A Discretionary Access Control Policy for the Process Handbook

by

Calvin Yuen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirement of the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 1997
[June 1997]

Author _____
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by _____
Professor Thomas W. Malone
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Discretionary Access Control Policy for the Process Handbook

by

Calvin M. Yuen

Submitted to the

Department of Electrical Engineering and Computer Science

May 23, 1997

In Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In the transition of the Process Handbook[1] from a single user application to a multi-user server, the issue of controlling the user access privileges has to be addressed. This thesis aims at establishing an efficient and flexible access control policy to regulate the usage as well as to increase flexibility for collaboration in the multi-user Handbook. In this thesis, a proposed access control policy, within the Process Handbook object API abstraction level, will be identified. Moreover, desirable changes in the server to complement the new access control policy will be suggested.

As a result of the research, groups of users can more easily control the design of processes relationships. While new forms of collaboration are made possible, users can view the entities stored in the Process Handbook from new perspectives. Moreover, unexpected alterations of information stored in the system can be reduced.

Thesis Supervisor : *Thomas W. Malone*

Supervisor's Title : *Patrick J. McGovern Professor of Information Systems*

---

[1] Process Handbook is developed at the Center for Coordination Science in the Process Handbook Project

Table of Contents

Table of Figures

Chapter 1: Introduction

The Process Handbook developed at the Center of Coordination Science has reached a stage to move from a single-user application to a multi-user collaborative system. While the range of functionality has increased substantially, numerous problems have evolved. In particular, there are the issues of access control, authentication, concurrency control, and user-interface semantic model. This research will attempt to address the specific issues relating to access control. Through the comparisons of different possible models of controlling access, the optimal policy, within the object API abstraction level, would be identified. Finally, any desirable changes in the server objects to complement the new access control policy would be suggested.

1.1 Motivation

Controlling access can provide the Process Handbook users with security and flexibility. As a result, proprietary and sensitive information in the Process Handbook can be protected. Moreover, information under construction can be made hidden from irrelevant users. Finally, users can incrementally disclose changes made to the handbook, taking advantage of new forms of collaboration available.

1.1.1 Protection and Hiding of Information

The presence of naïve or malicious users in the Process Handbook imposes serious threats to the stored information. Therefore, controlling the access of information becomes vital. In addition, with access control in place, information under construction can be hidden from users by granting read access to the relevant users alone.

In the Process Handbook, activities are organized into multiple hierarchies. In each hierarchy, inheritance allows the attributes of high level activities to be propagated down to lower levels. As modifying a high level activity affects all its descendants, the characteristics of high level activities should not be easily modifiable. Similarly, low level activities, especially in the specialization hierarchy, may be proprietary. As a result, such information should only be readable to a restricted set of users. Finally, processes in their initial specification stage, such as in the scratch areas, are often useful to their authors alone. Therefore, by hiding information under construction, users can benefit from a cleaner view of the information in the Process Handbook. Hence, because of the need to protect and to hide information in the Process Handbook, an access control policy is highly desired.

To further demonstrate the importance of controlling accesses in the Process Handbook, consider the situation where the access of information is unregulated. In such situation, a naïve or malicious user can not only modify information, but also delete it; hence, the operations of normal users in the handbook are seriously affected. Without a regulated environment, the normal users would not find the system able to satisfy their needs. Hence, the usefulness of the system would be tremendously affected. Again, the need for a secure regulated environment is definite.

1.1.2 Empowerment of Control

When multiple users are collaborating in the specification of a new process, they may prefer to delay exposing the new process to the public because many changes may occur before reaching its final version. In addition, an author of the process may want to collaborate with a different set of users during different stages. By extending the Process Handbook with a rich access control policy, such flexibility in controlling the development of information can be accomplished.

Development often comes in stages. By providing the flexibility of controlling the access of information, the Process Handbook provides a natural environment where users can control the development of processes by designating different access restrictions during different stages of the development process. For example, in stage I, an author may be trying out new ideas in his scratch area. Then, he may invite his partner company to review his design in stage II. Afterwards, the author may put the new information to test by requesting the comments of a restricted group of users. Finally, when fully matured, the information may be made open to the public in the final stage. With a rich access control model in the Process Handbook, the above scenario would become feasible. As a result, users can benefit from having a development environment tailored to their own needs.

Table 1: Design Objectives

Objective	*Description*
Flexibility	Empower users to control the development and maintenance of processes
Protection	Provide appropriate protection to different types of information
Efficiency	The time and space required for access control should be relatively small.
Scalability	As the number of users or the amount of information grows, efficiency should not be significantly affected.

1.2 Goal

With the integration of an access control policy, the Process Handbook should offer an increase in the flexibility of collaboration as well as a solid protection of the stored information. As a result of the increase in flexibility, users are empowered to flexibly control the development and maintenance of processes. Moreover, because of the increase in the level of protection, proprietary and sensitive information will not be available to non-authorized users. However, to be a secure information system; the Process handbook should also satisfy the following properties, as defined in "Information Technology Security Evaluation Criteria" [10]:

- *Confidentiality* : Protection of unauthorized access of information
- *Integrity* : Protection of unauthorized changes of information
- *Availability* : Protection of unauthorized withholding of information

Moreover, in the design of the access control mechanism, performance issues such as efficiency and scalability also arise. In particular, to be efficient, the time and space required for access control should be insignificant. As importantly, as the number of users and the amount of stored information increase, the efficiency of the system should not be significantly affected. Finally, the design of the control model should be simple for the ease of system maintenance. A summary of the design objectives is listed inTable 1.

1.3 Scope of Research

To fully protect information stored in the Process Handbook, other mechanisms are needed in addition to access control. In particular, an authentication scheme must be used to verify the identities of users. Moreover, the computer on which the Process Handbook resides must be protected from attacks. In a network environment, messages sent among computers can be intercepted, modified, or even replaced. Hence, a secure message exchange protocol would also be required.

In this research, most of the attention is directed towards the use of access control. Specifically, an access control mechanism will be implemented within the object API abstraction level. Moreover, this research will investigate the rules concerning how the access control policy regulates the inheritance of information, and what modifications are desirable in the API objects. Finally, mechanisms for establishing secure channels and for authentication will also be addressed.

Chapter 2: Background

In this chapter, the background of Process Handbook, and that of access control policies will be discussed. Moreover, readers will be presented with mechanisms available for implementing the high-level policies. Finally, to demonstrate how secure channels and server authentication can be done, a brief introduction to the Secure Socket Layer Protocol and the Message Digest algorithm will be presented

2.1 The Process Handbook

The Process Handbook is a system developed at the Center of Coordination Science to facilitate the redesigning and inventing of organizational processes. After collecting examples of how different organizations perform similar activities, the system could then present the relative advantages of alternative activities [1]. Consequently, users of the system can compare the activities currently in use in the their organizations, and inspect alternatives for improvement.

In the system, activities, and other entities including dependencies, resources, ports, as well as connectors, are interrelated with one another. Among activities, various kinds of relations are exhibited. The first one is specialization. In this relation, the super-class collects the common attributes of its sub-classes. Then, there is decomposition, which implies the description of activities in terms of its sub-activities. For example, the activity "deliver sub-component" can be decomposed into the activities "assemble sub-component", "package sub-component", and "ship sub-component." Next, activities can be dependent on one another, and this kind of interaction is represented by dependencies. In a dependency relation, activities may be dependent on each other because they are shared resources, because they have a producer/consumer relationship, or because they generate a simultaneity constraint [1]. On the other hand, dependencies can also be managed by activities, forming a dependency coordination relation. In any cases, the linkage among activities and dependencies is accomplished using ports. Finally, through connectors, ports can be connected with one another [20].

Figure 1 shows how the specialization and decomposition relations of activities. In this figure, "Selling a product" is decomposed into sub-activities including "Identify prospects" and "Inform prospects about product". Meanwhile, it is also specialized into more focused activities like "Direct mail sales" and "Retail storefront sales". These specialized activities automatically inherit the decomposition as well as dependencies from "Selling a product". Nevertheless, decomposition and other characteristics can be altered during specialization. For example, in "direct mail sales", the sub-activity "Obtain mailing lists" has been added to replace "Identify Prospects". Moreover, a new dependency is created to relate "Obtain Mailing lists" with "Inform prospects about products" in the decomposition hierarchy.

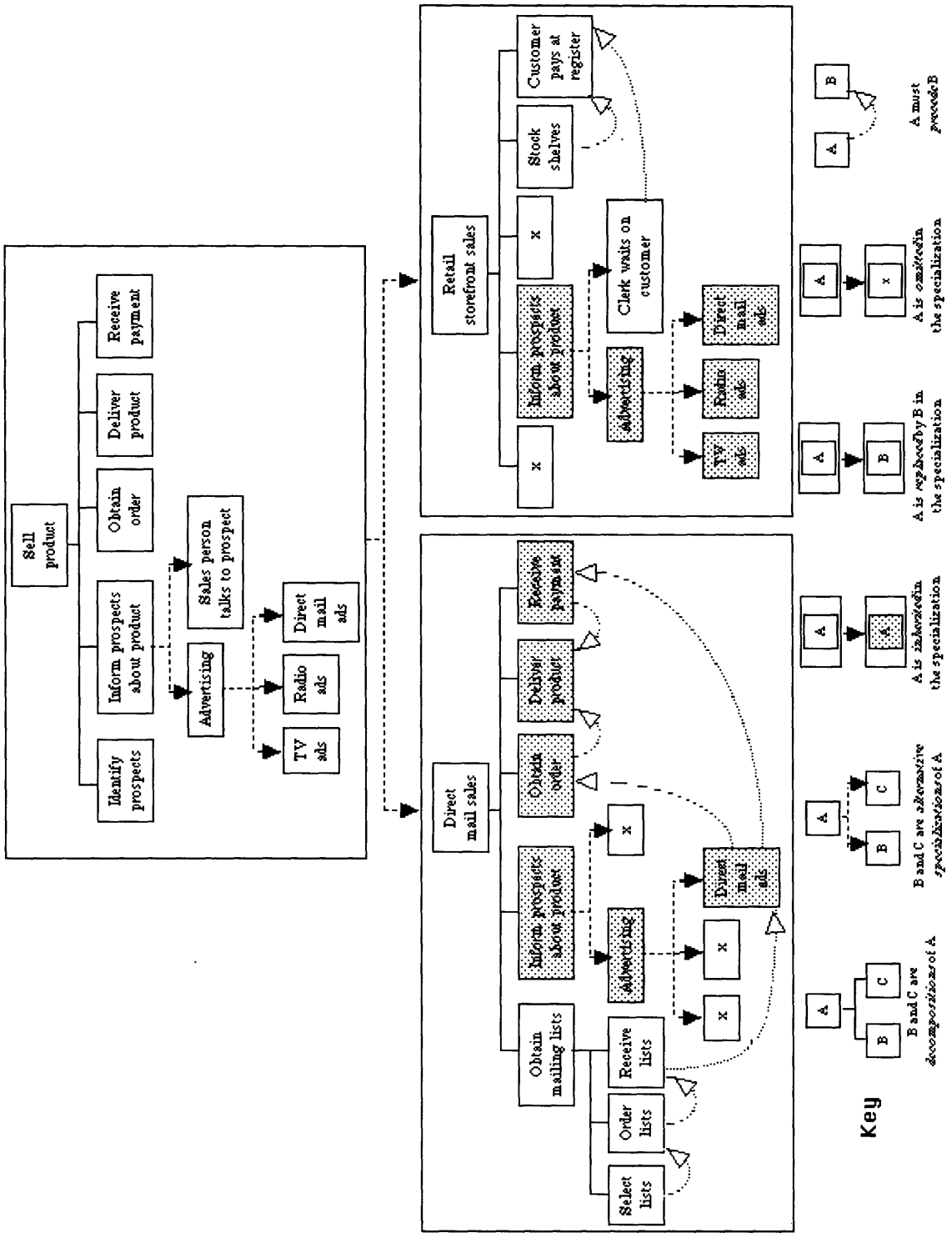


Figure 1: Interaction of specializations, decompositions, and dependencies. Adapted from (Malone, et. al, 1993)

While the Process Handbook is currently a single-user system, it is being extended to become a multi-user system. Nevertheless, in its transition to a multi-user system, numerous issues have to be addressed, as described in the previous section.

2.2 Access Control Policies

As stated by Brinkley and Schell [4], access control policies can be partitioned into two main categories: mandatory access control policy and discretionary access control policy. Each of them is described below:

In a mandatory access control policy, the sensitivity of objects and the users of the system are partitioned into access classes. When a user creates an object, the sensitivity of the object would be marked the same as that of the user. As a result, only users with the same or a more superior access class can access the object. In this way, the flow of information from one access class to another could be restricted.

To achieve the control over the flow of information, a mandatory access control policy must be both global and persistent [4]. In particular, when an access control policy is both global and persistent, all objects in the system would have the same level of sensitivity wherever they are and whenever it is. To achieve these properties, the set of access classes must form a partial order set [4]. In other words, for any pairs of access classes, class A and class B, they must fall into one of three cases: A is superior to class B, class B is superior to class A, or class A and B are not comparable.

Because the access class labels are attached to the objects in the system and that the labels are global and persistent, no one can pass information to users of an unauthorized access class. As a result, a mandatory access control policy provides a high assurance of protection to the information in the system, even in the face of Trojan horses and other forms of malicious software.

While the mandatory policy employs a simple strict rule to restrict access, given the characteristics of the Process Handbook, the discretionary access control policy provides a better match. (For further discussions, see Section 3.1) In a discretionary access control policy, the restriction on the flow of information is much less severe. Unlike a mandatory access policy where an object created by a user could only be viewed by others of the same access class, a discretionary access control policy would allow users to designate the permissions given to others in the system.

2.3 Access Control for Inherited Information

In an object-oriented system, security should be defined at the level of the object model [11]. While it is possible to provide security through lower level operations, this approach violates a basic principle for the design of secure system, i.e., access restrictions should be defined at the highest possible level where the semantics are explicit [13]. Therefore, regulations should be applied to the usage of defined object methods.

Moreover, the hierarchical structure of classes and sub-classes can be used to define implied access to avoid the proliferation of rules [11]. Fernandez, Larrondo-Petrie, and Gudes [11] propose the following rules for implied access in inheritance:

2.3.1 Rules for Generalization:

- Implied authorization: a user that is authorized to apply a given method to a class has the same right with respect to the corresponding inherited method in a subclass of that class.
- Class access: access to a complete class implies the right to apply all the methods defined in the class as well as methods inherited from higher classes.
- Visibility: the use of methods defined in a subclass is not implied by the right to use the methods in a super-class of this class. This also applies to the redefinition of inherited methods.
- Propagation control for specialization: the propagation of an inherited authorization for a method can be stopped by a rule specifying no access (negative authorization) to that specific method.

2.3.2 Rules for Aggregation:

- Propagation: access to a class only implies similar type of access for the components of the class if this type of access normally propagates to these components.
- Composite objects: access to all methods of a class implies the right to apply any methods in the component classes.
- Propagation control for aggregation: a negative authorization rule can stop the propagation of implied access in aggregation.

2.3.3 Rules for Relationships

- Relationship inheritance: access to a relationship or to some of its attributes can be inherited with the specific restrictions implied by the subclass.
- Propagation control for relationship: propagation of inheritance in generalization relationships can be stopped by using negative authorization rules.

2.3.1 Process-oriented Access Control

Unlike traditional object-oriented systems where the systems are organized around objects, the Process Handbook is composed of processes. Processes and objects differ in certain aspects; for example, while inheritance applies to only nouns in the object-oriented world, inheritance applies to both verbs and nouns in the process-oriented world. However, similar to objects, processes are interrelated with one another through specialization, decomposition, and other relationships.

Because of the similar types of inheritance relationship exhibited among objects and processes, the above ideas regarding the rules for regulating generalization, aggregation, and relationships can be heavily drawn upon so that the access control policy could take advantage of the inheritance relationship of processes. Moreover, it is clear that both the normal propagation of access, and the usage of negative authorization rules are applicable to the Process Handbook. In particular, as processes become increasingly specialized, they are more likely to become sensitive information for specific users. Therefore, the ability to view a high-level process in a hierarchy should not imply the ability to view its descendants. In other relationships, the author may desire similar types of control to restrict access. Therefore, the propagation control for specialization, decomposition, and relationships, together with the normal propagation rules, provide the flexibility to meet a wide range of requirements.

2.4 Implementation of Access Control Policy

Clark and Redell [8] suggest that the mechanisms that are practical for implementing access control policy fall into three categories: access control lists, capabilities, and their hybrid [8]. Each of them will be discussed in the following paragraphs.

2.4.1 Access control lists

In access control lists, each domain[2] in the system would maintain a list which identifies the permissions given to different users of the system. A typical access control list would be composed of two columns. While first column identifies the users, the second column includes the access privileges given to the corresponding users. Using the access control list mechanism, the system can easily keep track of the privileges given to users. However, the system has to search through the list during each access, making efficiency a concern.

In changing the content of an access control list, a system can either use the self-control approach or the hierarchical-control approach [9]. In the self-control approach, the user who created an object would be given the permission to modify its access control list. On the other hand, in the hierarchical-approach, the access control lists are organized in a hierarchical manner, and the administrator of the high level access control lists would also be permitted to modify the lists in lower levels. In this way, high level administrators can take care of low level lists in cases where the low level administrators are unavailable, or mistakes have been made. However, under the hierarchical scheme, the high level administrators would become too powerful. To create a balance of power, suggestions have been made to create an audit trail to keep track of access-control-list modifications. Another possible solution is to support a buddy system which allows modifications only if multiple authorized users have performed the same operations.

---

[2] Domain: A domain is made up of one or more objects in the system

2.4.2 Capabilities

In capabilities, each authorized user of a domain is given a capability which the user can use in subsequent access. A typical capability contains three fields: the type of the domain, the identifier of the domain, and an indicator of access privileges. For example, in accessing domain A, a user would present its corresponding capability. The system would then verify if the domain identifier on the capability is equal to A, and the privilege specified on the capability allows the requested operation. Using capabilities, the system can efficiently check user authorization, provided that the appropriate capability can be extracted from the user account efficiently during each access; otherwise, the system may need to search through a list of capabilities to find the appropriate match. In such cases, the performance of this method would not be strictly more efficient than that of using access control lists. Finally, keeping track or revoking capabilities given out to users is hard because the system has to search through all user accounts for capabilities which contain the matching domain identifier.

2.4.3 Hybrid Model

The hybrid model combines access control lists with capabilities. Like access control lists, each domain in the system would maintain a list which identifies the permissions given to different users of the system. However, when an authorized user first accesses a domain in a session, the user would receive a capability which corresponds to his access right. In subsequent access of the domain in the session, the system could take advantage of the capabilities given out to users rather than searching through the access control lists. In this way, authorization checking becomes efficient. Moreover, the access control list attached to each domain allows the system to keep track of user privileges. As a result, the hybrid mechanism provides both efficiency and accountability. However, as mentioned in the previous section, the claimed efficiency relies on the assumption that the appropriate capability can be extracted from the user account efficiently during each access.

