```
begin
  companyA, companyB : role
  user_company = {} : set of role
  companyA := role
    authorised user_company = {}or user_company = {companyA}
    user_company := {companyA}
    permissions {permissions for companyA}
  end_role
  companyB := role
    authorised user_company = {}or user_company = {companyB}
    user_company := {companyB}
    permissions {permissions for companyB}
  end_role
end
```

Objects may affect more than one conflict of interest set. Consider the example where the users are consultants to various firms. One conflict of interest set is accounting firms, and another mining companies. An object, which holds information about both a mining company and an accounting firm, checks on both conflict of interest sets for a user. As the labels associated with objects and users are global in such systems they are handled in Tower by including all conflict of interest variables and roles in a single Tower block.

5.5 Delegation

Delegation within Tower is handled by dynamically assigning and de-assigning roles. While roles are usually thought of as broad concepts covering complete job descriptions, they can also be used in a much more fine-grained manner. The collections representing the delegated authority can be placed in a new role. This role can be added to the authorised role of the user (which may represent a real world user or some active system entity) to whom the authority is to be delegated. When the delegated actions are completed, the role can be removed.

Consider the following delegation situation [18]: a departmental manager has access to view and modify the overall departmental portfolio object DP. The department may have several projects, each of which has an individual portfolio object DPi. A project manager can only view or modify his or her own portfolio object. A project manager, *pm*, can only view or modify another project's portfolio if, and only if, the departmental manager *dm* has delegated the appropriate privilege to it. That is, in this case, the project manager is acting on behalf of the departmental manager. This could be handled in Tower by the *dm* executing the following:

```
delegated_rights := permission
    ...
end_permission
delegation_role := role
    permissions {delegated_rights}
end_role
pm := pm + delegation_role
```

The departmental manager (after executing the above code) has created a new role holding the delegated permissions, and this role has been added to the set of authorised roles for the project manager. As the departmental manager is probably not the owner of the user structure for the project manager, the last line will not take effect until the project manager (assuming he/she owns his/her own user structure) gives permission for the update. As the department manager retains ownership of the new role and collection, the project manager cannot pass on the delegated rights without the department manager's agreement.

However, the department manager may wish the project manager to be able to further pass on the delegated permission without referring back to the department manager. This can be handled in a number of ways. The department manager could transfer ownership of the new structures to the project manager. The project manager could then distribute them freely. However, the project manager could also alter them before distribution. Even though such alteration would have to obey the access restrictions implied by ownership of objects, this may still be more than the department manager desires. In such a case the department manager could transfer ownership of only the *delegation_role*. This would allow the project manager to add this role to other user's roles or user structures, without being able to alter the encapsulated permission.

As an example of delegation at the access control system level, consider a situation where a new member wishes to join a club and requires two recommendations from existing members. For instance, we may have roles *candidate* and *member*. A user with a role *candidate* is considered to be applying for membership.

```
member := role
authorised candidate in user
user := user - candidate
    ...
end_role
candidate^.owners := member
member^.owners := member
member^.quorum := 2
```

Recall that the use of the '^' character with the name of a structure refers to the ownership of that structure. The constraint section of role *member* checks that the prospective member is a candidate and then removes the role *candidate* from that user's set of roles. The other statements allow any two members to approve a new member.

5.6 Joint action based policies

Joint action based policies [19] are used in situations where trust in individuals needs to be dispersed. Often this arises due to the fact that individuals are trusted according to their expertise which in turn maps the concept of trust to a specific set of actions. In delegation, there is a partial or complete granting of privileges, whereas in joint actions agents may acquire privileges, by working together in tandem, which none posses in isolation. For instance, consider the following examples:

Admission of a patient: a patient is admitted to the hospital if the patient and a doctor agree. The doctor and the patient jointly own a patient's record. Every doctor and patient has the following permission and privileges.

```
patient_record_permission := permission
    patient_record
    owner
    doctor_id* = 0, patient_id* = 0 : userid
    privileges {admit_privilege, ...}
end_permission
admit_privilege := privilege
    doctor_id <> 0 and patient in user and user <> doctor_id
    or
    patient_id <> 0 and doctor in user and user <> patient_id
    always
    if doctor in user then
      doctor_id := user
    else patient_id := user
    {admit}
end_privilege
```

The above example depends on ownership specified in the *patient_record* as only specific members of the roles *patient* and *doctor* could act on specific patient records. Note that the first attempt to admit the patient, by either the doctor or the patient, will fail; the access control system cannot predict future access. If actions are assumed to be sequential, then the first attempt must fail.

