COAST Tech Report 94-11

ACCESS CONTROL FOR COLLABORATIVE ENVIRONMENTS

by HongHai Shen

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

ACCESS CONTROL FOR COLLABORATIVE ENVIRONMENTS

A Thesis
Submitted to the Faculty

of

Purdue University

by

HongHai Shen

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

August 1994

To my wife Shen Fang, and my parents

ACKNOWLEDGMENTS

This dissertation would never have been finished without the guidance of my advisor, Prasun Dewan. Most ideas emerged from discussion with him. I am obliged to his patience, encouragement, and endless effort in guiding me through each stage of this doctoral program.

My graduate committee, Bharat Bhargava, John Korb, and Michal Young contributed valuable advice. Thanks also to Rajiv Choudhary, Eugene Spaford, John Riedl, William Gorman, Richard Newman-Wolfe, Luke Lien, and Rong Chang, who provided me with many constructive suggestions.

The financial and computing facility support provided by Purdue Research Foundation, SERC, the Purdue Computer Science Department, and IBM is gratefully acknowledged.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Appendix
Table

LIST OF FIGURES

Figure Page

Appendix
Figure

ABSTRACT

Shen, HongHai. Ph.D, Purdue University, August 1994. Access Control for Collaborative Environments. Major Professor: Prasun Dewan.

In this dissertation, previous work on access control for both collaborative and non-collaborative systems is surveyed. New access control requirements for general collaborative environments are identified, and it is shown that existing models do not completely meet these requirements. A new access control model is developed for meeting the requirements. In particular, a set of collaboration rights are identified based on a general collaboration model; exception-based, multiple inheritance mechanisms are used to support both flexible and high-level access specification; and dynamic, multiple ownership rules are developed to support flexible access administration. The model can emulate a variety of existing systems and meets the new access requirements. It has been implemented in a generic, extensible collaborative system, which relieves individual applications from implementing the model.

CSCW[85, 24, 69, 99, 105, 67, 106, 74, 40, 35, 34, 36, 77, 62, 61, 52, 66, 103, 100, 111, 46, 59, 23, 96, 6, 109, 5, 74] BACKGROUND:[47, 4, 51, 88, 19, 20, 21, 114, 115, 79, 108, 83, 75, 104, 82, 73, 18, 76, 81, 98, 108, 117]
SECURITY:[13, 15, 17, 94, 93, 102, 72, 22, 48, 87, 64, 50, 7, 54, 58, 1, 102]

1. INTRODUCTION

1.1 Background and Motivation

Collaborative systems, also called Computer Support Cooperative Work (CSCW), or groupware, have been defined by Ellis et al [44] as "computer based systems that support groups of people engaged in a *common task* and that provide an interface to a *shared environment*." Combining several disciplines including communication networks, user interfaces, and cognitive science, collaborative systems aim at assisting users in carrying out their group activities, potentially in a distributed fashion. Since people spend a large fraction of their time on group activities [24], such systems have the potential of increasing the productivity of working group significantly. Therefore, in recent years, there have been several research efforts that have focused on collaborative systems. Examples of such efforts are Vconf [74], Colab [112], Rapport [5], Rendezvous [90], XTV [3, 2], ICICLE [14], GROVE [43], LOTUS Notes [92], and Suite [31, 38, 40], to name just a few.

Unlike conventional systems that endeavor to make users feel as if they are the only users of the system, collaborative systems support and encourage the sharing of users' environments. To illustrate the nature of these systems, consider the software development application that is shown in figure 1.1. The application allows multiple users to edit C functions simultaneously and examine the coverage results of selected functions. This example belongs to our category of collaborative systems since users share the environment consisting of editing and testing results and work towards the common goal of collaborative software development.

One important aspect of collaborative systems is access control. It determines the set of operations a subject (process or user) can perform on an object. Access control can not only preserve data from unauthorized access from other users, but

Figure 1.1

Two users in a collaborative software development session. The users, `hhs` (left windows) and `rxc` (right windows), work on separate workstations to examine the coverage result of a C function, `insert`. User `hhs` first selects the function `insert` in his editing window (middle left window), and invokes an appropriate action in his menu window (upper left window). The coverage results of the selected function(s) are then shown in the testing windows of both users (lower windows). The statements that are not covered by the test data are underlined in the testing windows.

also protect data against unintentional accidental damaging operations made by users themselves. For instance, it can prevent a user from deleting a function or a line inserted by another user. It can also prevent a user from hiding a function that may affect other users' views.

Access control has been studied extensively in non-collaborative domains. Several models have been proposed in operating systems and database systems [71, 9, 27, 101, 118, 11, 13, 57, 64, 48, 50, 49, 95, 26]. From their experience with building these models, researchers have identified several general design requirements for access control systems (Saltzer [101]). Based on past work and scenarios of collaborative applications, we have identified several requirements that a collaborative access control model must meet. These include both high-level, general requirements and lower-lever, more specific requirements derived from the former:

- *Base the protection mechanism on permission rather than exclusion [101]:*

  The default situation should be lack of access, and the protection scheme should identify conditions under which access is permitted. This is safer than giving permission by default.

- *Total mediation [101]:*

  Every access to every object should be checked for current authority.

- *No secret design [101]:*

  The mechanisms should not depend on the ignorance of potential attackers, but rather on possession of specific, more easily protected, protection keys or passwords.

- *Least privilege [101]:*

  Every program and every user should operate using the least amount of privilege necessary to complete the job (the "need to know" principle).