In this research, the three mechanisms of access control would be compared by their ability to fulfill the goals listed in Section 1.2. Which mechanism is most appropriate for the Process Handbook would then be identified.

2.5 Secure Channels

To achieve confidentiality, integrity, and availability, secure channels have to be established between clients and servers. This section provides the background on Secure Socket Protocol (SSL), which aims at providing privacy and authentication between two communicating applications. [18]

2.5.1 SSL Handshake Protocol

The SSL Handshake Protocol has two major phases. The first phase is used to establish private communications. The second phase is used for client authentication.

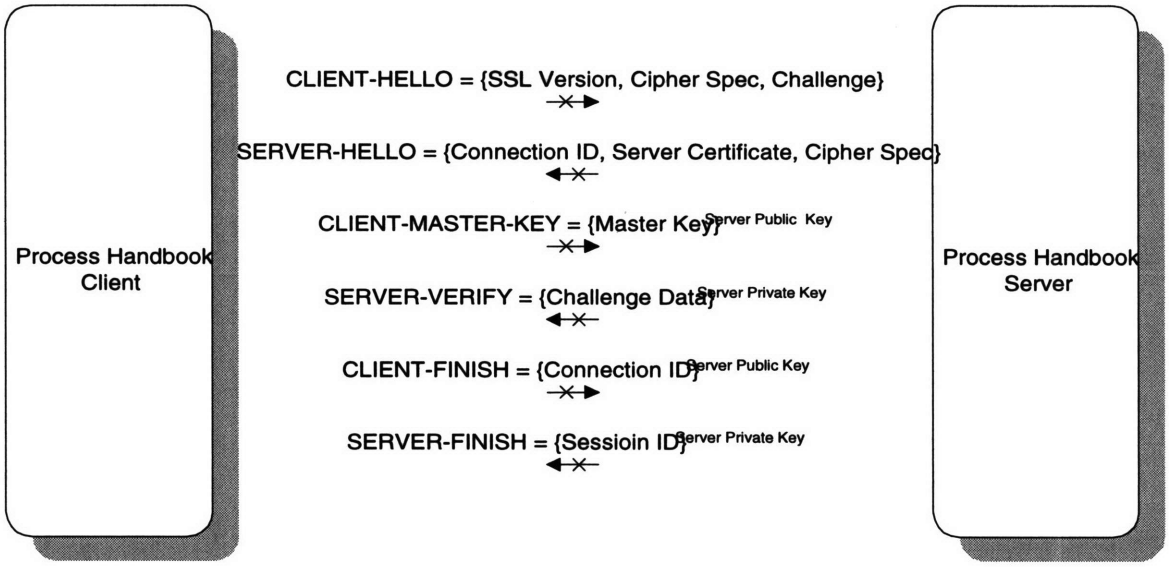


Figure 2: SSL handshaking for a new session with no client authentication. If a session identifier is already established, the CLIENT-MASTER-KEY message is not sent, and the CLIENT-HELLO message will include the established session identifier.

2.5.1.1 Phase 1

In Phase 1, the client initiates the protocol by sending the CLIENT-HELLO message[3]. After receiving and processing the message, the server responds with the SERVER-HELLO message. Depending on whether a session identifier has already been established, they would either continue in Phase 1 to establish the identifier, or enter into Phase 2.

To establish a new session, a new master key is needed. In producing the mater key, the client retrieves the necessary information from the SERVER-HELLO message[4]. After generating the new master key, the client sends a CLIENT-MASTER-KEY message which includes the new master key encrypted with the server's public key. Finally, the server returns a SERVER-VERIFY message to authenticate itself, returning the challenge data encrypted with the master key. If the challenge data is identical to the one in the CLIENT-HELLO message, the client is certain that the server is authentic, because only the server has the corresponding private key to decrypt the master key sent in the CLIENT-MASTER-KEY message.

---

[3] The CLIENT-HELLO message includes its SSL version, cipher specification, challenge data, and the session identifier. While the challenge data is used to authenticate the server, a positive session-identifier indicates that a session has already been established between client and server.

[4] The SERVER-HELLO message includes connection identifier, server certificate, and cipher specifications.

2.5.1.2 Phase 2

In Phase 2, the client is being authenticated. During client authentication, the client would be requested to produce certain authenticating information, such as a certificate. When authentication is done, both client and server will send a finish message[5], and the application protocol can start to operate. In all subsequent message exchanges, the application protocol will be operating on top of the SSL record protocol, which encrypts messages with the ciphers agreed upon during the handshaking protocol.

2.6 Message Digests

To protect authentication information from being exposed as plain text in the system, passwords and other related identifying information should be stored in a secure format. Such conversion can be achieved by the message digest and other hashing algorithms. In this section, the background of MD5, one of the most popular algorithms used to transform and compress passwords, is presented. The basic idea of the algorithm as well as the security issues related to MD5 will be discussed.

2.6.1 Description of MD5

The MD5 message digest algorithm takes as input a message of arbitrary length and produces as output a 128 bit "message digest" of the input [19]. The MD5 algorithm has a few basic steps. In the first step, the input is being padded so that its length is congruent to 448 mod 512. Afterwards, the least significant 64 bits of the original input will be appended so that the length of the resulting message will be exactly a multiple of 512. Then, the message will be transformed into a 128 bit message digest, making use of certain magic constants as well as specialized transformation tables.

While the MD5 algorithm is fast on 32-bit machines, it is also conjectured to be computationally infeasible to break. In other words, a computationally-bounded adversary cannot produce two different inputs with the same message digest with non-negligible probability. Furthermore, it is conjectured that the difficulty of coming up with such inputs is on the order of 2^{64} operations. More remarkably, the difficulty of coming up with a message having a given message digest is on the order of 2^{128} operations. Therefore, by creating message digests of the passwords used in the Process Handbook, the safety of users' authentication information can be greatly enhanced.

---

[5] The CLIENT-FINISH message confirms the connection identifier, and the SERVER-FINISH message confirms the established session identifier.

Chapter 3: Access control Policy Design

3.1 Towards a Discretionary Access Control Policy

In the design of the access control policy, one would realize the choice between a mandatory access control policy and a discretionary access control policy is obvious. Since the Process Handbook is a distributed system, information retrieved from the Handbook is loaded onto the screens of remote machines. In a distributed environment, while protecting the sensitivity labels of objects is feasible through cryptography, there is no way of preventing users from capturing and redistributing information stored in the objects. For example, the user may capture the content of a screen in a graphics file, and distribute the file to non-authorized users. In order to prevent this situation, the operations allowed in user machines must also be restricted. However, even if there are mechanisms which restrict the operations in the machines, what prevent the users from taking photographs of the screens? It becomes clear that mandatory policies under a distributed environment are not practical. Therefore, a discretionary access control policy should be used.

Moreover, the flexibility of discretionary access control policies matches the design goal of the system. Under a mandatory policy, users are divided into access classes. At a result, while it is possible to deny the access of an entire access class, this policy may not support the exclusion of an individual user. In the Process Handbook, it is often desirable to only permit access to the group of users currently participating in the collaborative design effort. Since the size of the group varies and the composition of the members changes, the fine-grained control offered by a discretionary policy is invaluable to the collaborative environment.

3.2 Access Rules for First-class Entities

In the current design, access control is enforced only for first-class entities[6] and their access control objects. While it is possible to regulate the retrieving of other objects[7], controlling the access of these objects alone is sufficient to protect the information contained in other objects. The reason is that information stored in other objects is useful only when its context is defined, i.e., when its associated first-class entity is known. Therefore, by restricting the access of first-class entities, the access of other objects can be controlled simultaneously[8]. In particular, by following a rule which

---

[6] First-class entities refer to instances of the Entity class, i.e., thing, activity, bundle, port, navigational node, dependency, resource, and attribute type.

[7] Other objects refer to instances of the Relation class and Attribute class. Here, we assume that it is sufficient for all attributes of an entity to share the same set of access control restrictions.

[8] On the other hand, placing access control on non-first-class entities alone do not imply simultaneous protection for first class entities.

requires the hiding of a relation unless the user has the privilege to access the binding entities, the system can filter relations which are not relevant or contain sensitive information. Moreover, the access control restrictions on the first-class entities can determine the restrictions for its associated attributes, assuming that it is sufficient for all attributes of an entity to share the same set of access control restrictions. Otherwise, attribute objects with their own access control restrictions would be desirable. Further discussion can be found in 3.2.1.1.

3.2.1 Classification of Privileges

From the access control perspective, the methods available in the Entity class can be classified into 7 categories: read, create, add relation, remove relation, full edit, move, and delete. Moreover, the level of authorization implied by the right to perform each of the 7 classified operations is strictly ordered. In particular, the level of authorization associated with the delete privilege is strictly higher than the move privilege, which in turn is strictly higher than all edit privileges. (For details, see Figure 3.)

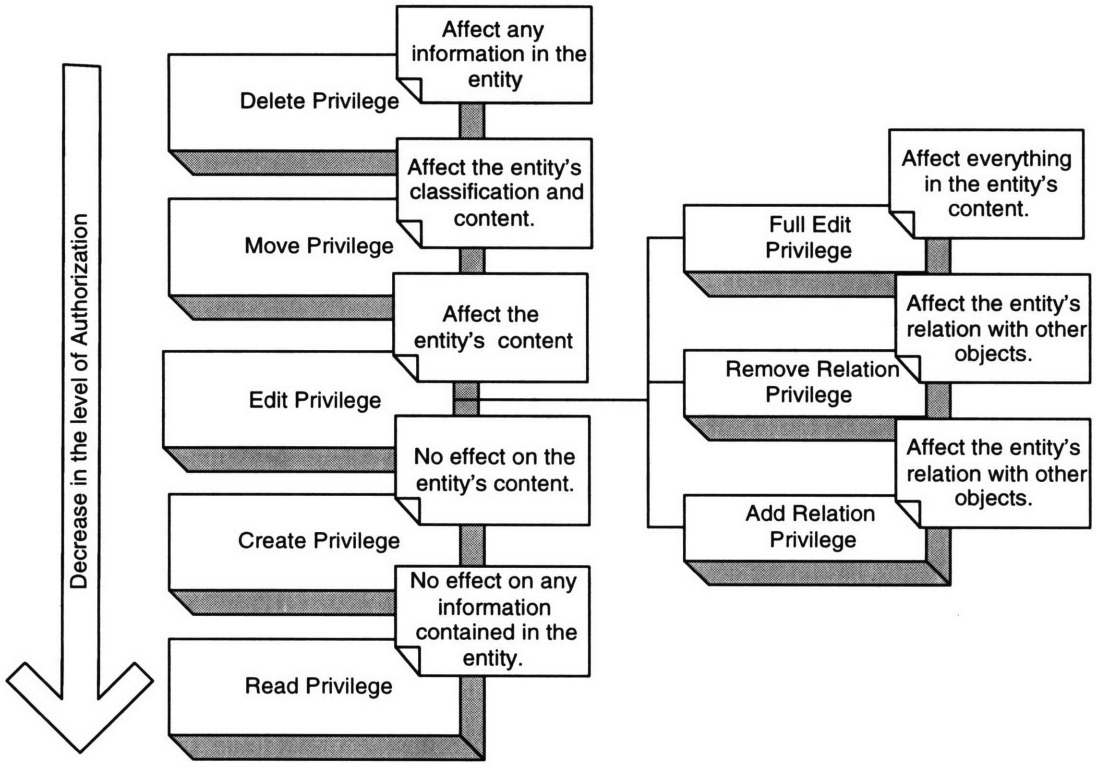


Figure 3: Authorization Levels of First-class Entities

3.2.1.1 Evolution of the Proposed Classification

The optimal classification of user privileges would provide not only adequate efficiency for the access checking mechanism, but also flexibility for access control management. In determining a balance between efficiency and flexibility, our investigation begins with a simple model where there

are only two kinds of privileges: read and write. Incrementally, the granularity of the access control is made finer and finer until any further partitioning of the categories would not be justified.

Looking at the simpler models, one would note that methods of rather different functionality are being grouped into a single category. For example, if there are only a read privilege and a write privilege, having the write privilege for an entity would imply at least the right to move as well as to delete. However, it is obvious that the author of an entity may want to allow some users to be able to move but not delete the entity. As a result, an increase in the flexibility of these models is highly desirable.

In the proposed model, each category corresponds to a single abstract operation of the Entity class. Therefore, in addition to read, edit, and delete, there are the categories of creating specialization and moving entities across specialization. In particular, creating specialization of an entity is different from editing the entity because the content of the entity is not altered. Moreover, moving an entity differs from editing an entity since moving the entity changes its classification in addition to its content. Finally, editing the content of an entity consists of not only modifying its attributes, but also adding and removing relations from the entity. Therefore, adding and removing relations are separated out in the edit operation. Hence, as each category of the proposed model corresponds to a single abstract operation, methods of different functionality would belong to different categories. Moreover, because of the similar behavior exhibited by the methods within each category[9], any further partitioning of the categories would not be justified.

Finally, the defined privileges can be ordered according to the impact of the corresponding operations on the entity. In particular, the read operation, which has no impact on the entity, should require the least privilege. Similarly, the create operation, which has no influence on the content of the entity, should require less privilege than any of the remaining operations. At the other end, the delete operation, which affects any information contained in the entity, should require the most privilege. Moreover, since changing the classification of an entity may modify the content of the entity, the move privilege should be immediately below delete. (For details, see Figure 3) As an important note, the remove relation category is placed higher than the add relation category. By making the removal of an existing relation more selective than the addition of a new relation, the gathering and the discussion of new ideas can be encouraged.

While attributes are not implemented as first-class entities in the current object API design, the advantage of having access control on attribute objects is foreseeable. By imposing access control restrictions on attribute objects, one can have different access control restrictions on different attributes. For example, while only a restricted set of editors can change the "Contact Information" attribute, almost all editors can change the "Comment" attribute. Following this design, having the add relation privilege will automatically allow a user to add new attributes to an

---

[9] Here, the assumption that it is sufficient for all attributes of an entity to share the same set of access control restrictions has been made.

entity. However, to remove an attribute, the user must have the remove relation privilege in the entity and the delete privilege in the associated attribute object. Finally, as in the current design, users with full edit privilege would be allowed to modify any attributes[10]. This can be accomplished by automatically granting the delete privilege for the associated attributes to users with full edit privilege for the entity.

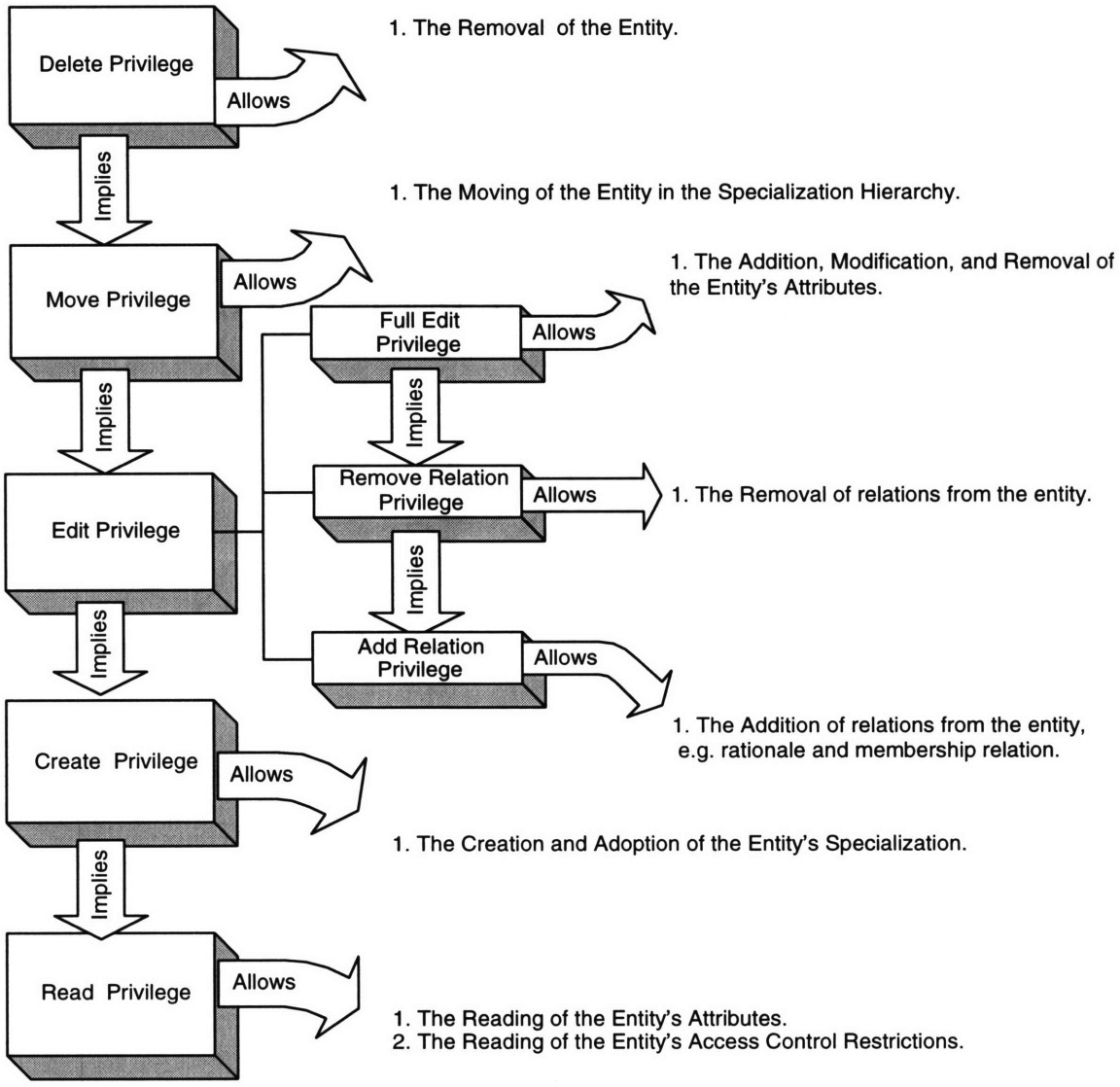


Figure 4: Summary of Access Privileges and Implications

[10] Except "Password" attributes.