Authorising payment for goods: any member of the role *buyer* and any member of the role *accountant* can authorise the payment. The two roles are assumed to be mutually exclusive.

```
payment_record := permission
    payment
    authorising_buyer* = 0, authorising_accountant* = 0 : userid
    privilege {authorise_privilege, ...}
end_permission
authorise_privilege := privilege
    authorising_buyer <> 0 and accountantin user
    or
    authorising_accountant <> 0 and buyer in user
    always
    if buyer in user then
      authorising_buyer := user
    else if accountant in user then
      authorising_accountant := user
{authorise}
end_privilege
```

5.7 Limiting number of accesses

Sometimes one user will wish to give access to another user, but limit that access to a certain number of operations. Such situations can be handled in Tower as follows. User a wishes to give user b access to method m of object o, but wishes to impose a maximum number of times (say five) that b may call m.

```
counting_permission := permission
  o_class
  object o
  count = 5 : integer
  privileges {m_privilege}
end_permission
m_privilege := privilege
  count > 0
  count := count - 1
  {m}
end_privilege
```

If the user wished to have more than one privilege, then count would apply to all calls on the object. If the requirement was to limit the number of calls to each method individually, then a separate variable would be required for each method.

If the limit was to be over a number of objects, then the variable could be declared within a block and used within the permission for each object; which would also have to be declared and initialised within the block.

As the count variables can be updated within the action parts of the permission, the exact limiting of the access can be quite flexible.

6 Brief comparison with other work

The language described above is far from the first attempt at expressing role based access control. Other proposals have been put forward which allow the access control policies to be expressed in a systematic manner for role based or related systems. These proposals range from the formal one such as in [6] to more practical ones such as in [5] and [11], to related mechanisms such as in [13]. While formal languages such as the ASL in [6] can have good expressive power, they suffer from a resistance amongst real users due to their highly intricate nature. For example, these languages often depend upon their users having a reasonable level of understanding of logical principles. This is not always found amongst real world users, even those entrusted with the management of access control policies for a system. The syntax will often contain symbols not commonly used, limiting their appeal. While it is true that such languages are not generally written for widespread use, perhaps this simply strengthens the argument that a different approach should be used for the expression of access control policies in real world systems. Another drawback of some such proposals is the attempt to be too general: while it may be useful in a theoretical language to be able to cover a number of access control approaches, in the real world it is more important to be able to address those that are used in practice and to develop tools tailored to support them. The language described in this paper is intended to address this practical issue, and does not assume an overly high level of theoretical ability.

Furthermore, in the language that has been proposed in this paper we have considered meta-variables and role constraints, which few of the other proposals have included. For example, [5] attempts to use an assortment of predefined functions to fulfil the functionality that we address using meta-variables. A set of predefined functions

is highly unlikely to present a sufficient degree of flexibility and capability. It is far better to provide the user who needs to specify access control policies with flexible mechanisms (such as those we provide) and allow them to build structures for expressing their policies. Finally, some other earlier work such as [13] is limited in its expressiveness: it does not address concepts such as time or object attributes; and the syntax is also somewhat limited, being targeted at specific operating systems. Other work, such as [15] considers the use of RBAC structures in the management of RBAC policies. We believe that Tower can be used in this manner, but leave the detailed exposition of this to a future paper.

7 Concluding remarks

We have proposed a language based approach to the specification of authorisation policies. We believe that such an approach is required to support the range of access control policies in commercial systems. We have discussed the issues involved in the design of a language for role based access control systems. The proposed language focuses in particular on object-oriented systems. The notion of roles is used as a primitive construct within the language. It is often the flexibility and management of the meta-level operations which are significant when it comes to the applicability of an access control system to practical real situations. The use of a language based policy approach helps us to better structure such meta-level policies. We have described the basic constructs of the language, and used the language to specify several access control policies. In particular, we have described policy example scenarios involving role hierarchy, separation of duties both static and dynamic, Chinese Wall policy delegation, and joint action based access policies.