- *Ease of use [101]:*

The model should be simple and easy to use. Otherwise, users will not routinely apply the protection mechanisms.

- *Generality:*

  The model should be general and be able to support arbitrary access control policies. In this way, it can support a wide-range of applications and meet the potential access needs of future systems.

- *Automation and Extensibility:*

  The model should make it easy for programmers to implement access control in multi-user applications. Code reusability resulting from automation is especially important for collaborative systems, since the amount of work on access control subsystem development could be substantial.

  Since it is impossible to anticipate the needs of all potential applications, parameterize them, and then automate them, it is important that the model be extensible. In particular, it should be possible for users to define their own type-specific access rights and access rules for a particular application. An automated and extensible model lets individual applications extend the model for meeting access needs not supported by the automated mechanism and supports the generality requirement.

- *Support collaboration rights:*

  Besides the traditional read and write operations, all other generic operations whose results can affect multiple users should be protected. In collaborative systems, users can share data objects and their properties, such as viewing properties, with other users. This has been defined by Dewan [37] as *coupling*. For instance, in the software development tool shown in figure 1.1, a user may share the view of a function with another user in the WYSIWIS (What You See Is What I See) coupling mode. In this mode, if one user changes the color or font of the function, or hides the function, the view of the other user will

be affected. This operation is not desirable if, for instance, the first user is not the "formatter" or the other user does not want his view to be disturbed. New, generic access rights such as the right to change a color and to format data are necessary in collaborative domains to prevent such unauthorized operations. We refer to these new rights as collaboration rights and use the term conventional rights to refer the traditional rights that protects the value of objects such as the read and write rights.

Defining rights for operations that affect multiple users supports the total mediation requirement and identifying a generic (hence automatable) set of these rights supports the automation requirement. Thus this requirement can be derived from the more general requirements of total mediation and automation.

- *Support multiple, dynamic user roles:*

  A role is an abstraction with a set of privileges that can be taken by one or more users. Ellis [44] and Patterson [89] point out that role-based specification should be supported so that users' access privileges can be inferred from the roles they take. In addition, we have found that users should be allowed to take multiple roles simultaneously and change these roles during different phases of collaboration.

  To illustrate the need for this requirement, consider the distributed conference system described in [116]. A conference in this system consists of several sessions. Each session begins with several talks, in which only the speaker is allowed to talk. Attendees are allowed to ask questions only if granted permission by the session chair. A panel follows the talks, in which each panelist takes turns speaking, followed by discussions in which all attendees are allowed to speak. In this system, a user may take the roles of speaker, attendee, session chair, and

panelist at the same time and may change roles during different stages of the collaboration. His access privileges may change as he changes roles.

This requirement can be derived from the "least privilege" requirement since it requires that users carry only the privileges that are required to perform their current roles, rather than all their privileges all the time. It can also be derived from the "ease of use" requirement since the concept of user roles, having a parallel meaning in daily life, can be easily understood and used.

- *Flexible specification:*

  The specification of access rights should be flexible. In particular, the model should support:

  - *Multi-grained specification.* The system should support variable-grained protection of subjects, objects, and access rights, that is, it should allow independent specification of each access right of each subject on each object. Similar to the analogous coupling requirement discussed in [39], this requirement supports data independence, which allows different protections over different shared entities; user independence, which allows different protections for different subjects taking different roles; and right independence, which allows different protections for different operations. All these independences are necessary to realize the "least privilege" principle because they make it possible to fine-tune the subjects' privileges according to their needs.

    To illustrate the need for this requirement, consider the software development tool again. The protected objects of the tool include hierarchical programs, which consist of functions, which in turn consist of lines of text. In this application, it is useful to prevent a function from being deleted by a new user `abc`. It is also useful to prevent an `observer` group from

modifying, that is, inserting, deleting, or replacing any text in the program. Furthermore, the application should protect individual comment lines from being read and the whole program from being deleted by unauthorized users. Therefore, access rights should be associated with not only coarse-grained user groups such as `observer`, but also particular users such as `abc`. Moreover, they should protect not only coarse-grained operations such as modify, but also fine-grained, specific operations such as delete. Finally, it should be possible to protect not only coarse-grained objects such as a program, but also fine-grained objects such as a comment line.

– *Pessimistic and optimistic access control.*

The system should allow the protections of both global data and their local buffers. In addition, the system should be flexible and allow independent access specification for both kinds of objects. In particular, the system should support the notion of optimistic access control, where access rights are checked at the time the local buffer of a user is committed. Under optimistic access control, users are allowed to change their local buffers even if they do not have the rights to change the corresponding global data. It is useful because allowing users to make changes to their local buffers makes it possible for them to communicate these changes to other users or save these changes locally, even if they do not have the privilege to change the global data. It is also useful to support a more restrictive form of access control, namely the pessimistic access control, where access rights are checked whenever the local buffer of a user is changed. Under pessimistic access control, users are denied the right to modify their local buffers if they do not have the right to modify the corresponding global data. This requirement is inspired and supported by some single user applications such as vi in Unix [1].

[1] In Unix, if a user edits a file X for which he does not have write access, vi will first issue a warning message: file X: [Read only], and thereafter let the user modify the local buffer arbitrarily.

To illustrate the need for this requirement, consider the software development application of figure 1.1. User `hhs` may not be allowed to directly modify the function user `rxc` created. However, it is useful to allow `hhs` to change his local buffer of the function so that he can communicate these changes to `rxc`, who can inspect these changes and decide whether to commit them globally. On the other hand, if more restrictive access control is to be imposed, `hhs` may not be allowed to even change his local buffer so that he cannot modify the function through communicating with a third party. The above two scenarios illustrate the use of optimistic and pessimistic access control, respectively.