3.2.2 Inheritance of Privileges

As discussed in Chapter 2.3.1, the idea of propagating access can be incorporated into our process-oriented system. However, because of the unique characteristics of the Process Handbook, the default inheritance rules differ from the ones proposed by Fernandez, Larrondo-Petrie, and Gudes [11]. In particular, in decomposition, access privileges are not inherited. The reason is that inheritance of access privileges in decomposition will result in massive generation of new specialized entities to accommodate changes made to shared entities' access control restrictions. Moreover, it is sometimes desirable to compose proprietary entities from public ones, or public entities from proprietary ones. Therefore, for the Process Handbook, access privilege inheritance in a decomposition hierarchy would not be appropriate. On the other hand, similar to the proposed model, all access privileges are inherited by default in specialization. However, to allow the creation of proprietary specialized entities, negative authorizations can be specified to indicate otherwise. In the following sections, the definitions and implications of the above classified operations will be discussed; a summary of the access control rules can be found in Figure 4.

3.2.3 Implication of Privileges

3.2.3.1 Implication of Read Rights

- The right to read a first-class entity implies the right to read its access control restrictions.
- The right to read a first-class entity implies the right to get its attributes.
- The right to get the associated first-class entities, such as those in the specialization, generalization, decomposition, where-used, or navigational relation, is given if and only if the right to read the entities is given.

Viewing the content of a first-class entity consists of retrieving not only its attributes, but also its access control restrictions. The reason is that access control restrictions can be thought of as attributes of the entities. Moreover, the ability to view the content of entities' access control restrictions is important to the collaboration environment because it encourages users to communicate with one another. For example, a user may want to contact and join a particular user group if he realizes the group has certain permissions on certain entities. Therefore, the ability to read an entity should imply the ability to view its access control restrictions.

As first-class entities are often closely related to one another, it may be desirable to enforce the inheritance of access control restrictions across relations. For example, the information stored in an activity is tied to those stored in its associated ports, sub-activities, dependency, dependency resource, and dependency managing activity. Therefore, as they complement one another, it may be convenient to the users if these entities would share the same set of access control restrictions. However, such rule tremendously decreases the flexibility of the system, as the constraints imposed would rule out interesting ways where processes can be built. In particular, it may sometimes be desirable to compose a proprietary activity from public activities. Similarly, it may also be valuable to build a public activity from proprietary processes. Therefore, to provide a complete hiding of sensitive information, the link to a sensitive entity should not be disclosed unless the user

has the permission to view the content of the entity. Hence, the ability to view a link should depend on the ability to read the entities at both ends of the link.

3.2.3.2 Implication of Create Rights

- The right to create for a first-class entity implies the right to create specializations of the entity and to adopt others' specializations.
- To adopt specializations for a given entity, a user must have the create privilege for the entity, as well as the move privilege for the entities to be adopted.
- All privileges are inherited by default in creating specializations. If otherwise specified, the access control restrictions of the specialized entities would be derived from their creators' preferences.
- In creating a new specialization entity, the new entity will first derive the set of users with the maximum management right from its creator's preference. If no preference is given, the new entity will derive the set of users with the maximum management right from the profiles of the set of selected context groups[11].

Since the level of authorization implied by the ability to create is greater than that implied by the ability to read, the possession of the create authorization for an entity implies the possession of its read authorization. In addition, having the create authorization permits the execution of all operations relevant to adding specialization links. The relevant operations include not only creating, but also adopting specializations.

In adopting specialization for a given entity, specialization links previously not exist will be created. Therefore, the create right must be present in order to adopt specializations. Moreover, in adopting specialization links, the content of the adopted entities as well as the structure of the specialization hierarchy would be modified. Hence, the user should also have the move privilege for the entities being adopted.

Furthermore, the more specialized an entity, the more sensitive or proprietary the information it contains. Therefore, the flexibility of overriding the default access inheritance mechanism is provided. As a result, while most users can view the content of general processes, the access of more specialized processes can be denied to the public.

Finally, when a user create a new entity, he can either directly input the set of users with the maximum management privilege, or select the user groups[12] which represent the context of his

---

[11] While there are multiple levels of management privileges, each entity must designate at least one user to have the highest level of management privilege. Since only users with the maximum management right are allowed to grant management privileges to other users, a newly created entity must be able to derive the set of users with such right. Otherwise, the entities could not be properly managed.

[12] The set of user groups to be selected must have a parent membership relation with the creator's authenticated group. Moreover, as the selected groups represent the context where the users perform the

create operation. In the latter case, the system would extract the set of users with the maximum management privilege from the profiles of the selected user groups. By storing the users with the maximum management right in the profiles of user groups, one can easily specify the same set of users with the maximum management privilege for related entities.

3.2.3.3 Implication of Add Relation Rights

- The add relation privilege for an entity implies the right to add relations from the entity.

The add relation privilege for an entity implies the right to read and create for the entity. In addition, it allows creating new relations from the entity to other entities. Since adding navigational relations and creating connectors will create new links to other entities, the add relation privilege should imply the right to add navigational relations as well as to create connectors.

3.2.3.4 Implication of Remove Relation Rights

- The add relation privilege for an entity implies the right to remove relations from the entity.

The add relation privilege for an entity implies the right to read, create, and add relations for the entity. In addition, it allows removing existing relations originated from the entity.

3.2.3.5 Implication of Full Edit Rights

- The right to edit a first-class entity implies the right to add, modify, or delete its attributes.

Since the full edit privilege has a higher level of authorization than that of read, create, add relation, and remove relation, having the full edit privilege implies having all of these privileges. In addition, having the full edit privilege for a first-class entity allows users to modify its attributes.

3.2.3.6 Implication of Move Rights

- The right to move for a given entity implies the right to make the entity be a specialization of other entities .
- The right to move can be exercised if and only if the right to move is given in the originating entity of the move operation and the right to create is given in the destination entity.

Having the move right implies the ability to move an entity across the specialization hierarchy and to perform any operations on the entity that require the privileges of read, create, and edit. However, in a move operation, the destination entity must be able to adopt the entities being moved. Since the ability to adopt is implied by the create privilege, a user should not be able to move an entity unless he has the create privilege for the destination entity.

---

create operation, the set of user groups to be selected must be privileged to perform the create operation. Further explanation of the parent membership relation can be found in Chapter 4.

3.2.3.7 Implication of Delete Rights

- The right to delete an entity implies the right to remove the entity from its specialization hierarchy.

The delete privilege is the highest authorization level for the execution of all normal operations[13]. When possessed with the delete privilege for an entity, users can perform any operations on the entity, except those ones which modify its associated access control restrictions.

In deleting an entity from its specialization hierarchy, a few alternatives exist. The first way is to delete the entity alone, leaving its immediate children as children of the deleted entities' parents. Alternatively, one can deny the removal of an entity unless the user has the right to remove the entity and all of its descendants. However, this method would be too restrictive, as any specializations of an entity could prevent it from being removed. Finally, another option is that the removal of an entity will also result in the removal of all its descendant for which the user has the delete privilege. Using such scheme, subsequent massive cleanup operations may be required, as a large number of orphans may be created. Hence, the first method appears to provide a good balance between its alternatives, and is most appropriate for the use of the Process Handbook.

3.3 Access control Management

Managing the access control restrictions of an first-class entity includes the tasks of designating the restrictions for the entity as well as adding the entity to entity collections[14]. By adding an entity to an entity collection, the related user groups of the collection will be able to access the entity. As a result, the ability to add to an entity collection should imply the ability to designate access permissions to users. Hence, users without the right to grant or revoke access privileges should not be authorized to perform such operation. In particular, only the users with the maximum management right should be given the addition power.

To enhance the flexibility for access control management, the right to designate access control restrictions is further partitioned to support a finer granularity of control. In particular, it consists of the rights to grant and revoke each individual privilege, i.e., the privilege of read, create, edit[15], move, as well as delete. As shown in Figure 5, the levels of authorization are strictly ordered. Under this ordering, the possession of a high level management authorization for an entity implies the possession of all its lower level authorizations. For example, a user who has the grant create authorization also has the grant read authorization.

---

[13] Normal operations refer to all operations that do not relate to editing the access control restrictions of the entities.

[14] Description of entity collections can be found in Chapter 4.

[15] Note that the edit privilege is further partitioned into the full edit privilege, the remove relation privilege, and the add relation privilege.

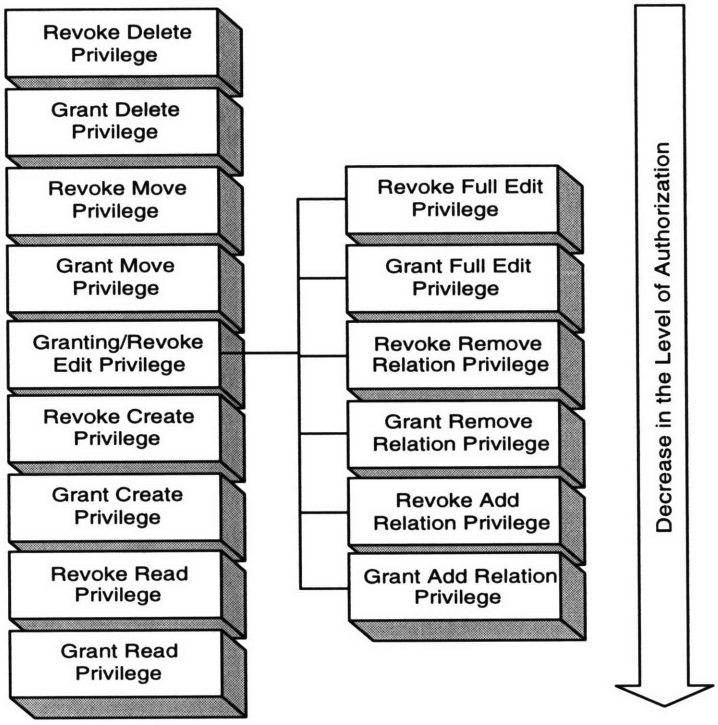


Figure 5: Management Authorization Levels

Using this fine-grained access control policy, a user who has the highest management right can give up his right, grant as well as revoke other's management rights. In the case where no user groups have the maximum management privilege for a specific entity, the privilege will be given to the system manager group[16]. As mentioned above, only users with the maximum management privilege should be authorized to add its managing entities to entity collection groups. Since the domain managers of an entity collection has the power to change its members' access control restrictions, adding an entity to an entity collection is identical to granting management privileges to the domain managers. Consequently, the addition power should only be given to users with the maximum management right.

Compared with other alternative access control management schemes, the chosen scheme offers both simplicity and flexibility. Alternatively, a scheme which provides a subset of the management operations offered by the chosen scheme would be simpler. However, in the chosen scheme, since the levels of authorization associated with the operations remain strictly ordered, little complexity

---

[16] To provide a central administration resource for the first-class entities of the handbook, a user group called system manager group is always defined in the user group hierarchy. The system manger group would be responsible for all central administration duties.

is added for the increased flexibility. Furthermore, it is desirable to provide a comprehensive set of operations in the server objects. While a client can naturally hide operations present in the server objects from the users, extending the functionality of a server object in a client is inappropriate and often complex. Finally, when only a limited set of operations is available in the system for access control management, users who wish to grant management rights to others would easily discover that he is granting either too much or too little power. As a result of the limited options, users would be much more reluctant to grant management rights, and hence, collaboration will be inhibited. Clearly, because of the collaborative nature of the Process Handbook, the chosen scheme is more desirable.

In addition, a rather different approach can be used. In particular, the management authorizations and the normal entity access authorizations can be combined to form a single ordered list. As a result, a very simple and straight-forward mechanism can be used to perform access checking. However, this scheme is less suitable because different combinations of the two sets of authorizations may be desired by different users of the system. For example, on one hand, Alice wants to allow users with read privileges to be able to grant read privileges to others; on the other hand, Bob may like to allow only users with delete privileges to be able to grant any privileges at all. Clearly, a more flexible scheme would be desirable to promote different forms and levels of collaboration. In addition, another advantage of the chosen scheme is that the separation of the user-vs-entity-management relationship from the user-vs-entity relationship allows a clean conceptual view of the control model, which is important in the design and maintenance of the system.

3.3.1 Comparison with Traditional Approaches

The approach proposed for the Process handbook combines the benefits of both traditional self-control and hierarchical-control approaches. In particular, in creating a new entity in the handbook, multiple user groups can be specified as having management privileges. When a single user group is designated as having the management privilege, management closely resembles a self-control approach. On the other hand, when multiple groups are designated, the chosen management scheme looks like that of the hierarchical-control approach, in which the benefit of having backup managers can be achieved. However, different from the hierarchical-control approach, the proposed method does not necessarily lead to concentration of power in a few user groups. The reason is that by allowing the presence of self-control entities, power can be distributed more evenly among users. As a result, the proposed approach is chosen over each of the traditional approaches.

Chapter 4: Further Extension to the Server API

This chapter proposes the appropriate extension to the server API to complement the access control policy discussed earlier. In particular, two new types of first-class entities, i.e., user groups and entity collections, will be introduced in the system. Any changes implied by the addition of these new entities will also be discussed in this chapter.

4.1 User Groups and Entity Collections

4.1.1 First-class Entities

To efficiently manage the access control relationship among entities and users, users are represented by user groups, and entities are organized into entity collections. User groups and entity collections are justified to be first-class entities because creating specializations and decomposition of these entities are useful. In particular, it is useful to create specialization of a user group where the specialized group will inherit the privileges granted to its parent, but that the restrictions imposed on accessing the methods of the specialized group differ from that of its parent. Similarly. It may also be desirable to create specialized entity collections where the access restrictions for the members of the specialized collections will be inherited from their parents.

Moreover, the decomposition as well as the membership relations are applicable to user groups and entity collections[17]. In particular, membership relations among user groups allow a user group to share the privileges given to its parents. Similarly, membership relations among entity collections allow the access control restrictions of an entity to be specified in its parents. Furthermore, as in all first-class entities, the decomposition relation provides information about how these entities are composed.

In Figure 6, sample membership hierarchies of user groups as well as entity collections are presented. Though the hierarchies do not need to be arranged as presented in the figure, the system manager group must be present in the user group hierarchy to act as the central administrator of the handbook. Moreover, the "User Universe" group and the "Entity Universe" collection will always be the roots of the two hierarchies. Any groups or collections created will automatically

---

[17] While the membership relation closely resembles the decomposition relation, they have slightly different interpretations. In particular, the membership relation is a more specific from of decomposition. To make the point clear, consider the membership and decomposition relation of organizations. It is apparent that the aggregation of all the members of an organization would not adequately represent the organization itself. Specifically, the organization would contain additional dimensions such as goals, rules, offices, etc.

become the members of the two roots respectively. Finally, since all user groups are entities, all user groups are also members of "Entity Universe."

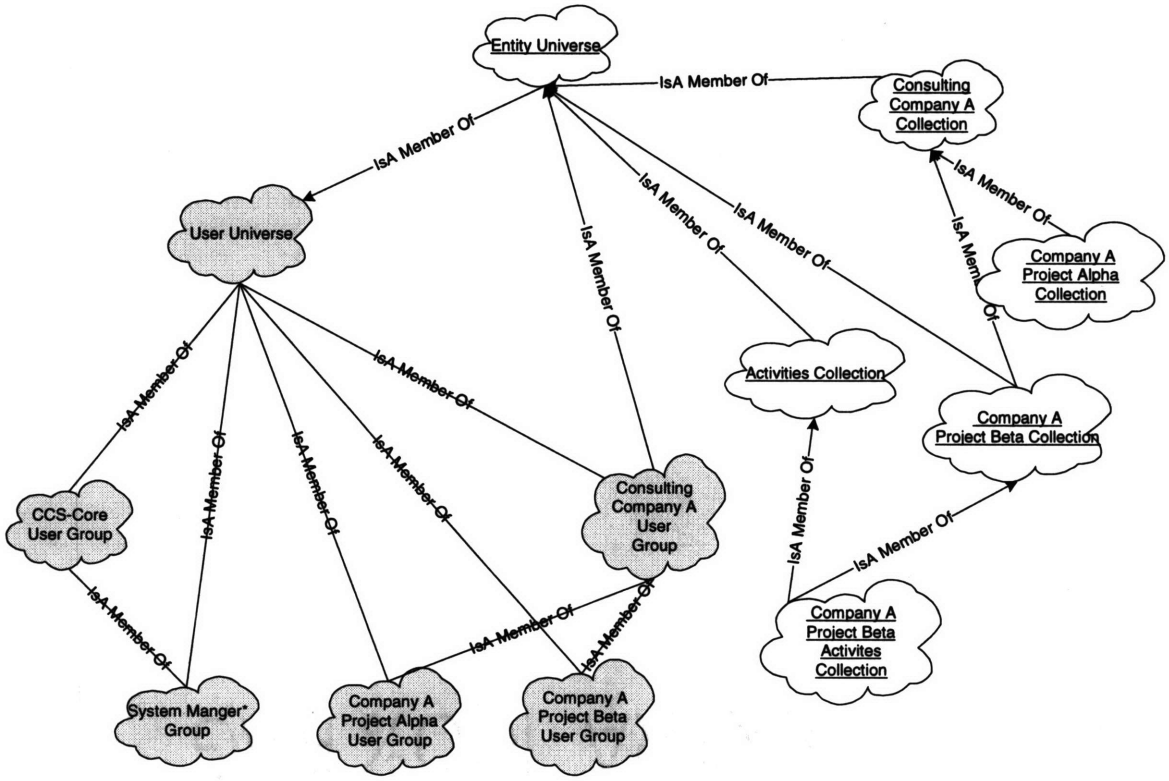


Figure 6: Membership Hierarchy of Entity Collections and User Groups. Non-shaded objects represent entity collections; shaded objects represent user groups.

With the entity collection and user group abstractions, membership relation among entities can be maintained. Moreover, an individual user group can be represented by a group without any members. In all groups and collections, their profiles[18], which provide useful information about their characteristics and status, will be maintained. Those information would be stored as attributes associated with the entities. Furthermore, passwords will be stored in user groups for authentication purpose. The passwords will be stored as an attribute and in a secure format[19]. For password attributes, their retrieval will be handled as an exception in which only users with the maximum management right are authorized to access.

---

[18] A profile for an entity collection includes its name, contact information and description. A profile for a user group includes its name, contact information and description, and session viewing preference.

[19] Since passwords are sensitive information, all passwords are compressed and transformed using a message digest algorithm, MD5. As discussed in Section 2.6, MD5 is conjectured to be computationally infeasible to break.

Finally, to avoid the situation where there exists an entity in the system where no users or groups have maximum management rights, special treatment is necessary. One possible solution is to prevent the above situation from occurring. Alternatively, one can resolve any inconsistencies that arise by making the system manager group assume the right to manage all entities which do not designate any groups to have the maximum management privileges. Comparing the two solutions, the latter is more flexible, since putting too many constraints in the server would leave few rooms for manipulation in the clients. In addition, since the complexity required to resolve inconsistencies after removing an user group is small, the second solution seems even more suitable.

