An Objectbase Schema Evolution approach to Windows NT Security

K. Barker
Advanced Database Systems and Applications Laboratory
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada

Raj Jayaplan, R. Peters
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

Abstract

The current security model for the Windows NT operating system is powerful and offers many valuable features. The *User Manager* provided by Windows NT is the primary method for the provision of security maintenance. Unfortunately, this tool does not offer several features that would make the end-user's task more intuitive. This paper demonstrates a new technique to support the security on a Windows NT platform. Our system supports at least the following features: (1) An object-oriented hierarchy, so roles and groups can be supported in a more automated way. (2) A more intuitive user interface so the administrative errors are less likely to be problematic. (3) Simplified security management on a Windows NT platform. (4) Avoids unnecessary creation of objects (users / group) and redundant granting / revoking of privileges. This paper discusses a new security model that has these features in addition to those currently available on Windows NT.

1. Introduction

One of the most challenging problems in managing large networked systems is the complexity of security administration. Some of the challenges of security administration are: depth of security, ease of access, sound management, protection of integrity, cost-effectiveness, secrecy and confidentiality of key software systems, databases, and data networks. Most of the security models currently used require a trade-off between the depth of security and ease of maintenance. The dilemma is that the more secure a system becomes, the more of a barrier that security becomes to the normal operations for which it is intended. Thus, a security model should be designed in such a way that it is transparent to the users. Further it must be easy to maintain and manage even if a very complex security model is required to ensure its proper functions.

The primary goal of our model is to provide a flexible set of operations that will make security management easier and more understandable. This research uses an Objectbase Management System (OBMS) because of its ability to handle the complex information with complex relationships often found in non-trivial security models. These models are often characterized by their dynamics. In other words, once a security model has been deployed, changing system and application requirements often demand that the model adapt. Fortunately, substantial research has been undertaken in recent years describing how to manage the dynamics exhibited by object-based systems. Ideally these changes to the model should occur while the system continues normal operation, as it is often undesirable or even impossible to stop the system to deploy new security policies. Our model proposes the use of *dynamic schema evolution*, which plays a vital role in object base management systems because of its ability to make changes to the database schema while applications are running. Typical changes that may be required are to the domain structure, the functionality of a particular application or to meet new performance requirements. This paper describes a security management model based on well-known schema evolution techniques from OBMSs [1].

The paper also contributes by demonstrating how these theoretical techniques can be applied to Windows NT. Our system implements the object oriented schema evolution strategy on the NT operating system as a way of demonstrating its correctness and utility.

2. Fundamentals

Three fundamental aspects need to be considered before we can turn our attention to the specifics of our research. The first of these is the work on schema evolution in object-oriented systems with particular focus on an *axiomatic model*. Secondly, work directly related to non-discretionary access rules that are often captured in *roles*. Finally, it is useful to consider other work attempting to provide a "new" interface to an existing systems security model. Each of these is briefly discussed below.

A few groups have undertaken schema evolution research but in the interest of space we will focus only on the one that is directly related to this paper. Peters and Özsu [5] describe a sound and complete axiomatic model for dynamic schema evolution in object-based systems that support the key features of types and inheritance. The model can infer all schema relationships from two sets associated with each type. These sets are the known as the *essential supertypes* and the *essential properties*. Formal definitions for these sets is beyond the scope of this paper but we will provide an intuitive definition. Essential supertypes are those supertypes in the class hierarchy that must be included in the definition of a type, while the essential properties are those properties that cannot be dropped as schema changes are made. We will return to these concepts in more detail later but the interested reader can find detailed information in Peters and Özsu [1]. This work also describes various dynamic schema policies used by TIGUKAT to support evolution and how these policies can be defined using axioms.

The second key foundation for this research is that of *role based access control* [6,7,8]. Ferrialo and Kuhn [3] describe a non-discretionary access control mechanism known as role based access control suitable for the needs of non-military systems. Ferrialo and Kuhn [4] argue that access control decisions are often based on the roles individual users adopt in the organization. Therefore, a role specifies a set of transactions that a user (or set of users) can perform within the context of that role in an organization.