- *High-level, exception-based specification:*

  To meet the previous requirement of ease of use, a simple, high-level model for specifying access rights must be provided to users. Saltzer [101] pointed out that the key to meeting the high-level and ease of use requirements "lies in better understanding the nature of the typical user's mental description of protection intent, and then devising interfaces which permit more direct specification of that protection intent". One specific method for realizing the high-level specification requirement is to allow exceptions to be part of the specification, since general setting with exceptions is how users typically specify their intentions.

  To illustrate exception-based specification, we continue with the software development application mentioned before. Assume a session chairman inserts the following comment line before a function: "This function is too detailed and trivial conceptually, please skip it for now and let user `abc` fix it". The chairman may want the line to be readable by the `suite` group, but not by its member `abc` because of its contents. With an exception-based specification, the above intention can be specified as "readable to `suite` except `abc`".

- *Efficient storage and evaluation:*

The access mechanism should keep the cost of storing, modifying, and checking access specifications low. In particular, the increase of system overhead by adding the access control mechanism should be kept low.

- *Flexible access administration:*

  The model should be able to support the following schemes for administrating access rights.

  - *Flexible ownership:*

    The model should be able to support the single-user ownership scheme supported in traditional systems such as Unix. In addition, it should be able to support group ownership for two related reasons. First, one user can create an object on behalf of a particular role. For instance, in the software development application, user `hhs` may create a function `common` on behalf of the `suite` role to allow every user who takes the `suite` role to be the owner of the function. Second, multiple users may contribute to the creation of elements of a complex object through collaboration. For example, two persons may work together to co-author a paper. If the paper is treated as a complex object, then it is meaningful that both persons jointly own the object because they both contribute to its contents. In the above two cases, multiple users may jointly own the object and each of them should have the privileges associated with ownership.

    The exact meaning of ownership may vary depending on the application. In a classroom exam application, the teacher owns the exam in the sense that he is in charge of administrating all the rights the students should have. However, the "least privilege" principle requires that the teacher not have all the rights directly himself because the teacher is not taking the exam himself. On the other hand, in the software development application

mentioned above, the owner of a comment line should have all the rights associated with the line, including the privileges to transfer these rights.

Therefore, the model should be able to support both single-user and group ownership and allow users to specify the semantics of ownership.

– *Flexible delegation and revocation:*

The model should support the notion of access delegation [1], which allows an owner or a user possessing certain privileges to distribute his privileges to others. This is useful when the responsibility of an authorizer should be partitioned and distributed among different parties. The delegation model should be flexible. It should allow a user to delegate not only all of his privileges, but also a selected part of his privileges, and perhaps the authority to distribute these privileges further to others. In the software development application, the chairman of a reviewing team may be in charge of distributing only the coupling rights, while a user interface specialist may be responsible for distributing only the formatting rights. Furthermore, the chairman may delegate some of his rights to the secretary to help him administrate these rights.

Related to access delegation is the concept of revocation, which is an operation that undoes the effects of delegation [45]. It is important because it provides a way to contain the spread of access rights. Two possible semantics of revocation can be defined: a) shallow revocation, which simply removes the previously granted right. b) deep revocation, which recursively revokes rights from those to whom the grantee granted the right. The model should be able to support both of the above semantics.

To illustrate the above requirement, consider again the example of a classroom examination. Assume that teacher `pd` has delegated to his teaching

assistants `rxc` the right to distribute the coupling rights of the class and the right to further distribute these rights. We can envision two scenarios in which `pd` may want to use both semantics of revocation to revoke the rights he delegated to `rxc`. In the first scenario, `pd` wants `rxc` to help him only temporarily with the administration of the coupling rights. Once `pd` has the time to handle the task himself, he would like to do it totally by himself to save himself the effort required to coordinate with `rxc`. In this situation, `pd` requires the semantics of shallow revocation to revoke the delegation since he would still want to keep what `rxc` has done. In the second scenario, assume that `pd` finds that some of the coupling administration by `rxc` is not correct. In this situation, `pd` needs the semantics of deep revocation since he would like to undo all the administrating work that `rxc` has done.

Some of the above requirements, including automation and extensibility, collaboration rights, pessimistic and optimistic control, and flexible ownership, are new requirements we have identified based on previous work and application scenarios. As discussed above, interdependencies exist among the access requirements we have listed. They are summarized in figure 1.2.

Some of the above requirements conflict with each other. It is difficult to achieve high-level and easy specification without sacrificing flexibility and generality, since understanding and configuring the parameters introduced for flexibility and generality adds complexity to the model, which will conflict with the high-level and ease of use specification requirements. It is also difficult to meet the generality, efficient storage, and efficient evaluation requirements at the same time. Generality will inevitably add many run-time parameters to the model, which can affect the efficiency. Efficient storage and evaluation can conflict with each other because of the traditional time and space relationship. Thus, an access control model must make tradeoffs among these conflicting requirements.