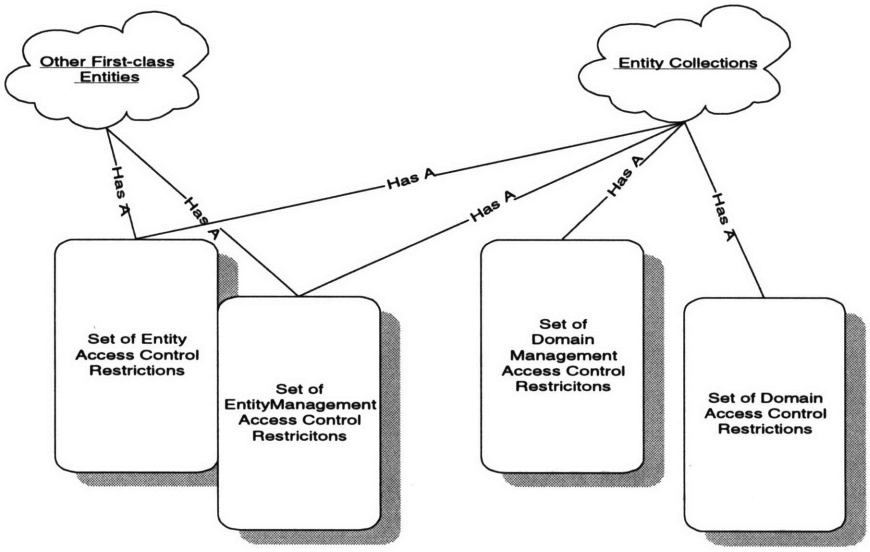


Figure 7: Access Control Restrictions for Entity Collections and other First-class Entities.

4.1.2 Enhancing Efficiency

By extending the server API with user groups and entity collections, access control restrictions can be maintained more conveniently and efficiently. This increase in maintainability is due to two reasons. The first reason is that an entity collection can specify the access control restrictions for its member entities. Different from other first-class entities, entity collections have two additional sets of access control restrictions, i.e., the set of domain access control restrictions and the set of domain management access control restrictions[20]. While the set of domain access control restrictions regulates the execution of its member entities' operations, the set of domain management access control restrictions specifies the users allowed to add and remove domain

---

[20] All first-class entities, except entity collection, have two sets of access control restrictions, i.e., the set of entity access control restrictions and the set of entity management access control restrictions. While the set of entity access control restrictions regulates the execution of the methods in the Entity class, the set of entity management access control restrictions specifies the users allowed to add and remove entity access control restrictions.

access control restrictions. Using domain access control restrictions, the access of a large group of entities can be controlled in a single location, and the need to update each entity individually is reduced. In Figure 7, the contrast between an entity collection and other first-class entities is presented.

Another reason for the increased maintainability is that access control privileges given to a user group applies to all its members. As a result, by giving the read privilege to "User Universe" to access the entity "Make Coffee,", all users in the Handbook will be authorized to read the content of "Make Coffee." In an alternative system where users are not related to one another, giving privileges to all users would require the system to give the privilege to each user individually. Moreover, when revoking privileges, one has to select each individual user to be removed, rather than selecting a group of users at a time. Clearly, membership relation among user groups provide great convenience for managing access control restrictions.

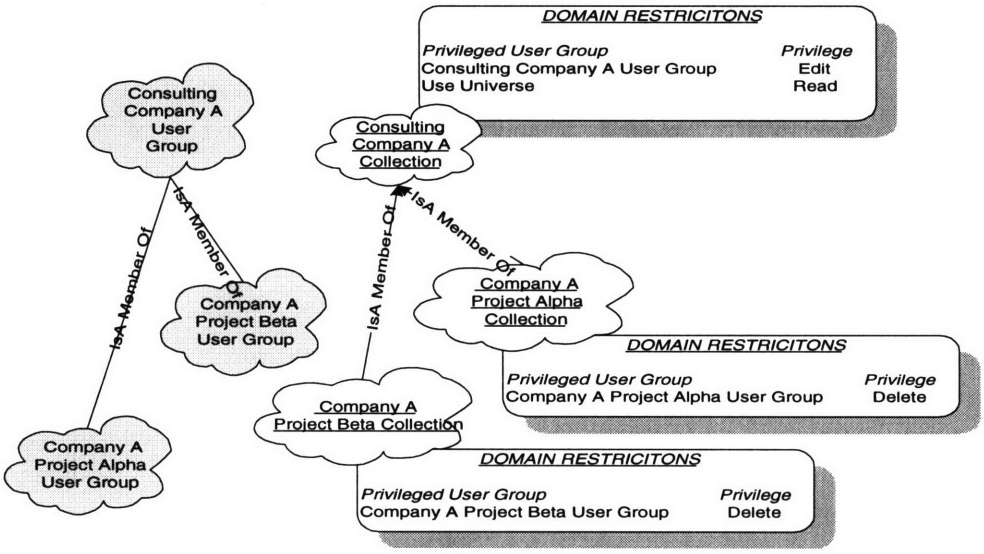


Figure 8: An Example of Using Entity Collections and User Groups. Non-shaded Objects represent entity collections; shaded objects represent user groups.

To provide a concrete understanding of the benefits provided by user groups and entity collections, an example is provided here. Consider that Company A is a consulting company which is currently undertaking two particular projects, alpha and beta. To efficiently maintain the access control restrictions for each of its projects, one possible way is that a new user group and entity collection is created for each of its projects, with all relevant entities added to the appropriate project entity collections. (See Figure 8.) Moreover, all project user groups are added to the company user group, and all project entity collections are added to the company entity collection. Finally, each project entity collection would designate its corresponding user group as the only group privileged to have full access of its members. Meanwhile, domain access privileges would be specified in the company entity collection. For example, all members of the company user group will have create privileges, and all members of "User Universe" will be granted read privileges in the company domain. As a result, changing the set of users with full access to the

projects only requires changing the members of the corresponding user groups. Moreover, when additional projects are undertaken, new user groups and entity collections can be created while the access control restrictions for each project can be maintained in isolation.

Clearly, the membership relation among user groups and entity collections can be carefully designed to satisfy different needs. The above example only presents a particular way where one could systematically manage the access control restrictions for a given set of entities.

4.1.3 Membership Relation

To add an entity to an entity collection, the user should have two specific privileges. In particular, the subscriber must have the add relation privilege for the desired entity collection, and the maximum management right for the entity to be added. Similarly, to remove a membership relation, the user must have the remove relation privilege for the desired entity collection, and the maximum management right for the entity to be removed.

As an important note, membership relation is transitive in nature. In other words, for user groups, if "Project Alpha" is a member of "Company A," and "Company A" is a member of "Enterprise B," "Project Alpha" will entitle to the union of privileges given to "Company A" as well as "Enterprise B." Similarly, for entity collections, the domain access restriction for a given entity collection applies to all direct or indirect members of the collection.[21]

Finally, membership relation can be two-way. For example, "Group X" can be made a member of "Group Y" while "Group Y" is currently a member of "Group X." Such property does not conflict with the proposed access control model. Specifically, for user groups, if "Group X" and "Group Y" are mutual members, privileges given to one group will be given to the other group. For entity collections, a mutual member relationship implies that the domain access restriction defined in a collection is applicable to the collection where the restrictions are defined.

4.1.4 User and Entity Universe

The "User Universe" group, and the "Entity Universe" collection are organized to provide single roots of navigation for both user groups and entity collections. Under this model, all entities will be members of "Entity Universe" while entities representing user groups will also be members of "User Universe." To maintain such relationship, all entities must be directly or indirectly related to "Entity Universe." Meanwhile, all entities representing user groups must also have a direct or indirect membership relation with "User Universe." A direct relation is one where an entity is explicitly specified as a member of a group or collection, while an indirect relation is one where the membership relation is realized through the transitivity property. However, if entities are allowed to relate to "Entity Universe" indirectly, the system must constantly monitor the membership relations among entities so that each entity is a member of at least one group. As a result, tremendous burden is placed on the system to maintain the membership relation. Even if back

---

[21] Definitions for direct and indirect members can be found in Section 4.1.4.

pointers are used, given the current database schema, repairing a corrupted database is complex[22]. On the other hand, if all entities are required to relate directly to "Entity Universe," accountability can be achieved without much work, and database repairing can be done with a single query. Therefore, in the proposed model, all entities are required to be a direct member of "Entity Universe." In addition, all entities representing user groups are required to be a direct member of "User Universe."

As a result of the above requirements, removing an entity from "Entity Universe" must correspond to the removal of the entity in the system. Similarly, removing a user group from "User Universe" must correspond to the removal of the user group. Hence, removing membership relations in "Entity Universe" and "User Universe" bear special consequences, and the privileges for performing such operation should be limited to the system manager group alone.

4.2 Addition and Removal of First-class Entities

With the addition of entity collections, the creation and removal of all first-class entities would require slight changes. In particular, when a first-class entity is created, it would become a member of "Entity Universe". Moreover, for entities representing user groups, they would become members of "User Universe." Using this procedure, the membership hierarchy of both entity collections and user groups will have a single root of navigation. Hence, a user can easily make certain entities world-readable by giving the "User Universe" group read privileges, or make all entities readable by certain user groups by granting the groups read privileges for the "Entity Universe" domain.

Similar to the procedure discussed above, the removal of first-class entities now require additional cleanup procedures. Specifically, when a first-class entity is removed, all membership relation which refers to the removed entity will be destroyed. Moreover, all access control restrictions pertaining to the entity being removed will be deleted as well. If the removed entity is a user group, the system manager group will assume the right to manage all entities which do not designate any user groups to have the maximum management privilege after the removal.

Finally, because of the presence of membership relations among entity collections and user groups, the access control inheritance mechanism has to be updated. In particular, when inheriting access control relations in creating specialization, the new specialized entity will be granted the same set of membership relations as its parent[23]. In this way, the new entity will share not only the same set

---

[22] If back pointers are used, database repairing requires multiple queries to be executed. On the other hand, if all entities are required to relate directly to "Entity Universe," database repairing can be done efficiently using a single query.

[23] As an important note, membership relations are granted, not inherited, to the specialized entity. As shown in the scenario presented in Appendix D, while an entity has changed from one review status collection to another, its children should not be affected at all.

of entity access control relations with its parent, but also the same set of access control relations implied by the membership relations.

4.3 Authentication to the Process Handbook

The discussion so far has only touched on the low-level structure for storing authentication information, i.e., passwords stored as attributes in user groups. To make the low-level procedures useful to the handbook, a high-level authentication protocol must be present to deal with the interactions among clients and servers. In this section, a brief description of the user authentication model is presented.

In the core of the authentication model, the SSL (Secure Socket Layer) protocol, discussed in Section 2.5, is used. Since SSL is one of the standard protocols used for establishing secure channels in the Internet, many existing free and commercial software are layered on top of this protocol. For example, most web clients and servers support communication using SSL. Hence, the Process Handbook Server can directly employ such tools, without the need to implement its own set of routines for establishing private channels. Through the SSL protocol, three basic properties of a secured channel are achieved:

- The channel is private. Encryption is used for all messages after a simple handshake is used to define a secret key.
- The channel is authenticated. The server endpoint of the conversation is always authenticated.
- The channel is reliable. The message transport includes a message integrity check (using a MAC).

In the Process Handbook authentication model, user group authentication will be done through matching the message digest[24] of the user group password with that set in its profile. Since the communication channel is private, assuming that the adversary is computationally-bounded, a valid name and password pair can only be supplied by the actual users. Moreover, the reliability and privacy of the channel protects subsequent exchange of messages from being manipulated.

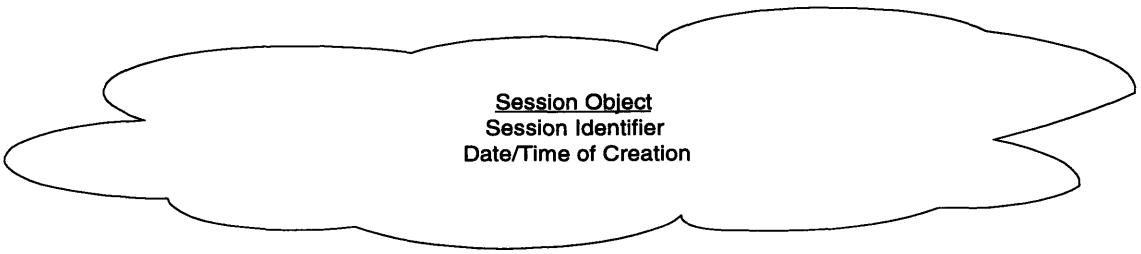


Figure 9: A Session Object

[24] The passwords are compressed and transformed to form message digests using the MD5 algorithm.

Once an user is authenticated to a user group, the server would keep track of the state of the communication channel by generating and exporting a session object. The session object will bind a large random number with a time stamp to form a session identifier. (See Figure 9.) By binding a random number with a time stamp, the search space required to guess a valid session identifier would not decrease as more and more new session objects are created. Assuming the system cannot process multiple create-session requests simultaneously, each session object would contain a different time stamp. Hence, in any circumstances, given a valid time stamp, the search space of a valid session identifier is always identical to the size of the random number generated.

In a web client and server environment, the web server can extract the session identifier of the session object and export a cookie similar to the one shown in Figure 10. During subsequent access, instead of having to check user passwords again for authentication, the server would verify the content of the cookie. After each session, the cookie will be automatically removed from memory.

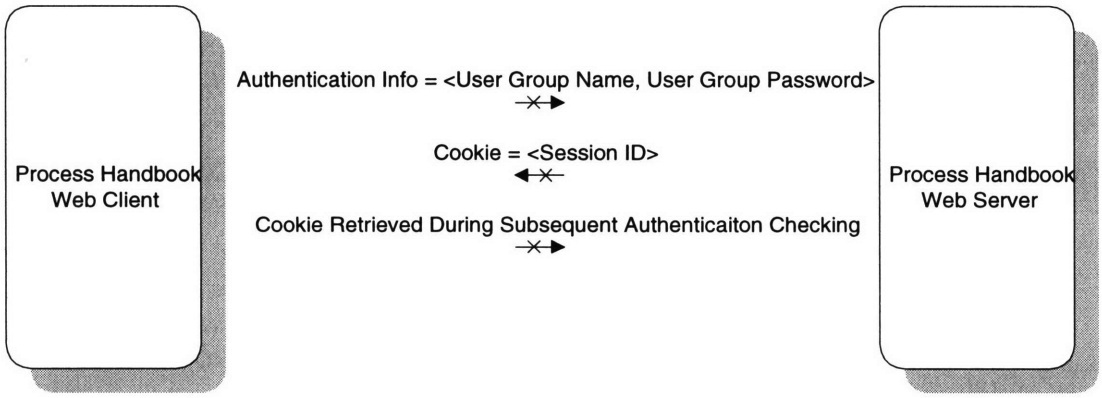


Figure 10: Process Handbook Authentication Protocol In a Web Environment

During the exchange of messages, all information will be private to the client and server because of the established secured channel by SSL. When authenticated, a user would become empowered to exercise privileges granted to his authenticated user group, as well as other groups which have a parent membership[25] relation with the authenticated group.

[25] If A has a parent membership relation with B, B is a member of A.

Chapter 5: Design of Access Control Mechanism

In the Process Handbook, since the information for user accounts and other entities are stored in a relational database, queries have to be executed in checking access authorization. From the high level perspective, access control lists and capabilities provide different virtual views of the access control relations. While access control lists provide the view from an entity's point of view, capabilities provide the view from a user's perspective. However, in the implementation level, the abstract view is flattened. Specifically, both views will be extracted from a single table storing the access control relations among user groups and entities.

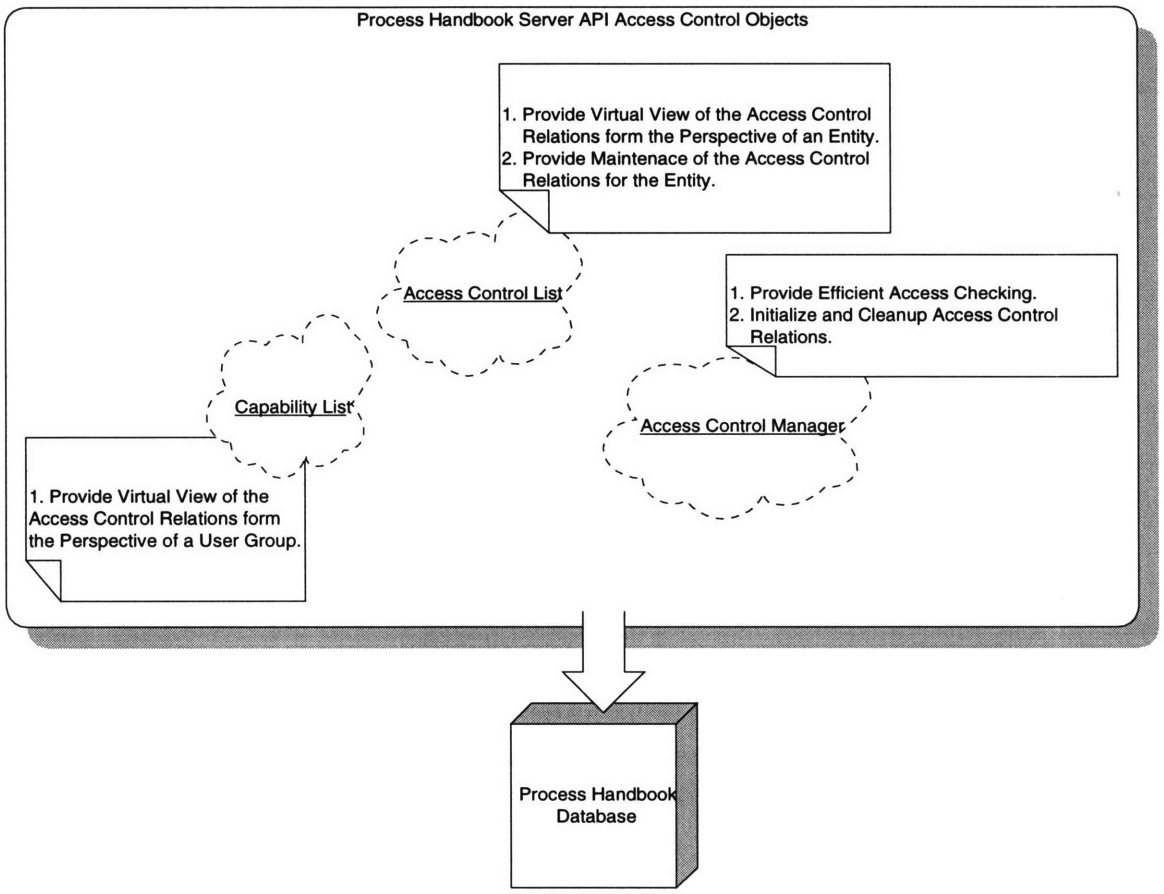


Figure 11: Process Handbook Access Control Objects

To make use of the access control list or capability models, caching will be necessary for fast retrieval. During subsequent access checking, the system would first retrieve the relevant access control or capability list from the cache, and then process the list to extract the result. Comparing a mechanism which uses the access control list model to one which uses the capability model, they have comparable performance. In particular, depending on the specific situations and implementations, each mechanism may be preferable to the other. For example, assume that both

access control and capability lists are implemented as linked lists. Given these specific implementations, if the access control list of the requested entity is short, access checking using the access-control-list view would be efficient. On the other hand, if the capability list of the requested user group is short, access checking using the capability-list view would be preferable. However, a major drawback of both mechanisms is that since the lists are often fairly long, the cost of maintaining the cache is extremely expensive. Yet, if caching is not employed, the appropriate lists have to be extracted form the database during each access checking, and the efficiency of the system would be seriously affected.