Finally, we turn our attention to the problem of retrofitting a security interface on an existing system in the way similar to that proposed in this paper. Hua and Osborn [9] provide an interface between the role based access control and UNIX. A model of how to access UNIX files using the role based access control is also described. A *role graph* is used to visualize the permissions granted to the files in the UNIX system. However, to completely model the existing permissions in a UNIX environment, the system file permission and the links between the files must still be modeled for them to complete their research.

3. The Axiomatic Model

This section briefly reviews the relevant details of the axiomatic model used in this paper. We define schema evolution as the timely change of the schema and the consistent management of these changes. *Dynamic schema evolution* (DSE) is the management of schema changes while a system is in operation. The axiomatic model has been demonstrated to provide a method to support dynamic schema evolution in the objectbased system by serving as a common, formal underlying foundation for describing evolution in existing systems [1]. This suggests that we should be able to apply it to other systems that exhibit similar characteristics but before demonstrating that this is correct we must first define some key terms in the axiomatic model.

Type τ: Type in the axiomatic model defines the properties of objects. Types are used as templates for creating objects. An element of type τ is denoted as *t*

Type Lattice: The type lattice can be represented with a directed acyclic graph where the types are the graph's vertices and sub-type relationships are captured as directed edges.

Immediate supertype P(t): The immediate supertype of type *t* are those types that cannot be reached from *t*, transitively, through some other type.

Essential supertype P_e (t): Essential supertypes of a type *t* are those types that are essential in the construction and existence of type *t*.

Supertype Lattice L_t : Supertype lattice of type *t* is a set that includes *t* together with all the supertypes (immediate, essential or otherwise).

Native Properties N(t): of type *t* are those that are not defined in any of the *t*'s supertypes.

Inherited Properties H(t): of type *t* is the union of the properties of all its supertypes.

Essential Properties $N_e(t)$: are those properties identified as being essential to the construction and existence of type *t*.

Interface I(t): of a type *t* is the union of native and inherited properties of type *t*.

Now we consider the basic operations common to schema evolution and security maintenance. Details and examples of each of these operations are available elsewhere [1] so we only provide the formal specifications in this paper.

Add a type: this operation adds a new type and integrates it with the existing lattice. The result of creating a new type *t* as the subtype of types $s_1, s_2 \ldots s_n$ with properties $P_1 \ldots P_m$ adds $s_1, s_2 \ldots s_n$ to $P_e(t)$, $P_1 \ldots P_m$ to $N_e(t)$ and the sets P(t), H(t), N(t), and I(t) are derived. If no supertypes are specified then T_object[1] is assumed.

Drop a Type: Removes a type from the schema. When a type is dropped it is removed from the P_e of all the subtypes of *t*.

Add Subtype Relationship: This operation adds a type as an essential supertype of another type, which effectively adds a subtype relationship between the two types. To add *s* as a supertype of *t*, *s* is added to $P_e(t)$ and the sets P(t), H(t), N(t) and I(t) are derived.

Drop a subtype Relationship: Removes a type as an essential supertype of another type, which effectively drops a subtype relationship between the two types. To drop type s as a supertype of *t*, *s* is removed from $P_e(t)$ and the sets P(t), H(t), N(t) and I(t) are derived. T_object cannot be removed as it is always essential.

Add a Property: Adds a property as an essential component of a type. To add a property *P* to type *t*, *P* is added to $N_e(t)$ and the sets N(t), H(t) and I(t) are derived.

Drop a Property: This operation drops a property as an essential component of a type. To drop a property *P* from type *t*, *P* is removed from $N_e(t)$ and the sets N(t), H(t) and I(t) are derived. Note that *P* is not removed from the interface of t because *P* may be inherited from one or more supertypes of *t*. However, if eventually the links to all supertypes that have *P* are removed, then *P* is no longer be part of *t*.

[1] T_object is the root class in the object hierarchy of the TIGUKAT model upon which we base this work.

4. Windows NT Security Model

Windows NT's security model is flat and does not support any hierarchical structure, let alone an object-oriented one. NT supports their security model with the *User Manager* [10]. The NT models security features supported by the *User Manager* include:

- Add / Remove a Group
- Add / Remove a User
- Add /Remove a member of a group
- Add / Remove privileges of a group
- Add / Remove privileges of a User

4.1 Add / Remove a group

NT's *User Manager* is used to add new groups to the system. Newly created groups do not have any privileges or user rights. A list of members can then be added to the group who then inherit the corresponding rights and privileges. This means that each user must be added to each group in which it should have privileges. It would be preferable to add groups of users to the newly created group thereby easing the process of creating new classes of users. In effect a hierarchy of user groups would be extremely helpful in security management.