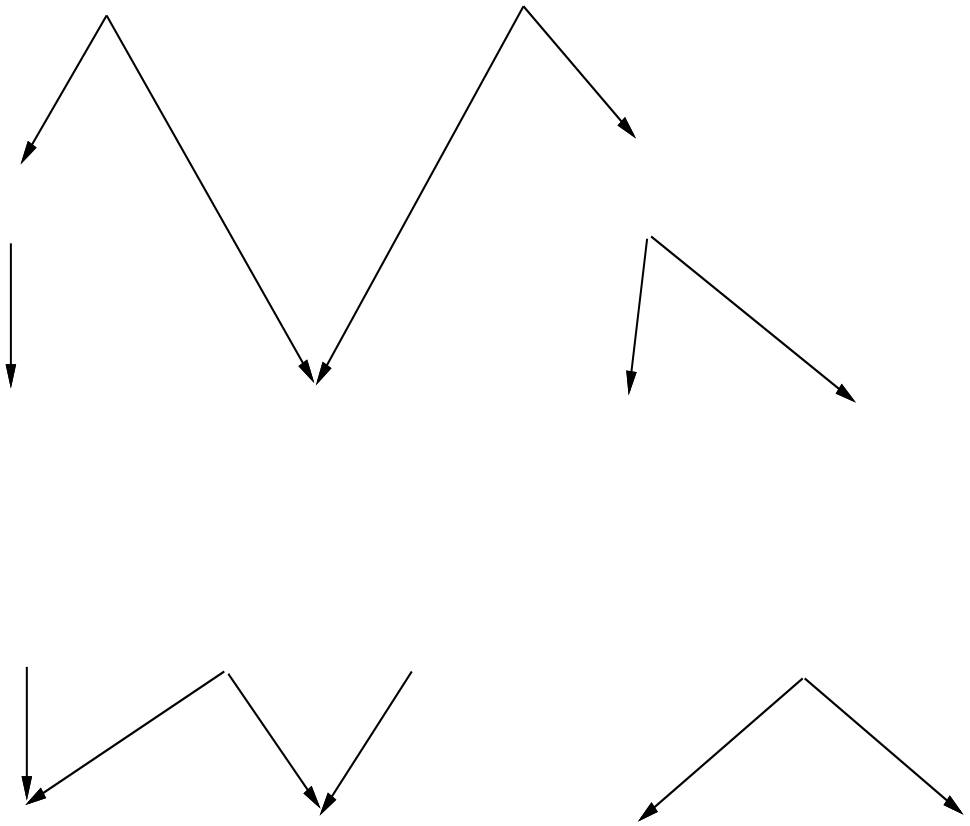

Figure 1.2 Dependency of access requirements. The arrow represents the relationship of being derived from.

1.2 Thesis Scope and Structure

Access control is a mechanism needed to realize system security and integrity. However, it deals with only one aspect of system security and integrity. To ensure system security and integrity, several addition concepts must be supported besides access control [29]. They include: cryptographic control, information flow control and covert channel analysis, inference control, user authentication, and concurrency control. Among these many aspects of system security and integrity, we focus our research on access control, which ensures that all *direct* accesses to objects are authorized. Although access control alone is not enough to ensure security and integrity, it does provide an important component for security and integrity.

The research methodology used to study this problem is as follows:

- identify access requirements based on other work and application scenarios.
- evaluate previous work using these requirements.
- design and implement a new model to meet the requirements.
- evaluate, using inspections, simulation, and experiments, how well the new model meets the requirements.
- compare the new model with others.

The objective of this thesis is to identify a model that provides a solution that meets one or more requirements better than existing models. In particular, the objective is to identify an access control model that supports high-level specification and automation without unduly sacrificing generality, flexibility, and efficiency. We will discuss various possible design alternatives for meeting these requirements, identify tradeoffs, and motivate our design choices.

To support the above access requirements, it is necessary to base the access control model on a general collaboration model, which determines how users interact with an application and how these interactions are shared. A general collaboration model

is required for two reasons. First, it is difficult to achieve the high-level specification requirement without using any information about the semantics of the underlying protected objects. A collaboration model provides such information. Second, to support collaboration rights, it is necessary to identify a set of generic collaboration operations that is application-independent. A general collaboration model defines generic operations and therefore the corresponding collaboration rights. We describe a general collaboration model in chapter 2.

Although our main purpose is to provide access control mechanisms, we also discuss collaborative access control policies to determine whether the mechanisms we provide are general enough to support a variety of access policies.

The following are the major novel components of our model:

- A generic set of collaboration rights based on a general editing-based collaboration model [38].
- Variable-grained specification of access rights.
- A set of rules in the subject, object, and right dimensions that allow new rights to be inherited from existing ones.
- A set of rules for resolving conflicts of definitions in a particular dimension and in different dimensions.
- A multiple ownership scheme defined through inheritance relationships.
- A flexible delegation and revocation mechanism based on indirect roles.
- Automation of access control based on a generic multi-user framework.

We have implemented the access control model as part of the Suite system [38] and use this system as a concrete example to discuss the model. Some preliminary results have appeared in [107].

The rest of the thesis is organized as follows. Chapter 2 describes related work in system security and integrity, discusses the collaboration model on which our access

control model is based, surveys previous work on access control with respect to the requirements we have identified, and discusses how well they meet these requirements. Chapter 3 describes the model we have developed for meeting these requirements and motivates our design decisions by concrete applications. Chapter 4 discusses the implementation of the model as part of the Suite system and describes preliminary experience with the implementation. Chapter 5 compares our work with other work. Finally, chapter 6 summarizes our model and presents future directions.

2. RELATED WORK

As discussed in chapter 1, our access control model is based on a general collaboration model. In this chapter, we describe the collaboration model. We also investigate the existing work on access control and discuss how well they meet the requirements we have identified. We do not describe these systems in full detail. Instead, we focus on only those parts of the systems that are related to the issues we address. We first begin by relating access control to other mechanisms for supporting our general goal of realizing system security and integrity.