Alternatively, a SQL query can be used to extract the correct user group access privilege directly from the database. Using this method, the access control list and capability list abstraction would be used solely as virtual views of the access control relations in the system, and for their maintenance. Since all computations are done in a single query, redundant operations can be reduced; hence, the performance of this mechanism is faster than that of extracting and then processing an access control or capability list from the database. Moreover, it can be further optimized by caching the membership relation among user groups and entity collections. Since modifications of the access control lists are infrequent comparing to normal access, the cost of cache misses would be small. In addition, the cost of maintaining the cache is much smaller than the above scenario because the amount of space needed to store the membership relations is much smaller. In the Process Handbook access control mechanism, the access control manager (ACMgr) class is implemented to by-pass the access control list and capability abstraction. As shown in Figure 12, it serves as an abstraction for efficient access checking, as well as for initializing and cleaning up of access control relations.

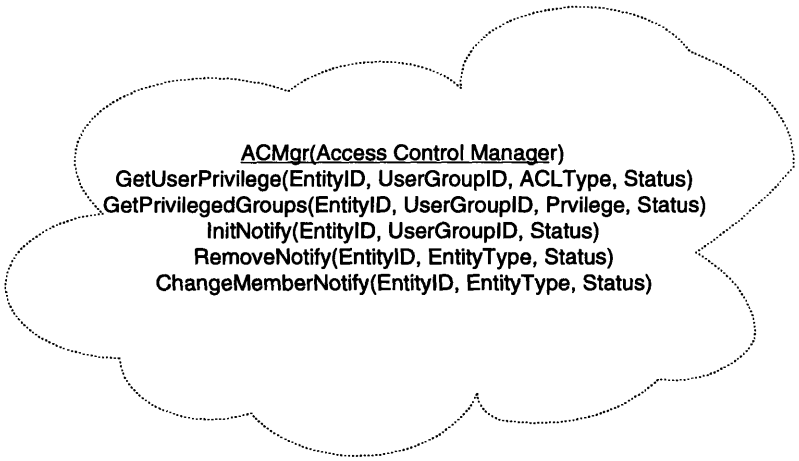


Figure 12: Access Control Manager Class

5.1 Access Control Manager

The access control manager class provides an abstraction for retrieving access privilege as well as for initializing and cleaning up of access control relations. Using the manager class abstraction, multiple checking or updating can be done using the same object. As a result, optimization can be performed to improve efficiency. In particular, caching can be done within a manager object. By maintaining a cache to store the membership relations, the system can reduce the search space needed in checking authorization. Moreover, since modifications of the membership relations are infrequent comparing to normal access, the cost of cache misses would be negligible. Therefore, the access control manager class, shown in Figure 12, provide enhanced efficiency over a more decentralized design. (For the specifications of the access control manager class, see Appendix A.)

5.1.1 Using the Access Control Manager

To demonstrate the usage of the access control manager, consider the scenario where Bob wants to read the attributes of the process "Make Coffee." (See Figure 13.) As the first step, Bob authenticates to his individual user group, "Bob", by providing the authentication information stored in his profile. After successful matching of identification using an authentication manager object[26], Bob will be given a session object and authenticated to use the privileges given to "Bob," as well as all other groups which have a parent membership relation with his group. These relevant groups will be cached in an access control manager object during the first time when the membership relations are retrieved. To get the content of "Make Coffee," the client sends a message to the GetAttributes method of the "Make Coffee" object. For the purpose of access control, it also passes the session identifier in the session object to the server, which would then send a message to the Authenticate method of the authentication manager object. If authentication is successful, Bob's user group identifier will be returned to the server. Afterwards, the server would pass the returned user group identifier, the type identifier for the access control relation[27], as well as the entity identifier of "Make Coffee" to the GetUserPrivilege method of the access control manager object to get Bob's actual privilege. As the last step, the server compares the returned privilege with that required by the read operation, and decides whether to return the requested information or not .

In addition, when creating specializations, it may be desirable for a user to select the context where he performs the create operation. The set of all possible contexts is represented by the set of all user groups in the Process Handbook which have the create privilege for the entity and the parent membership relation with the user. To determine this set of all possible contexts, the server can

---

[26] The specification of the authentication manager (AuthMgr) class can be found in Appendix A.

[27] The type identifier for the access control relation (ACLType) specifies whether access control relation represents entity access control restrictions (PH_AC_ENTITY), entity management access control restrictions (PH_AC_ENTITY_MANAGEMENT), domain restrictions (PH_AC_DOMAIN), or domain management restrictions (PH_AC_DOMAIN_MANAGEMENT).

send a GetPrivilegedGroup message to an access control manager object. In return, the manager object will then return an array of identifiers which correspond to the identifiers of the context groups. Meanwhile, designation of the user groups which have the maximum management right for a newly created entity can be done by sending a InitNotify message to a manager object. As a result, the groups specified will be given the revoke delete privilege in the entity management access control list, and the delete privilege in the entity access control list. If the entity is an entity collection, the groups will also be given the revoke delete privilege in the domain management access control list, and the delete privilege in the domain access control list. Finally, when an entity is removed, calling the RemoveNotify method of the access control manager class will remove all access control restrictions that relate to the entity, avoiding useless information from being accumulated in the database.

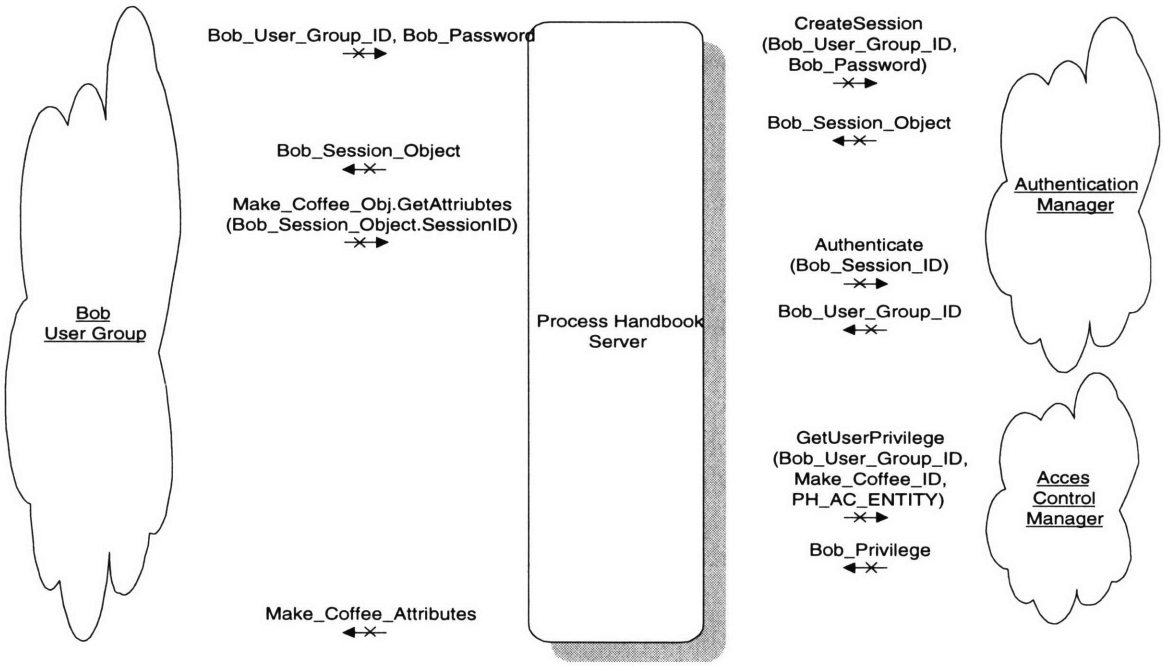


Figure 13: Example of Using the ACMgr Class.

5.1.2 Evaluating Access to Entities and Entity Collections

While each entity can be in multiple entity collections, each entity collection may be associated with multiple user groups. As a user can be members of multiple user groups, there may exist a situation where a user has conflicting access rights in performing an operation. Hence, one has to pay special attention in resolving access permission conflicts.

To resolve the issue of permission conflicts, the following solutions have been considered:

- Assume minimum privilege from the set of conflicting privileges

- Assume maximum privilege from the set of conflicting privileges
- Assume the privilege designated in the most specialized access control relation among the relevant entities and their related user groups.

In evaluating the alternative solutions, one would realize that the first solution could be easily rejected. While increasingly restrictive user groups should have increasingly privileged permissions, the minimum privilege rule could not take advantage of such property. To demonstrate the point, consider an example where the manager of a process wants to have two general groups of users. While the first group contains all users of the handbook, the second group is restricted to CCS members. Moreover, only the second group has the privilege of editing the content of the process. However, such setting cannot be accomplished since all CCS members are users of the handbook, and by the minimum privilege rule, the second group is only given the read privilege. Clearly, this example shows the limitation of the first solution.

Contrasted to the first solution, the second solution can accomplish the setting desired in the above example. Using the maximum privilege rule, a group and its members can relate to an entity simultaneously without destroying the more privileged permissions given to the members. As member groups are entitled to the privileges given to their parents, the problem of setting up a less privileged member group is not relevant in the defined access control model, and can be ignored.

Using the third solution, privileges will be partially determined by the positions of the relevant entities and user groups in the specialization hierarchy. Under this scheme, promotion and denial of privileges can be flexibly accomplished by propagating and overriding access privileges through specialization. However, such flexibility comes at a cost. The reason is that a unique ordering of the access control relations in the specialization hierarchy does not exist. To resolve ambiguities of ordering, rules such as the maximum and minimum privilege rules have to be used. Despite the flexibility provided, the complexity of this solution makes it less desirable to the Process Handbook. Hence, the good balance of flexibility and efficiency makes the maximum privilege rule the optimal solution to be used.

5.2 Access Control List and Capability List

The maintenance and accounting of the access control relations is provided by ACL (access control list) objects and CapL (capability list) objects. Using the ACL objects, a first-class entity can obtain the list of user groups privileged to access its content. Moreover, the managers of an entity can add and remove access control entries for its associated ACL object. On the other hand, CapL objects allow a user group to review its access control relation with other entities in the system. However, except for inheriting privileges in specialization, the content of a CapL object cannot be changed because granting and revoking privileges for an entity should not be done from the user group perspective. Both ACL and CapL classes are shown inFigure 14.

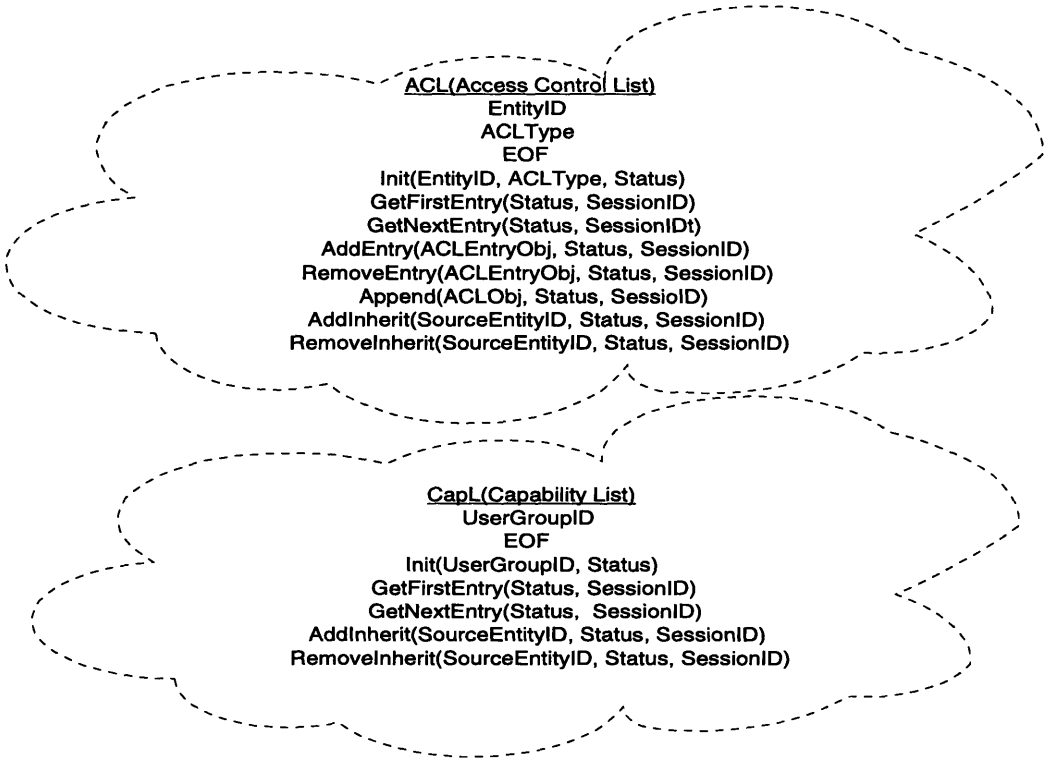


Figure 14: Access Control List and Capability Classes

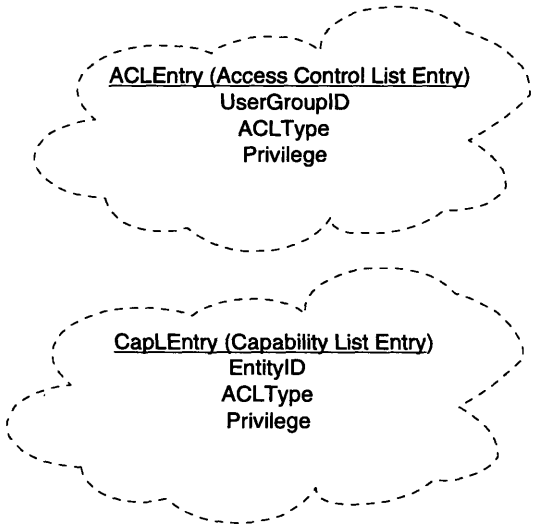


Figure 15: ACLEntry and CapLEntry Classes

Since both ACL and CapL objects provide virtual views of the database content, consistency issues arise. In particular, the views provided by the objects may become inconsistent with the data in the database immediately after the objects have been retrieved. However, inconsistency would not

affect the maintenance of ACL objects. The reason is that removing access control relations that have already been removed somewhere else does not have any undesirable consequences. Similarly, while different privileges could be granted to the same user group within a close interval, the maximum privilege rule is there to resolve any conflicts.

To allow users to traverse the content of ACL and CapL objects, ACLEntry and CapLEntry objects are used. Each ACL object produces a set of ACLEntry objects that correspond to the privileges given to the users. Similarly, each CapL object produces a set of CapLEntry objects that correspond to the privileges possessed by its associated user group. While an ACLEntry object specifies a privilege for a given user group, a CapLEntry object specifies a privilege for a given entity. Figure 15 shows the ACLEntry and CaplEntry classes.

5.2.1 Using ACL and CapL

To demonstrate the usage of ACL, consider the scenario of Bob granting Alice the create privilege for the activity "Make Coffee," and assume that Bob has already retrieved its ACL object. To grant the read privilege to Alice, Bob will construct a new ACL entry object which stores Alice's user group identifier, the type identifier[28] for the access control relation, as well as the permission level which corresponds to the create privilege. Using the AddEntry method, the specified access control relation between "Alice" and "Make Coffee" will be added to the database. Inside the method, access checking described in Section 5.1.1 will be executed to determine whether Bob has the grant create privilege for the entity "Make Coffee."

In another situation, Bob, authenticated to the user group "Bob", wants to look at what privileges are given to his user group. As a result, Bob requests his user group object to return its capability list. After the regular access control checking using an access control manager object, the system could determine whether Bob has adequate privilege to read the entity. Assuming Bob has sufficient authorization, his request will be granted, and a CapL object of Bob's group will be returned. Afterwards, Bob can examine each entry of the list by using the GetFirstEntry and GetNextEntry methods of the CapL class. Traversing an ACL object is identical to that of CapL; using the GetFirstEntry and GetNextEntry of ACL, the ACL object would return an ACLEntry object for every access restriction it contains, and users can then inspect the properties of each of its returned objects.

Finally, as a convenient way for combining ACL objects, the append method in the ACL class allows copying the access control restrictions contained in an ACL object to another ACL object. This method is particular useful in certain specific situations. For example, if a user wants entity "A" to use the same access control restrictions as entity "B", and that changes made to the access control list of entity "B" should not affect that of entity "A," the append method of the ACL class would perfectly satisfy his goal. On the other hand, if a user prefers to have a group of entities share the same set of access control restrictions, the group of entities can be placed under a single

---

[28] In this case, the type identifier for the access control relation is PH_AC_ENTITY.

domain. In other words, if the entities are added to a particular entity collection, the access of all entities would be controlled by its domain access restrictions. Finally, similar effects can be achieved with access control inheritance. By specifying the specialization of an entity to inherit its access control restrictions, any access control changes made to the entity will be propagated to its specialization.

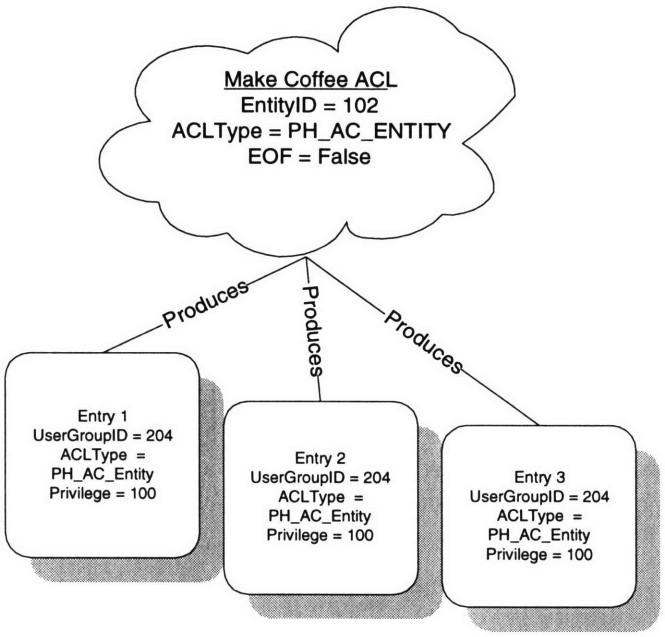


Figure 16: ACL for "Make Coffee" and its associated ACLEntry objects

Chapter 6: Discussion

6.1 Summary of API Changes

To integrate the proposed access control policy to the API objects, the first-class entities must be extended to execute access checking before performing every requested action. Moreover, since each operation may require a different level of privilege, the appropriate privilege requirement should be identified using the rules proposed in Section 3.2 as the basis. Meanwhile, to link entities to their corresponding access control relations in the database, ACL objects are maintained as entity attributes. When the entity represents a user group, an additional attribute storing its associated CapL object is also maintained. Finally, as discussed in Chapter 4, two new first-class entities are added, namely, user groups and entity collections.