Conversely, when a group is removed all members lose their membership. If the group has privileges, its members will all lose them unless they are explicitly given to the member through the granting of direct privilege. Ideally privileges could be grouped and formed into a hierarchy so that by dropping a subgroup the users would lose only a subset of privileges while maintaining those granted by the "super" group.

4.2 Add / Remove a User

New users initially belong to the "Users" group but the *User Manager* can insert them into additional groups. Users are atomic in that NT does not support the concept of a "user hierarchy" so users are inserted into groups only. When users are removed, they are physically removed from the system. The removed user is extracted from any groups to which they belonged. Finally, it should be noted that when a user is deleted it is completely removed from the system so even adding an identically named "new" user does not restore the old one.

4.3 Add / Remove Member to the group

Group membership can be specified while creating the group or user[2], or at any time after the group is created. Once a user is added to a group, all group privileges are inherited. Groups are composed of an arbitrary number of users but cannot contain other groups. In short, a group is only composed of existing users.

When a member is removed from a group it loses all privileges it inherited from the group except those that were granted directly. For example, if a user is given a privilege 'P' explicitly (direct privilege) which is also inherited from ones of its groups, the removal of the group does not remove 'P' because of the direct privilege.

4.4 Add / Remove privilege from a User

Users can have privileges granted to them from the *User Manager*. These are known as *direct privileges*. If this privilege is revoked the user will lose the direct privilege only. In other words, if the user is a member of a group that holds the privilege, the user will not lose it completely. This case is very difficult to handle with NT's *User Manager* as we discuss in the next section.

4.5 Add / Remove Privilege of a group

Whenever a privilege is added to a group it is propagated automatically to all members. Unfortunately Windows NT does not provide a mechanism to view privileges inherited as a result of group membership. The *User Manager* only provides a list of direct user privileges. Therefore, changes to group privileges are not reflected when viewing the users in the *User Manager*. A list of users/groups with a particular privilege can be created but it is impossible to find a complete list of privileges held by a particular user.

Consider a scenario where we would like to remove one of a user's privileges. To accomplish this we would like to remove the direct privilege and the privilege from all groups to which the user belongs that have the privilege. Deleting the direct privilege is trivial and once completed the *User Manager* will correctly display the absence of the privilege. But what about the privileges inherited from the groups? NT is very robust in that deleting a privilege from a group will remove it from both the group and its members. Although this is correct, it is impossible to tell by simply looking at the user privileges if the privilege has been completely removed. If we identify all groups but one containing the privilege, it will remain and be undetectable.

[2] Users can only be inserted at the time they are created if the group has already been created.

5. Modeling the Windows NT security with the Axiomatic model

The difficulties described above can be handled by treating users and groups as objects in an object-oriented hierarchy. We can then use the axiomatic model introduced earlier to manage the changes to these privileges. By treating these in a formal way we are able to ensure that the correct propagation of privileges occurs even if they are inherited in unexpected ways. Before discussing our implementation of this strategy we must first describe how Windows NT's security can be described with the axiomatic model.

5.1 Architecture Overview

Our security management system has three components, namely: the Windows NT layer, the axiomatic layer and the Security Manager Interface (see Figure 1).

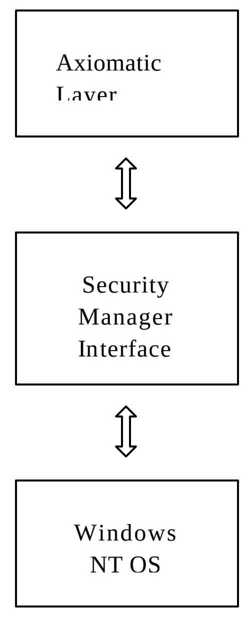

Figure 1: Security Management System Layers

The Security Manager Interface connects Windows NT's security system with the axiomatic model. Each component is discussed in the next few sections.