2.1 System Security and Integrity

As mentioned in chapter 1, access control is a mechanism needed to realize system security and integrity. However, it deals with only one aspect of security. It only assures that all *direct* accesses to objects are authorized. Besides violation of direct access control, threats to security can also occur *indirectly.* Denning [29] identifies the following threats to security: *browsing* – searches through main memory or secondary storage for information; *leakage* – the transmission of data to unauthorized users by processes with legitimate access to the data; and *inference* – the deduction of confidential data from non-confidential data. Consequently, to ensure system security and integrity, several concepts must be supported ([29, 30, 93, 94, 56, 27, 102, 25, 28, 72, 55, 8, 10, 70, 42]) besides access control. They include: *cryptographic control*, which prevents unauthorized browsing by making data unintelligible; *information flow control and covert channel analysis*, which controls the flow of information to ensure that information gained through legitimate access will not be leaked to unauthorized users; *inference control,* which controls access to queries on data to prevent statistical compromise, i.e., deduction of confidential data of individuals by correlating group

statistics; *user authentication*, which ensures that users can be properly authenticated from running processes by some authentication mechanism such as Kerberos [113]; and *concurrency control*, which controls access to shared data to maintain certain consistency properties such as serializability [88]. In the thesis, however, we focus our research on access control, instead of addressing all the above issues.

2.2 Collaboration Model

As discussed in section 1.1, in order to meet the requirements we have identified, it is necessary to base the access control model on a general, high-level collaboration model. By doing so, it is possible to identify a generic set of collaboration rights that are application independent. Furthermore, by exploiting the underlying semantics of objects defined by the collaboration model, it is possible to offer protection of objects in both a flexible and high-level way. We have chosen the Suite collaboration model as our basis [39] because, as described below, it offers a general model of interaction on which a set of generic collaboration rights can be derived, and it also offers a framework for realizing automation.

The Suite collaboration model has two components: an interaction model which decides what the interaction objects are and how users interact with them; and a coupling model, which determines the sharing among users. The Suite interaction model is based on the idea of generalized editing [53, 41]. It offers the abstraction of an *active variable*, which is a variable-grained program data structure of arbitrary type that can be edited to trigger actions [83, 32]. The model also offers a set of interaction objects that are manipulated by users. These objects include interaction variables, attributes, value groups, and object windows [38]. An *interaction variable* is the local copy of an active variable that an end-user interacts with. An *attribute* is an interactive property associated with an interaction variable such as its display format and the callback to be invoked when the variable is edited [31]. A *value group* is a set of related interaction variables that share some attributes. An *object window* is a rectangular area on the screen that displays the values of interaction variables.

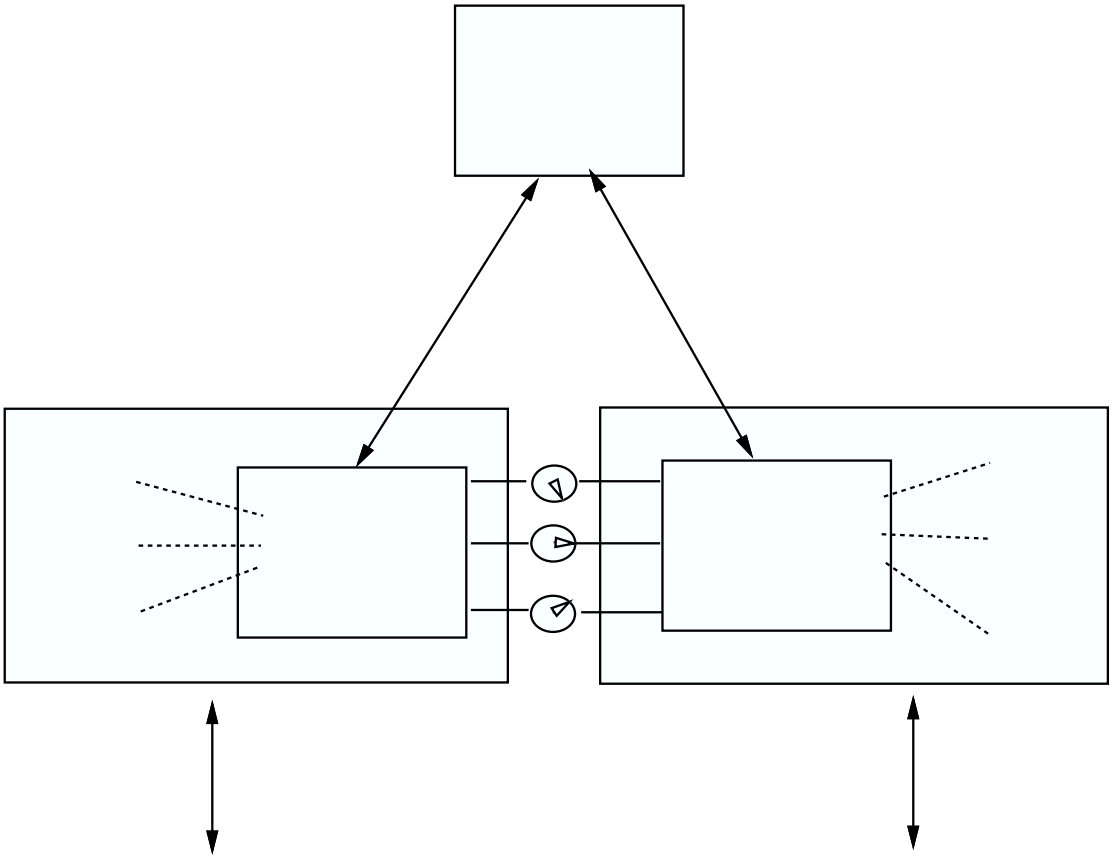

Figure 2.1 Collaboration model.