As user groups and entity collections are discussed in depth in Chapter 4, and that storing ACL and CapL objects as entity attributes is self-explanatory, this section focuses on the low-level details of access checking. In particular, the discussion will be divided into 3 parts. While the first part investigates what changes are needed in the parameters of first-class entities' methods, the second part summarizes how to make use of the parameters and the access control manager class to execute access checking. Finally, based on the rules proposed in Section 3.2, the minimal privilege required for each method of the Entity class will be identified.

6.1.1 Additional Parameters for First-class Entity's Methods

For every method implemented in the Entity class, additional parameters are needed to identify the user which sends the message to the entity. While it appears that a user group identifier will be sufficient for identification purpose, it turns out that additional security checking should be placed for authentication purpose. As malicious users can easily come up with valid user group identifiers, authentication based on user group identifiers would not be adequate. As discussed in Section 4.3.3, a possible solution is the use of session objects. Since the session objects binds a large random number with a time stamp, it would be extremely difficult for any computationally-bounded adversary to produce the content of any valid session identifiers. Therefore, the session identifier contained in a session object should be passed to the server during the execution of each method of the Entity class. If the parameters are consistent with the set of records stored in the server, the server could trust that the user identifier specified indeed corresponds to the actual user. Thus, for every method implemented in the Entity class, the content of the session object must be passed.

6.1.2 Access Checking Using an Access Control Manager Object

Once the server has confirmed the validity of a session object, it can then pass the corresponding user group identifier together with an entity identifier to the GetUserPrivilege method of an ACMgr object. Furthermore, during the call to GetUserPrivilege, the entity should specify the relevant type

of access control relation. In particular, when the user is executing the methods of an entity, the relevant type is PH_AC_ENTITY. On the other hand, when the user is modifying an entity's ACL object, the relevant type should be PH_AC_ENTITY_MANAGEMENT. Finally, the types PH_AC_DOMAIN and PH_AC_DOMAIN_MANAGEMENT, which refer to domain-related access control relations, are also available.

6.1.3 Identifying Minimal Privilege Requirement

Based on the rules proposed in Section 3.2, the minimal privilege requirements for the methods available in the Entity class are shown below:

Minimal Privilege Requirement	*Methods*
PH_AC_READ_PRIVILEGE	GetAttribute GetAttributes GetAttributeValue GetAttributeObject* GetNavRelations* GetSpecializations* GetGeneralizations* GetDecomposition* GetWhereUsed* GetEndPorts* GetConnectors*
PH_AC_CREATE_PRIVILEGE	AddNewSpecialization AdoptSpecialization
PH_AC_ADD_RELATION_PRIVILEGE	AddNewNavRelation CreateConnector
PH_AC_REMOVE_RELATION_PRIVILEGE	DeleteNavRelation
PH_AC_EDIT_PRIVILEGE	AddNewAttribute AddItemtoAttrCol RemoveItemfromAttrCol SetAttributeValue SetAttributeObject DeleteAttribute AddDecomp RemoveFromDecomp ReplaceInDecomp
PH_AC_MOVE_PRIVILEGE	MoveToParent AdoptSpecialization[29]
PH_AC_DELETE_PRIVILEGE	RemoveFromParent

[29] The move privilege must be present for the entity to be adopted.

For all methods marked with an asterisk, the minimal privilege required depends on the entities at both ends of the link. Therefore, for these methods, the system should check the user privileges at both end of the relation. Finally, to obtain the corresponding authorization level for granting and revoking management privileges, one can subtract a management offset, PH_AC_MANAGEMENT_PRIVILEGE_OFFSET, from the normal privilege level. For example, the authorization which corresponds to granting read privilege is PH_AC_READ_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET; and that corresponds to revoking read privilege is PH_AC_READ_PRVILEGE + PH_AC_MANAGEMENT_PRIVILEG_OFFSET. A summary of these constants can be found in Appendix B. Using this construct, the number of constants required would be tremendously reduced. As a result, one could avoid the need of huge case statements in the body of the access control code, and consequently, improve the maintainability of the system.

6.2 Evaluation

The objective of this section is to evaluate how the access control policy fulfills the goals identified in Section 1.2. In particular, we would like to answer the following questions: Is the system flexible enough? Does it offer adequate protection of the information stored? Is it efficient and scalable?

6.2.1 Flexibility

The ability to manipulate the membership relation among user groups and entity collections provide tremendous flexibility to the users, as they can decide how the access control restrictions for their entities are to be maintained. Moreover, the proposed system can easily hide information under development by adding entities under construction to a "Scratch Area" collection, and giving its domain privilege only to the relevant developers. Moreover, this design can support incremental disclosure of changes made to the Handbook by transferring entities between collections, e.g., between a "Scratch Area" collection and a "Product Release" collection.

In addition, many new forms of collaboration can be achieved. For example, users can now compose public entities using private ones, or private entities using public ones. Since the model does not impose any additional access constraints on related entities, the access control of proprietary entities would not be affected even when linked to public entities. In addition to the fine granularity of control offered by the model, entity access restrictions are separated from its management restriction. Consequently, in one entity, managers who can grant read privileges may only have read privileges for the entity; in another entity, all managers may have at least the edit privilege.

As an interesting application, the proposed access control model can support switching between different views of the Process Handbook. For demonstration purpose, consider that there are two user groups called "Expert Users" and "Novice Users". While "Expert Users" can view anything in the system, "Novice Users" can only view a restrictive set of entities. Moreover, assume that all users are allowed to add and remove members of both "Expert Users" and "Novice Users". With these arrangements, a user authenticated to his own user group can dynamically add and remove

himself as a member of "Expert Users" or "Novice Users"[30]. By changing the membership relations, the access control mechanism would automatically regenerate the set of privileges entitled to the authenticated group, and consequently switch the user view of the Process Handbook. For example, by removing the authenticated group as the member of "Expert Users" and adding it as the member of "Novice Users", the system would have switched from the "Expert Users" view to the "Novice Users" view.

Finally, since an entity can have more than one user group with the maximum management right, the management scheme resembles both self-control and hierarchical-control approaches. Therefore, the creator of an entity can choose to manage the entity by himself, if he fears that too much power is being concentrated in a few user groups, or to designate backup managers to share the management responsibility.

6.2.2 Protection

By granting access to the desired user groups alone, the proposed access control mechanism prevents irrelevant user groups from executing protected methods of the Entity class. Moreover, with the authentication protocol discussed in Section 4.3.3, faking the identity of another user group will be computationally infeasible. As communication among clients and servers is protected by SSL, eavesdropper could not gain information regarding user group passwords as well as contents of any entities in the system. Even if a user has read access to the database, the passwords of the user groups are converted to message digests using the MD5 algorithm. Again, predicting the password that correspond to a particular message digest is not feasible within a computationally-bounded environment. Therefore, the system can be safeguarded from attacks by any computationally-bounded adversary.

6.2.3 Efficiency and Scalability

The system is both efficient and scalable. As discussed in Chapter 5, because of efficiency reasons, a few alternative solutions have been rejected during the design of the access control mechanism. In particular, using the access control list or capability list metaphor for access checking is undesirable because the number of entities associated with a user or the number of users associated with an entity grows much faster than the membership relation among users groups and entity collections. Therefore, caching all access control relations would not be a scalable choice compared to that of caching only membership relations. Moreover, since finding the membership relations represents the bottleneck operation in access checking, caching the membership relations allows the efficiency of the access checking mechanism to be higher than that of other alternatives.

---

[30] The user should also have the maximum management right for his authenticated group in order to add and remove membership relation to his authenticated group.

Moreover, the proposed mechanism for initializing and cleaning up access control relations avoids the need of constantly monitoring and enforcing certain relationship among entities. Since each entity in the system is a member of "Entity Universe," database repairing can be done efficiently. Moreover, the system does not have to make sure that an entity is still a member of some other entities when a membership relation has been removed. Rather, the system has to pay attention only when a membership relation is being removed from "Entity Universe."[31]. Similarly, instead of checking whether an entity has designated at least one user group to have the maximum user privilege whenever a user group is being removed, the system delays the checking until an ACL object associated with the entity is requested. In this way, the checking can be done with much less work, as most of the work required has to be done anyway in retrieving an ACL object. Finally, the lag time is tolerable because the system manager group will be given the maximum management privilege before any users attempt to check or modify the ACL objects associated with the entity. Hence, they will know whom to contact in any circumstances.

---

[31] For user groups, when the membership relationship is removed either from "Entity Universe" or "User Universe," the user group should be removed from the system.

Chapter 7: Future Work

Currently, the access control model has been built in a simple environment aimed at demonstrating the functionality and behavior of the model. In this environment, each first-class entity is simplified to contain only a name and identifier pair, and that the only relation implemented is the membership relation. A sample screen of the user interface is shown in Figure 17.

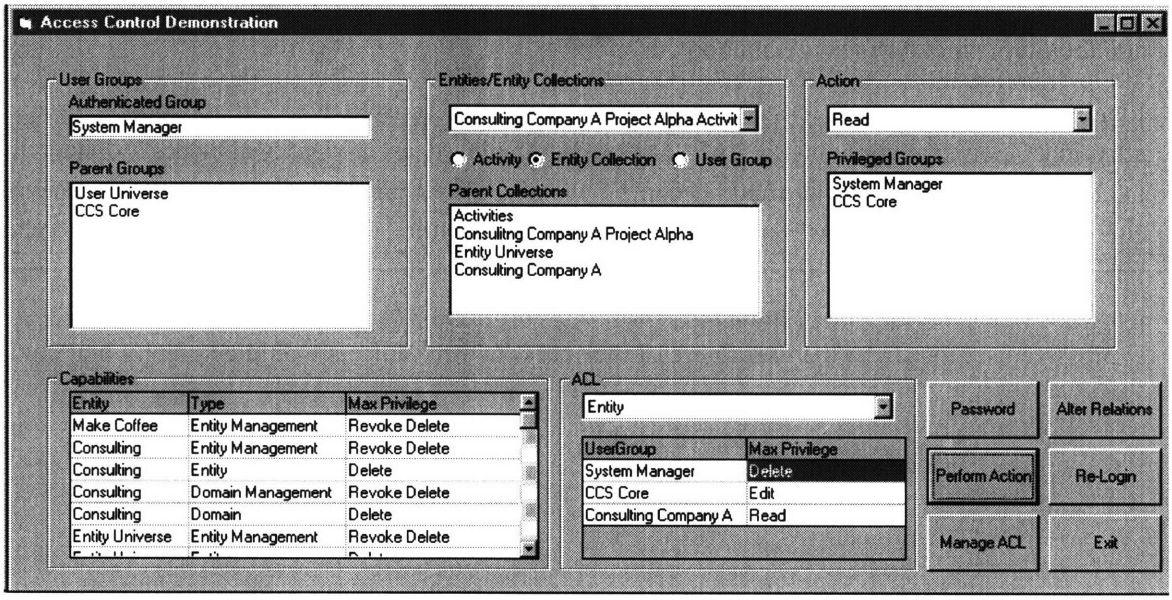


Figure 17: Demonstration Screen

When the extension proposed in Section 6.1 has been made to the Process Handbook object API, the access control module can be migrated to a client and server environment. As a result, multiple user sessions can be supported. Moreover, users can extract a much richer set of characteristics from user groups and entity collections in the actual environment. In particular, one could experiment with employing both specialization and the membership relation to satisfy specific access control requirements. In addition, as discussed in Section 3.2.1.1, extensions can be made by imposing access control restrictions on attribute objects. Consequently, one can have different access control restrictions on different attributes, achieving a even more flexible access control model.

Furthermore, when the access of entities is regulated, specializations may not be visible to all users. Consequently, modifying the content of an entity may lead to changes hidden to the users, i.e., changes that have been made to the hidden specialized entities. Therefore, the issue of version control arises. In particular, since modifications of an entity affect its specializations, what mechanisms should be present to maintain such propagation of changes? A possible solution is that users of the specialized entities will be notified when an entity higher up in the specialization

hierarchy has been modified. Afterwards, changes will be propagated, overwriting the original content of the affected entities. Yet, a better solution may be that the content of each affected entities will be copied to a new entity which will then be added as the child of a version control repository collection for the specific entity. Consequently, users can analyze the differences among various versions, and draw valuable observation. As the idea of version control fits nicely with that of incremental disclosure of changes, the idea of having a basic workspace for each entity in the Handbook can be introduced. In one part of the workspace, say the scratch area, the developer of the entity can conduct development process, and in another part of the workspace, say the product release area, new versions of an entity are released. The introduction of such workspace will be a vital step to reduce unexpected side-effects in the system and to improve the ease of collaboration among Process Handbook users.

Finally, the issue of how access control relations are to be displayed at the front end would require attention. For example, should ACL objects be displayed as navigational nodes? Under this model, all access control information can be found under a single navigational root for access control information. However, how should the navigational hierarchy for access control information be organized? Clearly, these questions have to be resolved to determine the optimal model for displaying and maintaining access control information. Without a friendly and expressive user interface, the usefulness of the access control objects would be greatly hindered.

Chapter 8: Conclusion

This research aims at establishing an appropriate access control scheme for the transition of the Process Handbook to a multi-user server. While protecting sensitive information stored in the system is a major objective of the required access control policy, it is even more important that by integrating the access control policy, users of the system can engage in new forms of collaboration, and the system could provide new perspectives of the stored information. With the addition of the new first-class entities, i.e., user groups and entity collections, together with the flexible access control policy proposed and implemented in this research, these objectives can be met.

As discussed in Section 6.2, authentication and message exchanges layered on top of a secure protocol provides adequate protection to the system in a distributed environment. Consequently, access control can be executed with the faith that the user who requested the access represents the actual user. Moreover, the fine granularity of control and the relaxation of management requirements greatly enhance the flexibility of the system. As users are free to manipulate the organization of user groups and entity collections, there are tremendous rooms for designing an organization which satisfies a specific access control requirement and can be easily maintained. The flexibility of the model to satisfy different combinations of access requirements allows different forms of collaboration to be engaged among the users. Among the possible combinations, many bear interesting implications. Furthermore, by combining the idea of version control and incremental disclosure of information, an entity workspace can be established, and the system would become an area of integrated workspaces. Finally, as entities can be organized into entity collection hierarchy in addition to other existing relational hierarchies, the system provides an addition dimension where users can view the entities. Such new perspective would be valuable to many users.

As a summary, a reiteration of the outcome of this thesis would be in order. In this thesis, a sensible access control policy, within the Process Handbook object API abstraction level, has been identified and implemented. Moreover, extensions and rules relevant to the access control policy of the server API are proposed. As a result of the research, users can benefit from an increase in the ease of control over the design of process organization and relationship. While new forms of collaboration are made possible, users can enjoy viewing the content of the Process Handbook from new perspectives. Moreover, unexpected alterations of information stored in the system can be reduced, and unauthorized access to the information can be guarded.

Appendix A

A.1 Specification of Access Control Manager Class

A.1.1 Properties

None.

A.1.2 Methods

GetUserPrivilege(MyUserGroupID As Long, myEntityID As Long, myACLType As Integer, myStatus) As Integer

Function: This method returns the privilege of type myACLType for myEntityID of the user group specified by myUserGroupID. If the identifiers are invalid, myStatus would be set to error code PH_AC_ERROR_INVALID_PARAMETER. Note: myACLType can be PH_AC_ENTITY, PH_AC_DOMAIN, PH_AC_ENTITY_MANAGEMENT, PH_AC_DOMAIN_MANAGEMENT; PH_AC_ALL is not allowed

GetPrivilegedGroups(MyUserGroupID As Long, myEntityID As Long, myPrivilege As Integer, ByRef myStatus)

Function: This method returns an array of group IDs whose authorization level is equal or greater than myPrivilege for the user groups specified by myUserGroupIDs. This method assumes that the myEntityID are of type = PH_AC_ENTITY. If the identifiers are invalid, myStatus would be set to error code PH_AC_ERROR_INVALID_PARAMETER. If the privileged group is empty, the first element of the returned array will be negative.

InitNotify(myEntityID As Long, myManagerID As Variant, ByRef myStatus)

Requirement: myEntityID must be valid, myManagerID is an array of long numbers.
Modify: ACRelation Table
Function: Create the ACL corresponding to myEntityID by adding the user groups with the maxium privileges, specified by myManagerID to the ACRelation table. If any of the myMangerID is invalid, myStatus will be set to error code PH_AC_ERROR_INVALID_PARAMETER.

CleanUpNotify(myID As Long, myEntityType As Integer, ByRef myStatus)

Function: Remove all relations relevant to myID which is of type specified by myEntityType. If any of the parameters are invalid, myStatus is set to error code PH_AC_ERROR_INVALID_PARAMETER.

ChangeMemberNotify(myID As Long, myEntityType As Integer, ByRef myStatus)

Function: Remove all internal information related to the changed relation If any of the para meters are invalid, myStatus is error code PH_AC_ERROR_INVALID_PARAMETER.

A.2 Specification of ACL Class

A.2.1 Properties

Property	*Value Data Type*	*Status*
EntityID	Long	Can be Read
ACLType	Integer	Can be Read
EOF	Boolean	Can be Read

A.2.2 Methods

Init(myEntityID As Long, myACLType As Integer, myStatus)

Function: This method initializes the object by retrieving the corresonding ACL from the database. If myEntityID, myACLType invalid, then ACL is empty, and myStatus is set to error code PH_AC_ERROR_INVALID_PARAMETER; else, the object is initialized. Finally, if no groups has the maximum management right in the ACL, PH_SYSTEM_MANAGER will be given such right.

GetFirstEntry(ByRef myStatus, mySessionID as String) As ACLEntry

Function: This method returns the first ACLEntry object in the ACL object. If the ACL is not init, myStatus will be set to error code PH_AC_ERROR_NOT_INIT, and GetFirstEntry will be an uninitialized ACLEntry. If access privilege is not satisfied, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED. If the ACL is empty, myStatus will be set to error code PH_AC_ERROR_EOF. Note: All properties of an uninitialized ACLEntry will be negative.