5.2 The Axiomatic Layer

Windows NT's security model is a flat structure. Users cannot be subtypes of others nor can a group be a subtype of another group. However, the same flat structure can be expressed with an axiomatic model thereby enabling both groups and users to be classified as types. The axiomatic model defines every type (users or groups) using subtypes and supertypes. Our system implements this object-oriented model with a file (external to NT's security system) that holds metadata about each of the types (users and groups). Therefore, the axiomatic layer is composed of two important components:

- The Axiomatic model
- The Metadata Axiom File (MAF)

NT user and group properties and their relationships are captured and expressed in terms of axiomatic properties. Any change to the subtype/supertype relationships or their properties result in corresponding change to the axiomatic model's properties. This means that all features supported by the axiomatic model such as: adding and dropping types, subtypes, supertypes and properties of a type are automatically supported by our security management system. Recall that Windows NT does not support these features because its security mechanism is not object-oriented.

The Metadata Axiom File is required because state information required by the axiomatic component is completely different from the structure of the traditional Windows NT security model. Thus the Metadata Axiom File holds all data required to build the axiomatic model of the NT defined users and groups.

Metadata stored in the MAF includes:

- All the types
- The subtypes of each type
- The supertype of each type
- Essential properties of all types
- Native properties contained in each type
- Inherited properties of all types

This metadata provides the information required to efficiently manage security on a NT machine. The Security Manager stores these properties in the MAF so all information about user and group state is saved. The axiomatic model for the user is dynamic so each time our Security Manager is loaded, the axiomatic model is built from the metadata in the MAF, thereby restoring the exact state of each of the objects.

Although Windows NT does not support the features provided by the axiomatic model, the Security Manager provides the various object-oriented features that enhances the security model.

5.3 The Security Manger Interface

The Security Manager Interface (Figure 1) ensures that all changes made by our tool are propagated to NT. It provides a translation from/ to Windows NT's security model and the axiomatic model. Since our model sees security privilege changes as schema evolution, updates to NT privileges are propagated as axioms to the axiomatic model. Conversely changes within the Security Manager are propagated to NT as privilege changes.

5.4 System Operation

Our security model is installed above the native security model of Windows NT. The axiomatic model is represented by a directed graph where a node represents each user or group and the subtype/supertype relationship is an edge. An edge from node *A* to node *B* indicates that the type *A* (user/group) is the supertype of type *B*. Type *B* inherits all properties of *A* as expected.

For any two nodes N_i, N_j , if N_i is contained in the interface of N_j, then there must be a path between N_i and N_j. The interface of a particular node *A* is a set that contains all the nodes that are supertypes of *A*.

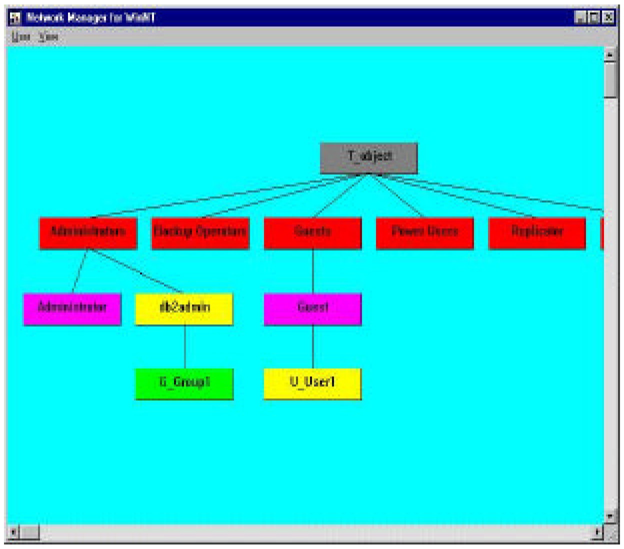

Figure 2: The Interface composed of groups and users

Figure 2 provides a snapshot of the object–oriented view of our security model for Windows NT. Type T_object is the root of all other types (both users and groups). This means type T_object has the minimum privilege and that all others inherit the privileges it possesses. The groups *Administrators, Power Users, Guests, Replicator* and *Users* are the immediate subtypes of T_Object. Since NT does not allow a group to be a member of another group, all groups are attached to T_object when the native security model is converted to the axiomatic model. The users *db2admin* and *Guest* are the members of the *Administrators* and *Guests* groups, respectively, so there is an edge from the type *Administrators* to the user *db2admin* and from *Guests* and *Guest*. For example, user *db2admin* and groups *G_Group1* and *Administrator* inherit all the properties of *Administrators*. Similarly the users *Guest* and *U_User1* inherit all the properties of the *Guests* group. To achieve greater clarity, different colors are given to System defined groups (Red), System defined users (Magenta), User defined groups (Green) and User defined Users (Yellow)[3].