The model also supports the notion of a session, which is a series of interactions by multiple users with a shared application. A session is activated by invoking the corresponding application. Users can then join the session, that is, connect to the application and manipulate the variables defined in the application. Users can also leave the session and then later rejoin it.

Figure 2.1 illustrates the Suite collaboration model. In the figure, two users are in a collaborative session. As the figure shows, the users have their own copies of the active variables (interaction variables), each of which is associated with a set of attributes

such as Attr1, Attr2, and Attr3. User1 and User2 manipulate the interaction variables through a set of editing commands provided by the system.

The Suite coupling model [37, 39] is built on top of the Suite interaction model. It offers sharing of various properties of interaction objects, such as the value or format of an interaction variable, or the scrollbar position of an object window. This sharing is represented in figure 2.1 by the lines that connect two users' workspaces. Moreover, the adjustable dials along the connection lines indicate that the sharing is flexible in that users can decide which properties of interaction objects are to be shared, and when changes made by a user to a property of an interaction objects are communicated to other users sharing it. The level of sharing can be decided according to several criteria, such as how structurally complete the change is, how correct it is and the time at which it was made [39]. For instance, users can specify whether they want to share the value of an interaction variable. If they do decide to share it, they can further specify whether they want to share changes to the value incrementally or only when the value is committed. Similarly, they can determine if the views of interaction variables are to be shared. The coupling among interaction variables is determined by the coupling attributes of these variables.

The Suite interaction model and coupling model also contain specification and implementation components. These components allow active variables to be declared using a conventional programming language such as C. When users connect to a program, interaction variables for them are automatically created. The attributes of these variables are initialized to default values. They can be changed dynamically by either the users or the program. An inheritance-based scheme is provided to specify attributes common to groups of interaction variables [32].

Let us illustrate the collaboration model using the concrete example of the software development application that we mentioned earlier. The active variables of this application are C programs, functions, and lines. A session is first created by invoking the application. Users can then join the session by connecting to the application and get object windows that display local copies of those variables defined in the

application, as shown in figure 2.2. These copies can be edited independently by the users to cause changes to the underlying active variables. For instance, the interaction variable containing the program text displayed in the object window can be edited and then committed to change the underlying program. Moreover, users can share properties of these interaction variables such as display formats. In the figure, the collaborators have set the `ValueCoupled` and `ViewCoupled` attributes [37, 33] of the interaction variables to `True`, thereby sharing both their values and views. As a result, when one user changed the view of the `minit` function by "eliding" it[1], the result was shared by both users, as shown in the figure.

Figure 2.2

Two users in a collaborative editing session. When `rxc` (left) elides a function `minit` by pressing the `elide` button, `hhs`'s view is also affected (right).

We have chosen the Suite interaction model as our basis because, in contrast to other models of interaction such as the teletype-based model, the generalized editing model offers several desirable interaction features such as uniformity, direct-manipulation, multithread/multiuser dialogues, structure-based commands, and continuous update of presentations [41]. Most interactive applications known to us such

[1]elide is an operation used to condense the visual representation of a data structure.

as program editors and spreadsheets support this model of interaction. We have chosen the Suite coupling model as our basis because it is the most general coupling model known to us.

2.3 Access Control Models

In this section, we evaluate existing work on access control using the access requirements listed in chapter 1. We do not consider all of the requirements since some of the requirements are low-level and are covered by higher-level requirements. Specifically, we evaluate the work using the following requirements: flexible and high-level specification, multiple, dynamic roles, collaboration rights, flexible ownership, automation and extensibility.

2.3.1 Matrix Model

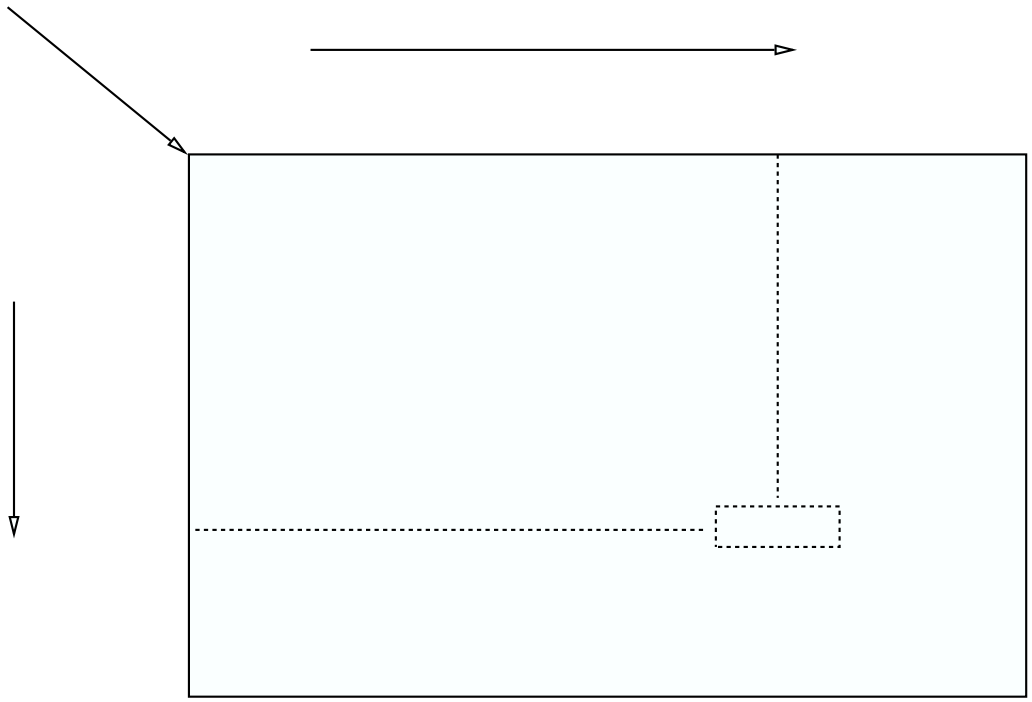

Figure 2.3 Matrix model.