GetNextEntry(ByRef myStatus, mySessionID as String) As ACLEntry

Function: This method returns the next ACLEntry object in the ACL object. If the ACL is not init, myStatus will be set to error code PH_AC_ERROR_NOT_INIT, and GetNextEntry will be set to an uninitialized ACLEntry. If access privilege is not satisfied, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED If ACL is empty or at the end already, myStatus will be set to error code PH_AC_ERROR_EOF.

AddEntry(myEntry As ACLEntry, ByRef myStatus mySessionID as String)

Modify: ACRelation Table
Function: This method updates the ACRelaion table of the handbook database to add the new ACLEntry to the current object and other entities inheriting access control restrictions from this object. If the ACL is not initialized, myStatus would be set to error code PH_AC_ERROR_NOT_INIT; If any of the parameters are invalid, myStatus would be set to error code PH_AC_ERROR_INVALID_PARAMETER. If myUserGroupID does not have enough privilege, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED.

RemoveEntry(myEntry As ACLEntry, ByRef myStatus mySessionID as String)

Modify: ACRelation Table

Function: This method updates the ACRelaion table of the handbook database to remove myEntry from the current object and other inherited access control restricitons. If any of the parameters are invalid, myStatus would be set to error code PH_AC_ERROR_NOT_INIT. If myUserGroupID does not have enough privilege, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED. After the entry is found and remove, the entry is set to the first entry in the list.

Append(mySource As ACL, ByRef myStatus, mySessionID as String)

Modify: ACRelation Table

Function: This method updates the ACRelaion table of the handbook database to append the ACL of mySource. If ACLs are not initialized, myStatus would be set to error code PH_AC_ERROR_NOT_INIT. If myUserGroupID does not have the maximum management right, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED.

AddInherit(mySourceID As Long, myStatus, mySessionID as String)

Function: Create an access control inheritance relationship between mySource and the current ojbect. If ACLs are not initialized, myStatus would be set to error code PH_AC_ERROR_NOT_INIT. If myUserGroupID does not have the maximum management right, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED.

RemoveInherit(mySourceID As Long, myStatus, mySessionID as String)

Function: Remove the access control inheritance relationship between mySource and the current ojbect. If ACLs are not initialized, myStatus would be set to error code PH_AC_ERROR_NOT_INIT. If myUserGroupID does not have the maximum management right, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED.

A.3 Specification of CapL Class

A.3.1 Properties

Property	*Value Data Type*	*Status*
UserGroupID	Long	Can be Read
EOF	Boolean	Can be Read

A.3.2 Methods

Init(myEntityID As Long, myStatus)

Function: This method initializes the object by retrieving the corresponding data from the database. If myEntityID is invalid, then the object is empty, and myStatus is set to error code PH_AC_ERROR_INVALID_PARAMETER; else, the object is initialized.

GetFirstEntry(ByRef myStatus, mySessionID as String) As CapLEntry

Function: This method returns the first CapLEntry object in the CapL object. If the CapL is not init, myStatus will be set to error code PH_AC_ERROR_NOT_INIT, and GetFirstEntry will be an uninitialized CapLEntry. If access privilege is not satisfied, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED. If the CapL is empty, myStatus will be set to error code PH_AC_ERROR_EOF. Note: All properties of an uninitialized CapLEntry will be negative.

GetNextEntry(ByRef myStatus, mySessionID as String) As CapLEntry

Function: This method returns the next CapLEntry object in the CapL object. If the CapL is not init, myStatus will be set to error code PH_AC_ERROR_NOT_INIT, and GetNextEntry will be set to an uninitialized CapLEntry. If access privilege is not satisfied, or user not authenticated, myStatus will be set to error code PH_AC_ERROR_ACCESS_DENIED If CapL is empty or at the end already, myStatus will be set to error code PH_AC_ERROR_EOF.

AddInherit (mySourceID As Long, myStatus, mySessionID as String)

Function: Create an access control inheritance relationship between mySource and the current object. If *UserGroupID* does not have the maximum management right, or user not authenticated, *Status* will be set to error code PH_AC_ERROR_ACCESS_DENIED. If any of the parameters are invalid, or if the user group associated with the current object is not a specialization of the user group represented by mySource, myStatus will be set to error code PH_AC_ERROR_INVALID_PARAMETER

RemoveInherit (mySourceID As Long, myStatus, mySessionID as String)

Function: Remove the access control inheritance relationship between mySource and the current object. If *UserGroupID* does not have the maximum management right, or user not authenticated, *Status* will be set to error code PH_AC_ERROR_ACCESS_DENIED. If any of the parameters are invalid, or if the user group associated with the current object is not a specialization of the user group represented by mySource, myStatus will be set to error code PH_AC_ERROR_INVALID_PARAMETER

A.4 Specification of ACLEntry Class

A.4.1 Properties

Property	*Value Data Type*	*Status*
UserGroupID	Long	Can be Read and Updated
ACLType	Integer	Can be Read and Updated
Permission	Integer	Can be Read and Updated

A.4.2 Methods

None

A.5 Specification of CapLEntry Class

A.5.1 Properties

Property	*Value Data Type*	*Status*
EntityID	Long	Can be Read and Updated
ACLType	Integer	Can be Read and Updated
Permission	Integer	Can be Read and Updated

A.5.2 Methods

None

A.6 Specification of MD5Msg Class

A.6.1 Properties

Property	*Value Data Type*	*Status*
NumByte	Integer	Can be Read and Updated
MsgByte	Byte	Can be Read and Updated

A.6.2 Methods

Init(NumByte As Integer)

Function: Reallocate dynamic array to size NumByte

CreateCopy() As MD5Msg

Function: Copy the content of the Object to a new MD5Msg Object

ShiftL(n As Integer)

Function: Shift the Object to Left by n bit

ShiftR(n As Integer)

Function: Shift the Object to right by n bit

RotL(n As Integer)

Function: Rotate the Object to Left by n bit

RottR(n As Integer)

Function: Rotate the Object to right by n bit

Copy(mySource As MD5Msg)

Function: Copy the content of mySource to the Object

MSB( ) As Integer

Function: Return the location of MSB in the fixed size byte chunks starting from 1

Negate( )

Function: Negate d_byte of the Object

Equal(myMsg As MD5Msg) As Boolean

Function: Return True if myMsg is identical to the object

A.7 Specification of MD5Engine Class

A.7.1 Properties

Property	*Value Data Type*	*Status*
NumByte	Integer	Can be Read and Updated
MsgByte	Byte	Can be Read and Updated

A.7.2 Methods

Str2MD5Msg(myStr As String) As MD5Msg

Requirement: Length(myStr) <= 64 characters
Function: Convert a string a MD5Msg of 64 byte chunks. If myStr is empty or more than 64 characters then return a MD5Msg containing 0.

MD5Msg2Str(myMsg As MD5Msg) As String

Function: Convert myMsg into a string

Hash(myInput As MD5Msg) As MD5Msg

Requirement: myInput must have 64 byte chunks, and have less than 448 bits.
Function: Pad myInput to 448 bits and then append 64 LSB of myInput to form 512 bit message. Transform the message in 4 rounds, and return the result.

A.8 Specification of AuthMgr Class

A.8.1 Properties

None

A.8.2 Methods

Authenticate(mySessionID as String) as Long

Function: Check the parameters against the record stored in the object. If equal, the function return the user group id of the corresponding user group, else -1 is returned.

CreateSession (myUserGroupID as Long, myPassword as String, myStatus) as Long

Function: Hashes the password using MD5 and compare to the one stored in the database for user group, myUserGroupID. If equal, return a session object, and maintain it in a collection object; else, myStatus is set to be an error with code number PH_AC_ERROR_AUTHENTICATION_FAIL, and the returned object will be pointed to NOTHING.

CloseSession(mySessionID as String, Status)

Function: Remove the session object that corresponds to the parameters. If such object does not exist, Status will be set to an error with code number PH_AC_ERROR_OBJ_NOT_EXIST. Otherwise, the object will be removed from the collection.

CloseAllSessions( )

Function: Remove all the session objects maintained in the object.

A.9 Specification of Session Class

A.5.1 Properties

Property	*Value Data Type*	*Status*
SessionID	String	Can be Read and Updated
DateTime	Date	Can be Read and Updated

A.5.2 Methods

None

Appendix B

B.1 Table of Privilege Constants

Privilege	*Constant*
Read	PH_AC_READ_PRIVILEGE
Create	PH_AC_CREATE_PRIVILEGE
Add Relation	PH_AC_ADD_RELATION_PRIVILEGE
Remove Relation	PH_AC_REMOVE_RELATION_PRIVILEGE
Edit	PH_AC_EDIT_PRIVILEGE
Move	PH_AC_MOVE_PRIVILEGE
Delete	PH_AC_DELETE_PRIVILEGE
Grant Read	PH_AC_READ_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Read	PH_AC_READ_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Create	PH_AC_CREATE_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Create	PH_AC_CREATE_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Add Relation	PH_AC_ADD_RELATION_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Add Relation	PH_AC_ADD_RELATION_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Remove Relation	PH_AC_REMOVE_RELATION_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Remove Relation	PH_AC_REMOVE_RELATION_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Edit	PH_AC_EDIT_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Edit	PH_AC_EDIT_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Move	PH_AC_MOVE_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Move	PH_AC_MOVE_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Grant Delete	PH_AC_DELETE_PRIVILEGE - PH_AC_MANAGEMENT_PRIVILEGE_OFFSET
Revoke Delete	PH_AC_DELETE_PRIVILEGE + PH_AC_MANAGEMENT_PRIVILEGE_OFFSET

B.2 Error Codes

Error Description	*Value*
Invalid Parameter	PH_AC_ERROR_INVALID_PARAMETER
Not Initialized	PH_AC_ERROR_NOT_INIT
Access Denied	PH_AC_ERROR_ACCESS_DENIED
End of File/Record Encountered	PH_AC_ERROR_EOF
Object not Found	PH_AC_ERROR_OBJ_NOT_EXIST
Authentication Fail	PH_AC_ERROR_AUTHENTICATION_FAIL

B.3 Other Global Constants

Variable Description	*Value*
Entity ID of Entity Universe	PH_ENTITY_UNIVERSE
Entity ID of User Universe	PH_USER_UNIVERSE
Entity ID of System Manager User Group	PH_SYSTEM_MANAGER
Type ID for User Groups	PH_AC_USER_GROUP
Type ID for Entity Collections	PH_AC_ENTITY_COLLECTION
Type ID for Other First-class Entities	PH_AC_NORMAL_ENTITY

Appendix C

C.1 Integration of Access Control Mechanism

C.1.1 Parameter Specification

EntityObject.MethodName(OwnParameters, ACParameters)

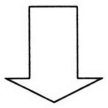
Where:

EntityObject refers to an instance of the Entity class;

MethodName refers to a method of the Entity class;

OwnParameters refers to the non-access control parameters required;

ACParameters refers to the access control parameters required. It consists of the session identifier of the exported session object.



EntityObject.MethodName(OwnParameters, SessionID as String)

Where:

SessionID refers to the session identifier in the session object exported to the user;

C.1.2 Access Control Procedures

C.1.2.1 Authentication Specification

In each method of the Entity class, it has to perform user authentication, the specification for the authentication mechanism is shown below:

Specification:

AuthMgrObj.Authenticate(*SessionID*) as Long

Function: Check the parameters against the record stored in the object. If equal, the function return the user group id of the corresponding user group, else -1 is returned.

Usage:

If *AuthMgrObj*.CheckAuth(*SessionID*) = -1Then

....Authentication Fail

End If

Where:

AuthMgrObj refers to an instance of the AuthMgr class;

SessionID refers to the session identifier in the session object exported to the user;

C.1.2.2 Access Checking Specification

In addition to authentication checking, each method of the Entity class checks the privileges possessed by the authenticated user. The specification of the access checking mechanism is shown below:

Specification:

ACMgrObj.GetPrivilege(*UserGroupID*, *EntityID*, *ACLType*, *Status*) as Integer

Function: This method returns the privilege of type *ACLType* for *EntityID* of the user group specified by *UserGroupID*. If the identifiers are invalid, *Status* would be set to be an eror with code number PH_AC_ERROR_INVALID_PARAMETER. Note: *ACLType* can be PH_AC_ENTITY, PH_AC_DOMAIN, PH_AC_ENTITY_MANAGEMENT, PH_AC_DOMAIN_MANAGEMENT; PH_AC_ALL is not allowed

Usage:

If *ACMgrObj*.GetPrivilege(*UserGroupID*, *EntityID*, *ACLType*, *Status*) < *RequiredPrivilege* Then

....Access Denied

End If

Where:

ACMgrObj refers to an instance of the ACMgr class;

UserGroupID refers to the user group identifier of the authenticated user;

EntityID refers to the entity identifier of the entity requesting access checking.

ACLType refers to the type of the access control relation. *ACLType* can be PH_AC_ENTITY, PH_AC_DOMAIN, PH_AC_ENTITY_MANAGEMENT, or PH_AC_DOMAIN_MANAGEMENT

RequiredPrivilege refers to the minimum privilege required for executing the method, as identified below.

Status stores any error occurred during access checking. If any of the identifiers are invalid, *status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

C.1.3 Initialization

C.1.3.1 Initializing Session Objects for Authentication

Users are given a session object when they log onto the system. The specification for the creation of session objects is given below:

Specification:

AuthMgrObj.CreateSession(*UserGroupID, Password, Status*) as Session

Function: Hashes the password using MD5 and compare to the one stored in the database for user group, *UserGroupID*. If equal, return a session object, and maintain it in a collection object; else, *Status* is set to be an error with code number PH_AC_ERROR_AUTHENTICATION_FAIL, and the returned object will be pointed to NOTHING.

Usage:

Set *mySessionObj* = *AuthMgrObj*.CreateSession(*UserGroupID, Password, Status*)

Where:

mySessionObj refers to the variable which would be assigned the new Session object.

AuthMgrObj refers to an instance of the AuthMgr class;

UserGroupID refers to the user group identifier of the user;

Password refers to the password for the user group;

Status stores any error occurred during the creation of the session object. If the password does not correspond to that of the user group specified by *UserGroupID, status* will be an error with code number PH_AC_ERROR_AUTHENTICATION_FAIL.

C.1.3.2 Initializing Users with Maximum Management Privilege

When a new entity is created, users with the maximum management privilege can be assigned through the InitNotify method of the ACMgr class. The specification of the InitNotify method is shown below:

Specification:

ACMgrObj.InitNotify (*EntityID, ManagerID, Status*)

Require: *EntityID* must be valid, *ManagerID* is an array of long numbers.
Modify: ACRelation Table
Function: Create the ACL corresponding to *EntityID* by adding the user groups with the maximum privileges, specified by *ManagerID* to the ACRelation table. If any of the *MangerID* is invalid, *Status* will be set to an error with code number PH_AC_ERROR_INVALID_PARAMETER.

Usage:

ACMgrObj.InitNotify *EntityID, ManagerID, Status*

Where:

ACMgrObj refers to an instance of the ACMgr class;

UserGroupID refers to the user group identifier of the authenticated user;

ManagerID refers to an array of identifiers, representing the users to be assigned the maximum management privilege.

EntityID refers to the entity identifier of the entity to be initialized;

Status stores any error occurred during access checking. If any of the identifiers are invalid, *status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

C.1.3.3 Initializing Entity to Inherit Access Control Restrictions

An entity can inherit a specific type of access control restrictions from another entity if both the source entity and the destination entity support the type. The specification of the access control restrictions inheritance mechanism is shown below:

Specification:

ACLObj.AddInherit (*SourceEntityID, ACLType, Status, SessionID*)

Function: Create an access control inheritance relationship between *SourceEntityID* and the current object. If the object is not initialized, *Status* would be set to error code PH_AC_ERROR_NOT_INIT. If the corresponding user group of the session does not have the maximum management right, or user not authenticated, *Status* will be set to error code PH_AC_ERROR_ACCESS_DENIED. If any of the parameters are invalid, *Status* will be set to error code PH_AC_ERROR_INVALID_PARAMETER; Note: if the current object is of type PH_AC_ALL,*ACLType* can be PH_AC_ALL, PH_AC_ENTITY, PH_AC_DOMAIN, PH_AC_ENTITY_MANAGEMENT, and PH_AC_DOMAIN_MANAGEMENT. Otherwise, *ACLType* must be the same as the type of the object.

Usage:

ACLObj.AddInherit *SourceEntityID, Status, SessionID*

Where:

ACLObj refers to an instance of the ACL class;

SourceEntityID refers to the entity identifier of the parent entity;

ACLType refers to the type of access control relations. It can be PH_AC_ALL, PH_AC_ENTITY, PH_AC_DOMAIN, PH_AC_ENTITY_MANAGEMENT, or PH_AC_DOMAIN_MANAGEMENT

Status stores any error occurred during access checking. If any of the identifiers are invalid, *status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER. If the user group specified by *UserGroupID* does not have the privilege to inherit access control restrictions, *Status* will be an error with code number PH_AC_ERROR_ACCESS_DENIED.

SessionID refers to the session identifier in the session object exported to the user;

C.1.3.4 Initializing User Group to Inherit Privileges in Specialization

A user group can inherit the privileges given to its parent in the specialization hierarchy. The specification of the privilege inheritance mechanism is shown below:

Specification:

CapLObj. AddInherit (Source*EntityID, Status, SessionID*)

Function: Create an access control inheritance relationship between *SourceEntityID* and the current object.. If any of the parameters are invalid, or if the associated user group of the current object is not a specialization of the user group represented by *SourceEntityID, Status* will be set to an

error with code number PH_AC_ERROR_INVALID_PARAMETER. If the corresponding user group of the session does not have the maximum management right, or user not authenticated, *Status* will be set to error code PH_AC_ERROR_ACCESS_DENIED.

Usage:

CapLObj.AddInherit Source*EntityID*, *Status, SessionID*

Where:

CapLObj refers to an instance of the CapL class;

SourceEntityID refers to the entity identifier of the parent entity;

Status stores any error occurred during access checking. If any of the identifiers are invalid, S*tatus* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

SessionID refers to the session identifier in the session object exported to the user;

C.1.4 Clean Up

C.1.4.1 Entity Removal

When an entity is being removed from the system, a RemoveNotify message should be sent to an ACMgr object. The specification is shown below:

Specification:

ACMgrObj. RemoveNotify (*EntityID*, *EntityType*, *Status)*

Function: Remove all relations relevant to *EntityID* which is of type specified by *EntityType*. If any of the parameters are invalid, *Status* will be set to an error, with code number PH_AC_ERROR_INVALID_PARAMETER.