We now summarize some of the axiomatic components defined for NT objects:

Type: NT objects are commonly referred to as types. This includes both users and groups.

Properties: The privileges associated with each group or user are termed *properties*.

Subtype and Supertype: Subtyping permits an object to be built based on another. For example, when a user is added as a member of a group, then we say that the user is a subtype of group or the group is the supertype of the user. Both users and groups are viewed as types so a user can be a subtype of another user (this prevents the unnecessary creation of groups in some cases), group can be a subtype of another group or a group can be a subtype of another user. By subtyping, an object (user or groups) inherits all the properties of the supertype. An object can have multiple supertypes and in that case it inherits the properties of all the supertypes[4].

Essential Supertype: The essential supertype of an object (user or group) contains all the users or groups that are essential to construct the object. All immediate supertypes are essential so every group is an essential supertype to its members.

Supertype Lattice: An object's type lattice contains the object and all its super-types.

We now turn our attention to the security management features supported by our security manager:

- Add / Remove a Group
- Add / Remove a User
- Add /Remove a sub-type (both users and groups)
- Add / Remove privileges of a group
- Add / Remove privileges of a User

Each of these is discussed in detail in the following sections.

5.4.1 Add / Remove Group

When a group is added, axiomatic metadata should be specified. This data might include the essential supertypes, immediate subtypes and the privileges the group possesses. Unlike NT's security model, ours allows the users or groups to be essential supertypes or immediate subtypes. In other words, both the users and groups can contain others. The group and its privileges are sent to the axiomatic component, which adds this group as a new type. Once the axiomatic model is updated, the affected NT objects (users/groups/privileges) are modified.

The insertion process requires that the group is created and all supertype privileges are given to this group. Thus, the group has inherited all the properties of its supertype. In cases where the properties of the supertypes overlap, only one copy of the properties are inherited thereby avoiding conflicts. These privileges must now be propagated to all its immediate subtypes. Two cases must be considered:

Case 1: If the immediate subtype is a user then the user is added to the group's membership roster. Once added to the group, NT propagates the group's privileges so there is no need to do this explicitly.

Case 2: If the immediate subtype is a group, then all of the group's privileges must be explicitly propagated to

[3] These appear as different shades of gray in this paper but we appeal to the readers imagination and intuition for the purpose of this submission. Demos of the system can be acquired by contacting the authors.

[4] This ability to subtype from both groups and users is extremely flexible. This expressive power is extremely useful but not all combinations of group/user subtyping will be required for NT. In fact, some may be irrelevant.

the subtype groups. This explicit propagation must be handled recursively to ensure that all children, grandchildren, etc. receive the necessary privileges.

Figure 3 (a) depicts groups *A, B, C, D, E* and *F* with a graphical depiction of the axiomatic model. Addition of group *X* as a supertype of *A* results in the propagation of *X*'s privileges (*P1, P2, P3, P4*) to the immediate subtype *A* and to groups *C, D, E* and *F* which is the behavior expected in an object-oriented inheritance hierarchy (see Figure 3 (b)).

Deleting a group removes the corresponding type from the axiomatic model. These changes are reflected to all its subtypes but the effect differs depending on the subtype's type (i.e. user or group). If the subtype is a user, it is removed from membership in this group. If it a group, the privilege is removed unless it is held as a direct privilege or inherited through another path. These changes are subsequently propagated recursively to all the affected types (groups and users) below this point in the hierarchy.

5.4.2 Add / Remove a User

Adding a user to our system results in the creation of the new user on NT. The user can be created with no supertypes (except *T_object*) or as a subtype of a group or user. Once again several cases need to be considered:

Case 1: Users without supertypes are subtypes of *T_object*. This is consistent with the axiomatic model because every type must be a subtype of *T_Object*.

Case 2: Users added as a subtype of a group (or many groups) are made members of each supertype group. In this way the user inherits all the privileges of the supertype, which is consistent with the object-oriented model.