The matrix model provides a framework for describing protection systems. It was proposed by Lampson [71] and refined by Graham and Denning [57] for protection in operating systems. Conceptually, the model has a set of access rules which describe what a protection state is and how state transitions occur. It has a set of protected objects, which are the entities to which access must be controlled and a set of subjects, whose access to objects must be controlled. The protection state of the system is represented by an access matrix A, with rows representing subjects, columns representing objects and A[s,o] denoting the access rights a subject s has over an object o. The access checking rule of the model states that a request by subject s for accessing object o is granted only if A[s,o] contains the requisite right. To protect the access matrix itself, the model introduces the notion of *-rights. A *-right is defined for each regular access right and its holder has the privileges to grant the corresponding regular right. For example, the write* right allows its holder to grant the write right to others. The model also contains a set of rules (commands) governing changes to the subjects, objects, and rights in an access matrix. In particular, the model supports the notion of ownership and defines rules allowing the owner of an object to grant rights to others.

Because the access matrix is usually sparse in practical situations, systems based on the matrix model seldom choose the matrix as the implementation data structure. Instead, for efficiency reasons, they usually use two common approaches to implement the basic matrix model: access control list (ACL) and capability list (C-list). The former stores the matrix by columns, i.e., associates each object with a list of (subject, right) pairs. Systems using this approach include Unix [97], Multics [101], and System R [45, 60]. The latter stores the matrix by rows, i.e., associates each subject with a list of (object, rights) pairs, which are called capabilities. Hydra [118] and CAP [84] are two examples of systems using capability lists. A variation of the capability mechanism is called the "lock and key" approach [57]. Each subject is associated with a C-list of the form (X, K) where X is an object and K is a key. Each object is associated with a lock list of the form (L, R) where L is a lock and R is a right.

When subject S is checked against right R over object X, its C-list is first searched to get its key K of X. The lock list associated with X is then searched to get lock L of right R. The access is granted only if K matches L. The above mechanism can be regarded as an indirect capability approach, which makes it easy to realize the deep revocation semantics.

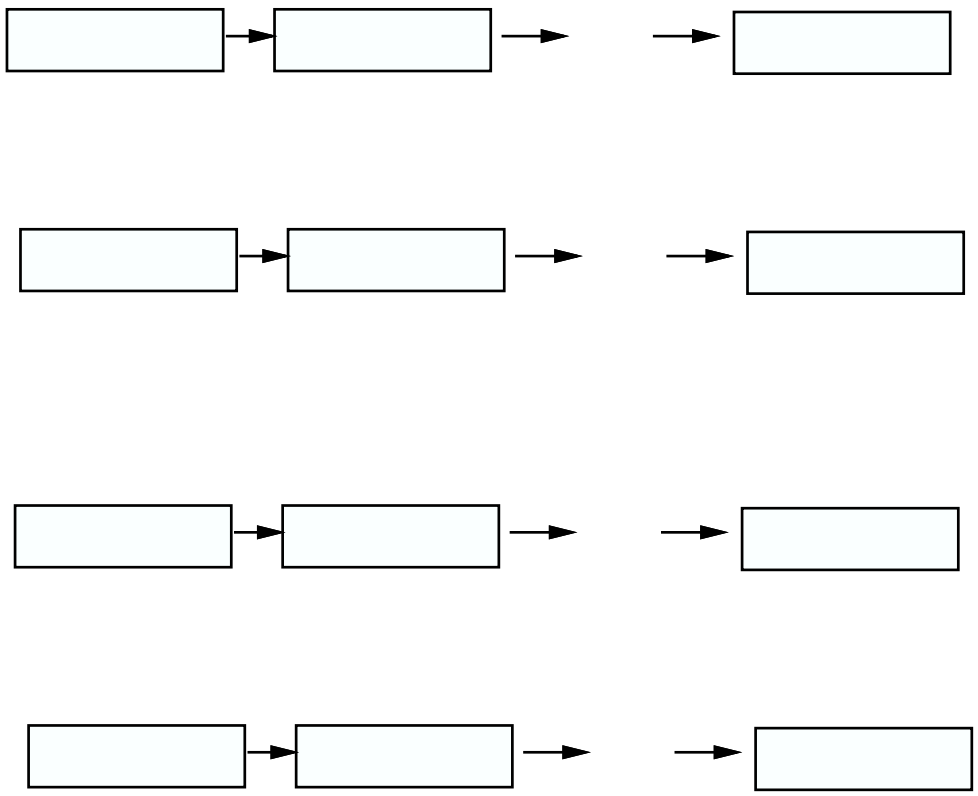

Figure 2.4 Access matrix data structures.