The implementation of Tower is in its early stages. We found implementation on top of other access control list mechanisms to be somewhat inefficient. Hence we have chosen to implement it directly. The chosen vehicle is based on the CORBA interceptor mechanism [20]. This allows the access control to be independent of the rest of the system, while still being able to allow or deny access. The implementations in each ORB can communicate, thereby allowing distributed access control. However additional implementation issues arise when providing RBAC management in a distributed environment. These will be reported when the implementation is completed.

8 Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments.

9 References

1 PFLEEGER, C.P.: 'Security in computing', (Prentice Hall, 1997), 2nd edn.
2 SANDHU, R., COYNE, E.J., and FEINSTEIN, H.L.: 'Role based access control models', *Computer*, 1996, **29**, (2), pp. 38–47
3 OSBORNE, S.: 'Mandatory access control and role-based access control revisited'. Proceedings of the 2nd ACM RBAC Workshop, Fairfax, VA, USA, 1997, pp. 31–40
4 SANDHU, R., and FEINSTEIN, H.: 'A three tier architecture for role-based access control'. Proceedings of the 17th national computer security conference, Baltimore, MD, USA, 1994, pp. 34–46
5 SIMON, R., and ZURKO, M.: 'Separation of duty in role-based environments'. Proceedings of the 10th computer security foundations workshop, Rockport, MA, USA, 1997, IEEE CS Press, pp. 183–94
6 JAJODIA, S., SMARATI, P., and SUBRAHMANIAN, V.: 'A logical language for expressing authorizations'. Proceedings of the IEE Symposium on *Security and information privacy*, 1997, Oakland, CA, USA, pp. 31–42
7 SANDHU, R., COYNE, E., FEINSTEIN, H., and YOUMAN, C.: 'Role-based access control: A multi-dimensional view'. 10th Annual computer security applications conference, Orlando, FL, USA, 1994, IEEE CS Press, pp. 54–61
8 HILCHENBACH, B.: 'Observations on the real-world implementation of role-based access control'. Proceedings of the 20th National information systems security conference, 1997, Baltimore, MD, USA, pp. 341–52
9 ZURKO, M., SIMON, R., and SANFILIPPO, T.: 'A user-centered, modular authorization service built on an RBAC foundation'. Proceedings of the IEEE symposium on *Security and privacy*, 1999, Oakland, CA, USA, pp. 57–71
10 BAI, Y., and VARADHARAJAN, V.: 'A logic for state transformations in authorization policies'. Proceedings of the 10th IEEE computer security foundations workshop, 1997, Rockport, MA, USA, IEEE, Computer Society Press, pp. 173–183
11 VARADHARAJAN, V., CRALL, C., and PATO, J.: 'Authorization for enterprise wide distributed systems: Design and application'. Proceedings of the IEEE computer security applications conference, ACSAC'98, 1998, Scottsdale, AZ, USA
12 MOFFETT, J.: 'Control principles and role hierarchies'. Proceedings of the 3rd ACM Workshop on *Role based access control*, 1998, Fairfax, VA, USA
13 KARGER, P.: 'Implementing commercial data integrity with secure capabilities'. Proceedings of the IEEE Symposium on *Security and privacy*, 1988, Oakland, CA, USA, pp. 130–39
14 FERRAIOLO, D., and KUHN, R.: 'Role based access controls'. Proceedings of the 15th NIST-NCSC national computer security conference, 1992, Baltimore, MD, USA
15 SANDHU, R.: 'Role activation hierarchies'. Proceedings of the 3rd ACM RBAC Workshop on *Role based access control*, 1998, Fairfax, VA, USA, pp. 33–40
16 GIURI, L., and IGLIO, P.: 'Role templates for content-based access control'. Proceedings of the 2nd ACM Workshop on *Role based access control*, 1997, Fairfax, VA, USA
17 BREWER, D., and NASH, M.: 'The Chinese Wall security policy', Proceedings of the IEEE Symposium on *Security and privacy*, 1989, pp. 206–214, Las Alamitos, CA, USA
18 VARADHARAJAN, V., ALLEN, P., and BLACK, S.: 'Analysis of proxy problem in distributed systems', Proceedings of the IEEE Symposium on *Security and privacy*, 1991, Las Alamitos, CA, USA
19 VARADHARAJAN, V., and ALLEN, P.: 'Joint action based authorization schemes', *ACM 30, Oper. Syst. Rev.*, 1996, 3, pp. 32–45
20 Object Management Group (OMG), 'CORBAservices: Common object services specification' and 'security services in common object request broker architecture', 1996–98.