Case 3: Users added as a subtype of another user is not available under the native Windows NT security model. This feature would prevent the unnecessary creation of groups in cases, where a user needs to possess all the properties of a different user. This operation would permit a user to inherit all privileges of a particular user. Additional research would need to be undertaken to define the semantics of removing a supertype of a user, but the issue is beyond the scope of this paper.

Our Privilege Propagation algorithm carries out this propagation for all cases.

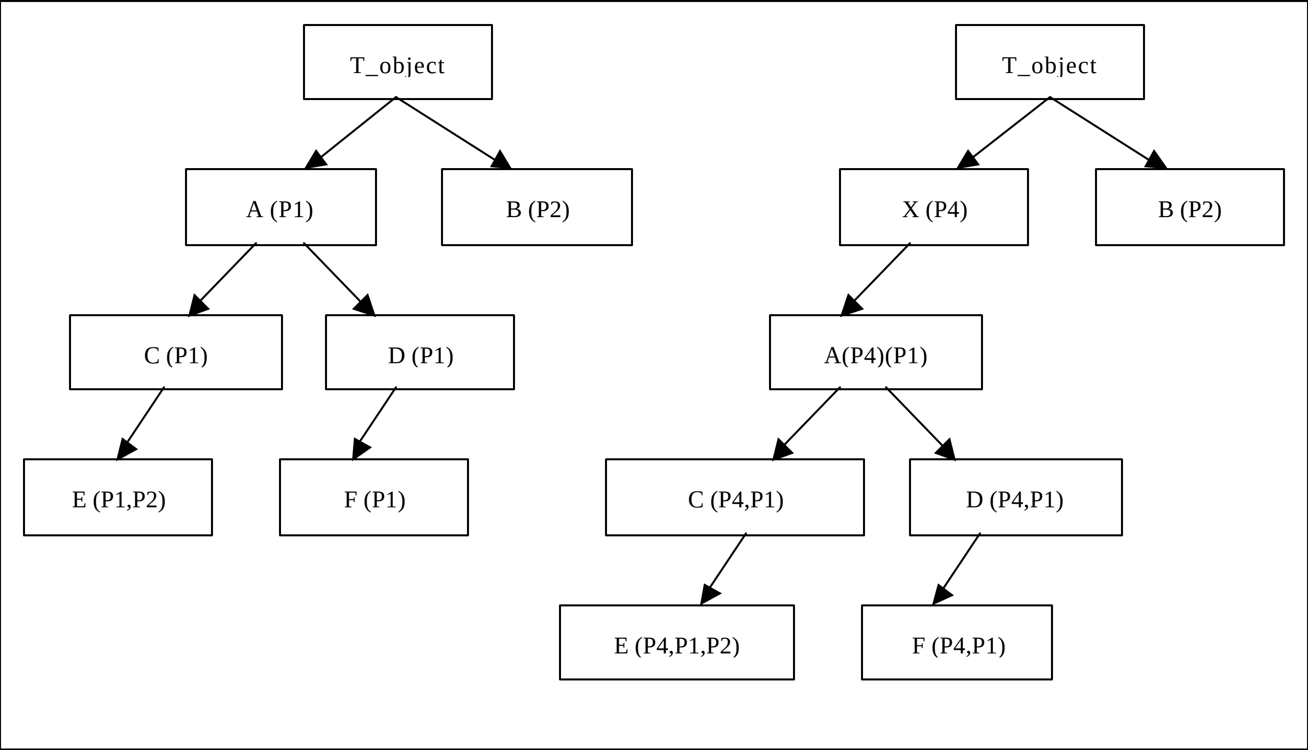

Figure 3: Addition of Groups (a) Before (b) After

If a user is deleted from the system, the axiomatic model is first updated and the corresponding effects are propagated to the NT objects. The Privilege Propagation algorithm (modified to revoke privilege) removes privilege from the object's subtypes. Finally, the user looses membership in the supertype's groups.