Usage:

ACMgrObj.RemoveNotify *EntityID*, *EntityType, Status*

Where:

ACMgrObj refers to an instance of the ACMgr class;

EntityID refers to the entity identifier of the entity being removed;

EntityType refers to the entity type of the entity being removed; If entity represents a user group, *EntityType* is PH_AC_USER_GROUP; if entity represents an entity collection, *EntityType* is PH_AC_ENTITY_COLLECTION; otherwise, *EntityType* is PH_AC_NORMAL_ENTITY.

Status stores any error occurred during access checking. If any of the identifiers are invalid, *Status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

C.1.4.2 Membership Change

When a membership relation is being added or removed, a ChangeMemberNotify message should be sent to an ACMgr object. The specification is shown below:

Specification:

ACMgrObj. ChangeMemberNotify (*EntityID*, *EntityType*, *Status)*

Function: Remove all relations relevant to *EntityID* which is of type specified by *EntityType*. If any of the parameters are invalid, *Status* will be set to an error with code number PH_AC_ERROR_INVALID_PARAMETER.

Usage:

ACMgrObj. ChangeMemberNotify *EntityID*, *EntityType, Status*

Where:

ACMgrObj refers to an instance of the ACMgr class;

EntityID refers to the entity identifier of the child entity of the membership relation being removed;

EntityType refers to the entity type of the entity being removed; If entity represents a user group, *EntityType* is PH_AC_USER_GROUP; if entity represents an entity collection, *EntityType* is PH_AC_ENTITY_COLLECTION; otherwise, *EntityType* is PH_AC_NORMAL_ENTITY.

Status stores any error occurred during access checking. If any of the identifiers are invalid, *Status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

C.1.4.3 User Log Out

When a user logs out of the system, the session object associated with the session should also be removed from memory. The specification for the cleanup process is given below:

Specification:

AuthMgrObj.CloseSession(*SessionID, Status*)

Function: Remove the session object that corresponds to the parameters. If such object does not exist, *Status* will be set to an error with code number PH_AC_ERROR_OBJ_NOT_EXIST. Otherwise, the object will be removed from the collection.

AuthMgrObj.CloseAllSessions( )

Function: Remove all the session objects maintained by the object.

Usage:

AuthMgrObj.CloseSession *SessionID, Status*

AuthMgrObj.CloseAllSessions

Where:

AuthMgrObj refers to an instance of the AuthMgr class;

SessionID refers to the session identifier in the session object exported to the user;

Status stores any error occurred during the removal of the session object. If the object does not exist, *Status* will be an error with code number PH_AC_ERROR_OBJ_NOT_EXIST.

C.1.4.4 Removal of Inheritance Relations

Inheritance relations can be removed by calling the appropriate procedures. The specification of removing inheritance relations are shown below.

Specification:

Obj. RemoveInherit (Source*EntityID, Status, SessionID*)

Function: Remove the Inherited given to the user group, *SourceEntityID*, and other relevant objects. If any of the parameters are invalid, or if the associated user group of the current object is not a specialization of the user group represented by *SourceEntityID, Status* will be set to an error with code number PH_AC_ERROR_INVALID_PARAMETER. If the corresponding user group of the session does not have the maximum management right, or user not authenticated, *Status* will be set to error code PH_AC_ERROR_ACCESS_DENIED.

Usage:

Obj.RemoveInherit Source*EntityID, Status, SessionID*

Where:

Obj refers to an instance of the CapL or ACL class;

SourceEntityID refers to the entity identifier of the parent entity;

Status stores any error occurred during access checking. If any of the identifiers are invalid, *Status* will be an error with code number PH_AC_ERROR_INVALID_PARAMETER.

SessionID refers to the session identifier in the session object exported to the user;

Appendix D

D.1 Usage Scenario for the Process Handbook

In the evaluation of the access control policy, a preliminary Process Handbook editorial methodology has been established. In particular, roles for the normal access of the entities have been defined, and each role owns a different set of entity access control privileges. In Figure 18, the different roles and their corresponding responsibilities are presented. While the "User" role has the least responsibility, the "Administrator" role has the most. As an interesting note, whereas each entry in the Handbook may have a different author, many related entries would be monitored by the same editors. In addition, the area of access of the entries in the Handbook can be divided into "Public", "Own", "Group", and "All". Specifically, the "Public" area would contain only the entries readable by the general, and the "All" area would contain all entries exist in the Handbook. Furthermore, while the entries in the "Own" area would be maintained by a single user, the entries in the "Group" area would be maintained by a group of users. Users allowed to access the more restrictive areas, such as the "All" area, would also be allowed to access the more public areas, such as the "Public" area,.

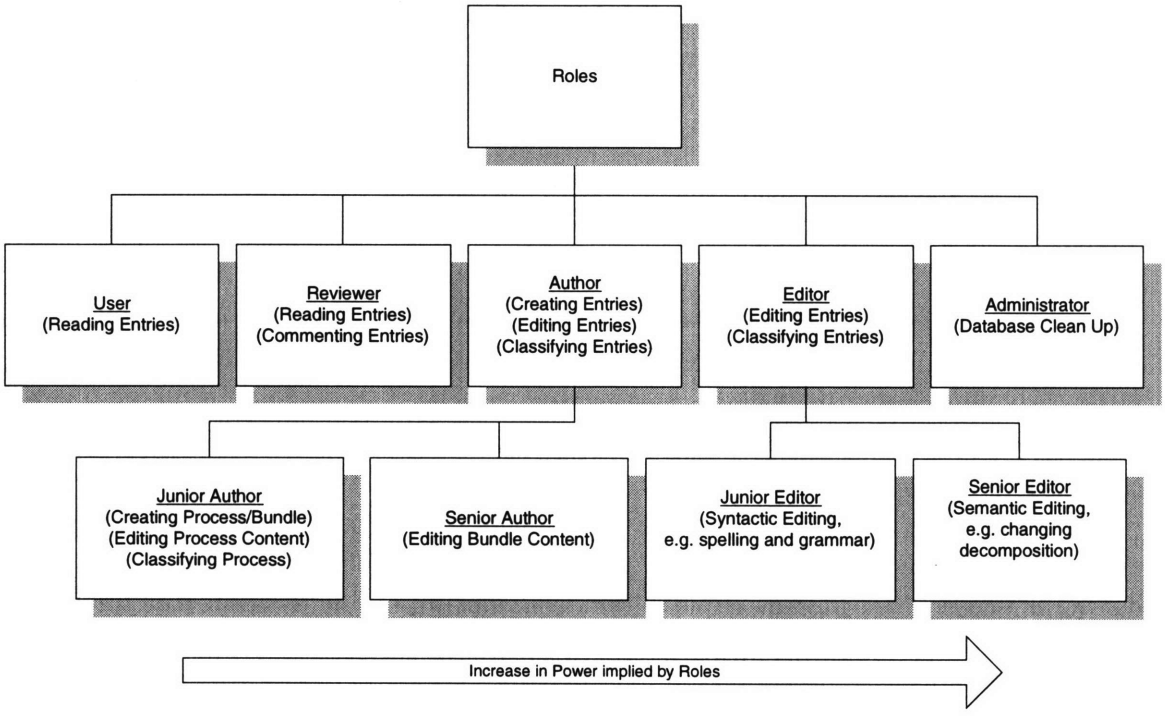


Figure 18: Decomposition of Roles. The parentheses in each box contain the responsibilities of the corresponding role.

Orthogonal to the concept of roles and areas of access is the idea of review status. When the status of the entries changes, the privileges given to each role also alter. As shown in Table 2, when the status of an entry is "draft", users in the "Author" role are allowed to change the entry whereas users in the "Editor" role are not.

Table 2: Impact of Review Status on Role Responsibilities. Writing corresponds to reading, editing, and classifying entries.

	Draft	**Review**	**Registered**
User	None	None	Read
Reviewer	None	Read	Read
Author	Write	Read	Read
Editor	Read	Write	Write

D.2 Using the Access Control Policy

By making use of entity collections and their interrelationship, the editorial methodology described in the preceding section can be supported. In particular, the various areas of access would directly map to various entity collections. As a result, all public entries would be members of the "Public" entity collection. Moreover, while all entries maintained by a single user would be members of the "Own" collection, all entries maintained by a group of users would be members of the "Group" collection. Finally, the "All" collection will map to the "All" area, containing all relevant entries in the Handbook as members.

In the given scenario, users allowed to access the members of more restrictive areas are also allowed to access the members of more public areas. To satisfy the above requirement, one can make use of the membership relation among entity collections[32]. By making a public collection the member of a more restrictive collection, domain access control restrictions specified in the restrictive collection would also be applicable to the public one. Thus, users who are given the read privilege in the domain of the most restrictive collection, i.e., the "All collection", could read any entities in the system[33]. Following this strategy, the "Public" collection should be a member of the "Own" collection. Moreover, the "Own" collection should be a member of the "Group" collection, which in turn should be a member of the "All" collection.

---

[32] Recall that domain access control restrictions of an entity collection applies to its members.

[33] The "All" collection is the most restrictive collection, and the "Public" collection is the most public collection. Moreover, the "Own" collection is more public than the "Group" collection.

On the other hand, to support the ability to change user privileges according to review status, each of the preliminary collections defined should be subdivided so that each of the new collections would contain entries of a particular review status. As a result, by designating different domain access restrictions to collections containing entries with different review status, the privileges given to users of different roles can be updated when the status of an entry changes, i.e., when the entry is moved from one collection to another.

In addition, because senior authors are allowed to edit the content of bundles while junior authors are not allowed, the organization of the entity collection has to be further modified by separating activities from bundles. The final organization[34] of the entity collection is shown in Figure 19.

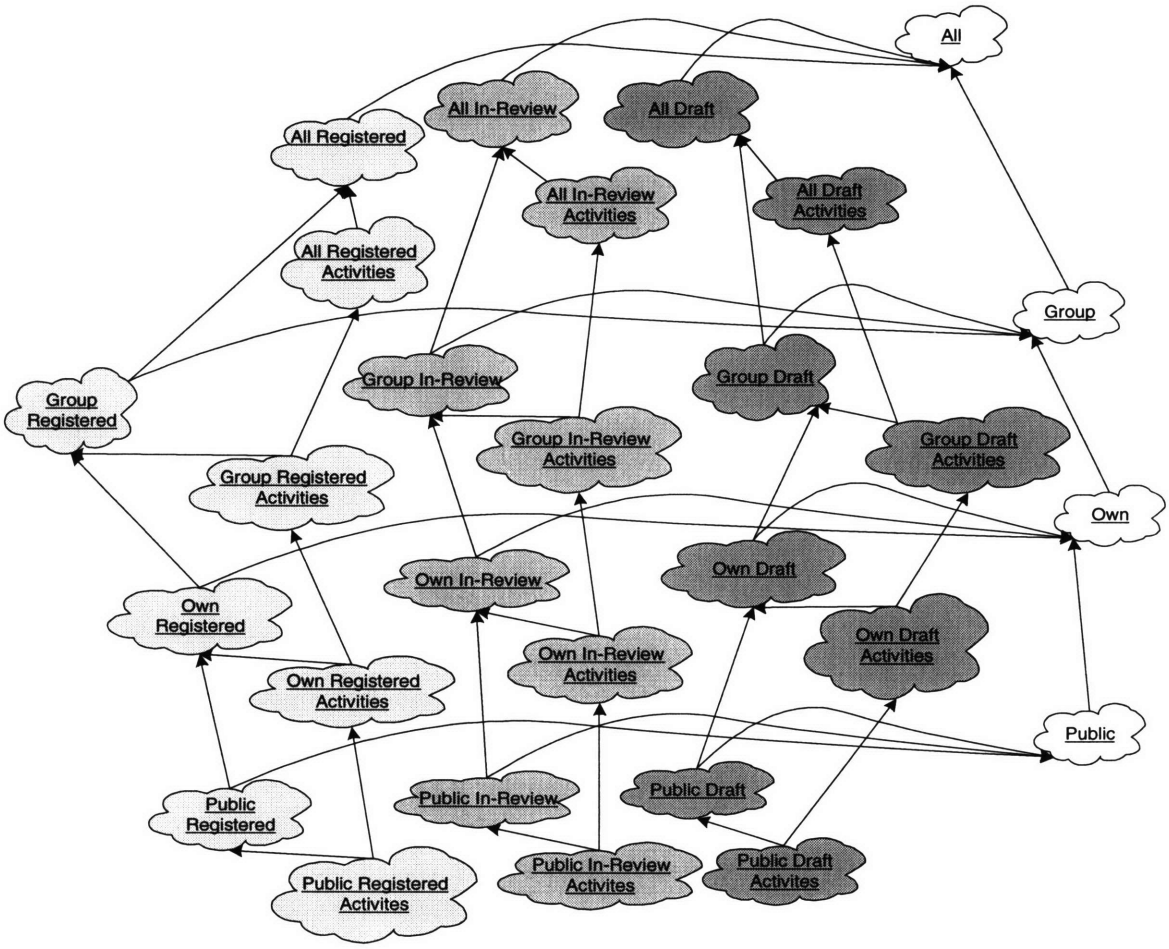


Figure 19: Membership Relation among Entity Collections. An arrow represents a membership relation. For viewing convenience, "Entity Universe" is not shown.

---

[34] When a specialization is created in a particular entity collection, the access control inheritance mechanism will automatically grant the appropriate membership relations to the new entity. In this way, the domain access control relations in each entity collection will be applicable to any new entries.

To allow different roles to possess different responsibilities, each role is mapped to a different set of access privileges. However, the distinction between syntactic and semantic editing is not supported by the proposed access control policy. Otherwise, the complexity of the access control model will increase drastically, as the privilege levels could no longer form an ordered list. As a result, instead of using the proposed model, the distinction will be made through social means. In other words, junior editors would be trusted to act appropriately and not to modify the decomposition of the entries. InTable 3, the mapping between user roles and privileges is shown.

Finally, changing review status as well as placing bundles and activities into the appropriate collections have to be done manually by changing the members of the entity collections. In particular, the designated users would have the revoke delete privilege and the remove relation privilege in the relevant entity collections. For example, to manage the members of the "Public" area, the designated managers should have the revoke delete privilege and the remove relation privilege in the domain of the "Public" collection.

On the other hand, access control privileges can be maintained in two ways. In one way, the responsible users would be given the revoke delete privilege in the relevant domain to perform access control management. Alternatively, one could also make use of user groups to facilitate the maintenance of access control restrictions. In this way, the domain privileges for a particular user role in a given entity collection would be granted to a fixed user group. For example, the privileges for the "Editor" role in the "Public Draft" domain is only given to the "Public Draft Editor" user group. Through adding and removing members in the "Public Draft Editor" user group, the authorized editors in the "Public Draft" domain can be specified.

Table 3: Mapping between Roles, Review Status, and Privileges.

Review Status	Roles	Privilege Requirements
Draft	User	No Privilege in Draft Domain
In-Review	User	No Privilege in In-Review Domain
Registered	User	Read Privilege in Registered Domain
Draft	Reviewer	No Privilege in Draft Domain
In-Review	Reviewer	Read Privilege in In-Review Domain
Registered	Reviewer	Read Privilege in Registered Domain
Draft	Junior Author	Delete Privilege in Draft Activities Domain
In-Review	Junior Author	Read Privilege in In-Review Domain
Registered	Junior Author	Read Privilege in Registered Domain
Draft	Senior Author	Delete Privilege in Draft Domain
In-Review	Senior Author	Read Privilege in In-Review Domain
Registered	Senior Author	Read Privilege in Registered Domain
Draft	Editor	Read Privilege in Draft Domain
In-Review	Editor	Delete Privilege in In-Review Domain
Registered	Editor	Delete Privilege in Registered Domain

Bibliography

1. MALONE, T. W., CROWSTON, K., LEE, J., PENTLAND, B., "Tools for inventing organizations: Toward a handbook of organizational processes", CCS WP #141, Sloan School WP # 3562-93, 1993
2. SARIN, S., GRIEF, I., "Computer-based Real-Time Conferencing Systems", in "Computer-Supported Cooperative Work: A book of Readings", edited by, GRIEF, I., Morgan Kaufmann Publishers, Inc., 1988
3. SARIN, S., GRIEF, I., "Data Sharing in Group Work", in "Computer-Supported Cooperative Work: A book of Readings", edited by, GRIEF, I., Morgan Kaufmann Publishers, Inc., 1988
4. BRINKLEY, D., SCHELL, R., "Concepts and Terminology for Computer Security", in "Information Security: An integrated collection of essays", edited by ABRAMS, JAJODIA, PODELL, IEEE Computer Society Press, 1995
5. JADODIA, S., KOGAN, B., SANDHU, R., "A Multilevel Secure Object-Oriented Data Model" , in "Information Security: An integrated collection of essays", edited by ABRAMS, JAJODIA, PODELL, IEEE Computer Society Press, 1995
6. MCOLLUM, C. J., MESSING, J. R., NOTAGIACOMO, L., "Beyond the Pale of MAC and DAC-Defining New Forms of Access Control", in "1990 IEEE Symposium On Security and Privacy", 1990
7. LINDSAY, D. T., PRICE, W. L., "Information Security", Elsevier Science Publishers B. V., 1991
8. CLARK, D., REDELL, D., "Protection of Information in Computer Systems", IEEE Computer Society, 1975
9. SALTZER, SCHROEDER, "Protection of Information in Computer Systems", IEEE Computer Society, 1975
10. "Information Technology Security Evaluation Criteria", Der Bundesminiter des Innern, Bonn, Germany, 1990
11. FERNANDEZ, E. B., LARRONDO-PETRIE, M. M., GUDES, E., "A Method-based Authorization Model for Object-Oriented Databases", in "Security for Object-Oriented Systems", edited by THURAISINGHAM, B., SANDHU, R., TING, T. C., Spring-Verlag, 1993
12. TANENBAUM, A., "Modern operating Systems", Prentice Hall, 1992
13. FERNANDEZ, E. B., SUMMERS, R. C., WOOD, C., "Database Security and Integrity", Addison-Wesley, 1981.
14. GIURI, L., IGLIO, P., "A formal Model For Role-Based Access Control with Constraints" in "9th IEEE Computer Security Foundations Workshop", IEEE Computer Society Press, 1996.
15. "Computer Security - ESORICS 96" edited by BERTINO, E., KURTH, H., MARTELLA, G., MONTOLIVO, E., Springer, 1996.
16. "Computer Security - ESORICS 92" edited by DESWARTE, Y., EIZENBERG, G., QUISQUATER, J.-J., Springer-Verlag, 1996.
17. "1996 IEEE Symposium on Security and Privacy", IEEE Computer Society Press, 1996.
18. HICKMAN, K., "The SSL Protocol", http://home.netscape.com/newsref/std/SSL_old.html
19. RIVEST, R., "The MD5 Message-Digest Algorithm
20. FAROOQ, U., "An Object Server for the Process Handbook", Master Thesis, EECS, MIT, 1997.