The algorithm used to add and remove groups and users is described in the Algorithm 1.

```
Algorithm 1: Addition or Removal of a group or user
1. Update the axiomatic components including
        Essential Super-type
        Immediate sub-type
        Type Lattice
2. If X is a User Then
        Delete / Create X as User
            If the operation is addition Then
            Add X as the member of all its super-type
            Endif
   Else
        Delete / Create X as a new group
        Revoke / Grant all the privileges of its super-types to X
   Endif
3. Initialize the groupList with all the sub-types of X
4. For each element in the groupList do
        If the element I is a user Then
            Remove / Add this user I as a member tothe group X
        Else
            revoke / grant the privilege of group X to this group I
        Endif
        For each sub-type of the group I do
            Add this sub-type to the groupList
        Endfor
        Remove the element I from the list
        If the groupList is empty Then
          Return
        Endif
   Endfor
End Algorithm 1
```

5.4.3 Modify Subtype Relationship

Addition (removal) of an object (user or group) as a supertype of another object results in the addition (removal) of the set of essential supertype for the object too. Our model permits the supertype to be either a user or group. As in previous cases, once the axiomatic model is updated the process of privilege propagation and addition/deletion of users as members is performed on the NT machine itself.

5.4.4 Add/Remove Privilege of a group

We now turn out attention to the specific issue of privilege management used in our system. Before presenting the details of our implementation we provide a few definitions[5]. Privileges in our model are broadly classified into two three types:

- *Native Privilege*: are directly given to the types
- *Inherited Privilege*: are acquired by the types from the parent (user or group)
- *Privilege Interface*: the union of the inherited and the native privileges

Our system clearly presents a list of each privilege held by an object as illustrated in Figure 4.

Figure 4 depicts user *U_User1* membership in Guests and that he holds the 'Shutdown the System' privilege. The inherited privilege 'Log on to local machine' would not appear as an NT privilege (privileges as seen using the *User Manager*) because these are not displayed.

The addition and removal of privileges is the final aspect of our system to consider. The key issues here are related to how privileges are propagated throughout the object hierarchy and how these are passed onto the flat structure found in NT. A brief discussion of the addition and removal of privileges is provided followed immediately by a sketch of the algorithm that implements this aspect of our system.

Add a Privilege: When a privilege is added to a type, it is added to the set of native privileges. The privilege must be propagated to its sub-types. The process of privilege propagation is similar for both users and groups. These privileges only need to be granted to the groups because NT propagates them to the group's members on our systems behalf.

Remove a Privilege: Removing a privilege from a user or group requires it first be removed from the set of direct privileges. In our system, if the privilege is found in the set of inherited privileges it has been acquired from at

[5] We have used these terms intuitively early in the paper but we need a more precise definition to describe this aspect of our implmentation.

least one of its parents. This means the privilege is not revoked from the Windows NT system and to do so would require that the corresponding privilege be removed from the parent. Even in the native NT model it is not possible to remove the inherited privilege. Only direct privileges can be deleted and this property is not changed in our model too. Alternatively, the object could be removed from the parent carrying the privilege (see Section 5.4.3). Once removed the privilege must be recursively applied to its subtypes.

The system has been implemented and proven to provide an excellent intuitive interface that meets the primary goals of our project. The system is sufficiently flexible that a system administrator can use our system to deploy new roles, groups and users but if they choose to use the *User Manager* to complete a task, our system will resynchronize with those changes once it is restarted. In this way, both our system and the one provided by NT can be used for complementary tasks. We believe however that our model is much more intuitive and it should be further investigated with the ultimately goal of deploying it as native to NT.

6. Conclusion

This paper describes a new security management model based on well-known schema evolution techniques in OBMSs. The model is successfully implemented on Windows NT above the original security model in such a way that it does not require modification or introduce conflicts to NT's current approach. We believe that one of the nicest features of our approach is that both our system and the *User Manager* can operate together. Any changes made with the *User Manager* are reflected in our system and changes done using our tool can be seen with the *User Manager* in precisely the way you would expect.

Several drawbacks associated with Windows NT's security model and its maintenance tool (the User Manager) are addressed by this research including:

- Failure to provide a clear link between inherited privileges arising from participation in groups and the lack of a technique to extract this information in an easy clear way.
- Inherited privileges of the members are hidden.
- Lack of easier mechanism to find a complete list of privileges held by a particular user/group.
- Lack of easier mechanism to find out the various groups to which each user belongs.
- An improved visual interface to the security model.

We have also demonstrated how our model permits the user to perform various operations that makes the management of security easier and more understandable. Some of the benefits of the new model include:

- ♦ Avoids unnecessary creation of groups.
- ♦ Prevents redundant and unnecessary granting and revoking of privileges.
- ♦ Provides a better visual interface that clearly illustrates the privilege flow.

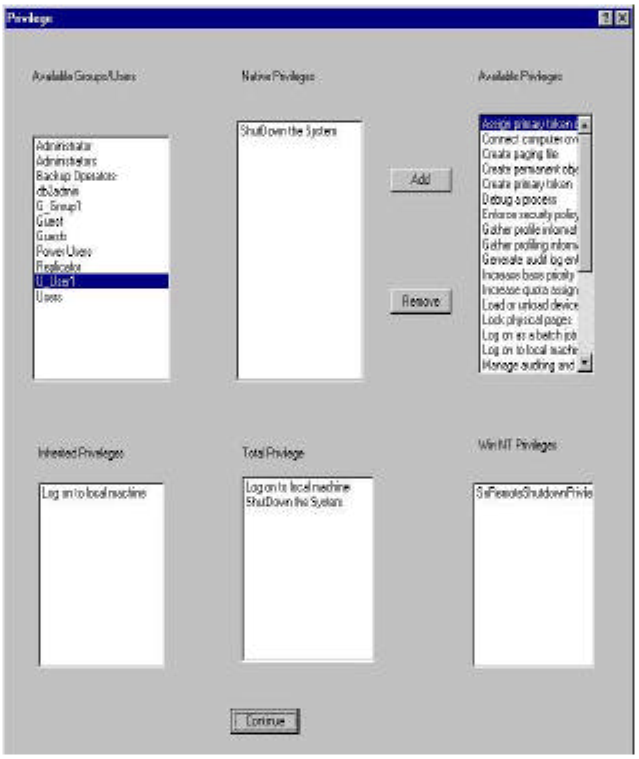

Figure 4: Interface Showing the Available Privileges

- ♦ Features of the object-oriented model that enhances the maintenance of the security model are used.
- ♦ Grants no more privilege than is necessary to perform a task. This property ensures the adherence to the security principle of least privilege.

References

[1] R. J. Peters and M. T. Özsu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Management Systems", ACM Transactions on Database Systems, 22(1): 75-114, March 1997.

Algorithm 2: Addition / removal of privileges

```
1. Add / Remove the privilege P from the set Native Privilege of type T
2. If privilege P not found in set Inherited Privilege of type T Then
      Add / Remove the privilege P from the type T (can be user or group)
      If T a group Then
          Add all the sub groups of the type T to the GroupList
      Else
          Add all the sub-types (both users and groups) of T to the GroupList
      Endif
      For each element I in the GroupList do
          Add / Remove the privilege P from the set Inherited Privilege
          If privilege P not found in set Direct Privilege Then
              Add / Remove the privilege P from the type T (can be user or group)
              If T a group Then
                  Add all the sub groups of the type T to the GroupList
              Else
                  Add all the sub-types (both users and groups) of T to the GroupList
              Endif
          Endif
          Remove the element I from the list
          If the groupList is empty Then return
      Endfor
  Endif
EndAlgorithm 2
```

[2] R. J. Peters. TIGUKAT: A Uniform Behavioral Objectbase Management System. Ph.D thesis. University of Alberta. TR94 – 06. April 1994

[3] D. Ferrialo and R. Kuhn. Role Based Access Control. 15th National Computer Security Conference. 1992.

[4] D. Ferrialo, A. Cugini, and R. Kuhn. Role Based Access Control : Features and Motivation . Computer Security Application Conference. 1995.

[5] R.J.Peters and M.T.Özsu. Axiomatization of Dynamic Schema Evolution in Objectbases, In 11th International Conference on Data Engineering, Taiwan, March 1995.

[6] J.Barkley. Implementing Role Based Access Control Using Object Technology. First ACM Workshop on Role Based Access Control. November 1995.

[7] J.Barkley. Comparing Simple role Based Access Control Models and Access Control Lists. Second ACM Workshop on Role Based Access Control. August 1997.

[8] J.Barkley and A.Cincotta. Managing Role/Permission Relationships Using Object Access Types. Third ACM Workshop on Role Based Access Control. July 1998.

[9] L.Hua and S.Osborn. Modeling UNIX access control with a Role Graph. In the Proceedings of the Ninth International Conference on Computing and Information. Pages 131 –138. June 1998.

[10] M.Minasi, C.Anderson, E.Creegan. Mastering Windows NT Server. Fourth Edition. 1997.