The matrix model is general in that it does not specify the exact nature of the subjects, objects, and access rights of the system. Nor does it specify how the access matrix is to be specified and implemented. It is flexible in the sense that an individual system can decide how fine-grained these elements are. On the other hand, because it

is general and does not assume any information about the semantics of the underlying system, it is unable to address the issue of high-level specification. In addition, it does not address the issue of collaboration rights. Finally, the matrix model only provides a general conceptual model for protection and does not support automation. It is up to the individual systems that are based on the model to implement the model.

2.3.2 Unix

Unix [97, 55] is an example of a system based on the matrix model that makes some assumptions about the underlying domain to offer high-level specification. It assumes that the protected objects are files and directories, and thus does not allow finer-grained protected objects. Users are grouped into owner, group and others, which correspond to the file's initial creator, users that are in the file's group, and everybody else in the system. Each file is associated with the following rights for the above user groups: read, write and execute.

Unix uses the access control list (ACL) approach to realize the access matrix. To specify the ACL associated with each file, a user need only specify the above three access rights for the three categories of users. Unix allows a user to be in multiple user groups. A user's rights are the union of the rights that all of these groups have. The coarse granularity of objects, simple partitioning of users and limited access rights enable Unix to offer relatively high-level access specification. As a result, few specifications are needed to protect the objects.

Unix also supports the notion of ownership. Each directory and file is associated with a unique owner, who is responsible for specifying the access definitions for the object. In addition, ownership can be changed by the superuser. Unix also allows a umask to be associated with any process. The value of the umask decides the default protection of a file when the file is created. In particular, it specifies the default right the owner has over a file.

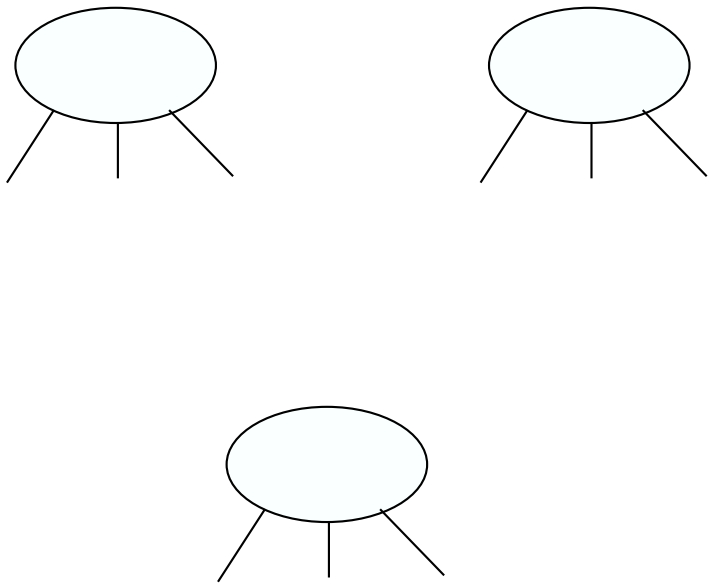

Figure 2.5

Unix specification model: operations are grouped into `read`, `write`, and `execute`; users are grouped into `owner`, `group`, and `others`; and objects are grouped into files.

The Unix approach, however, does not meet the requirement of flexibility due to its simplifying assumptions. Protection objects at the file level are too coarse-grained and rigid. Take the software development example of the previous section. We described scenarios where it is desirable to protect a fine-grained object, such as a comment line from being read by some user. In Unix, however, we would have to treat each comment line as a separate file to achieve the above access requirement, which is cumbersome and inefficient. Furthermore, the simple Unix user grouping does not support role-based specifications and exceptions, therefore only partly meets the high-level specification requirement. It also does not support collaboration rights such as the right to couple with others. Finally, it does not support the notion of group ownership because at any time only one owner is associated with a file.

Unix supports automation at the file system level. Users do not need to worry about the implementation of protection of new files they create. However, it does not support extensibility. That is, it does not allow users to add new types of protected objects and access rights.

2.3.3 AFS

AFS [16] is a file system that extends the work of Unix. It assumes that the protected objects are files and directories. Users are allowed to create their own groups directly and use multiple groups in the access control lists. A user's rights are the union of the rights of those groups of which he is a member. The rights defined over directories are the following: l (list a directory), i (insert files to a directory), d (delete files in a directory), and a (administrate the access list). The rights defined over files are the following: r (read a file), w (change a file) and k (place an advisory lock on a file). These rights, however, are all associated with directories, and all the files in a directory have the same access permissions. Therefore, only one specification needs to be made if all the files in a directory have identical rights. The access list of a directory is copied from its parent directory upon creation. The Unix access list associated with a directory is ignored by AFS. However, the access list of a file defined in Unix is used to represent file-level access control. Its semantics is changed in the following way. Only the access rights associated with owners are used and others are ignored. In addition, the right specification for the owner is used to represent all users and an operation on a file is permitted if and only if the ACL defined on both the file and directory allows it. Therefore, if the file bits look like `-r---??????`, then ALL users will not have the write right to the file. AFS uses a variation of the *-rights approach of the matrix model to support access administration. Those users that have an 'a' right over a directory can change the access list of the directory, i.e., the 'a' right corresponds to the *-right of the matrix model. In addition, those users who have the write right to the file can change the access list of the file. That is, the write right of a file is also used as a *-right for the file. AFS also supports

subject inheritance by allowing access rights associated with a specific user to override the rights associated with more general user groups. In addition, AFS supports the notion of negative access lists, which explicitly specify the denial of access rights for users. In this way, it was able to support exception-based specification.

The AFS approach, however, does not meet the requirement of flexibility. Access protection at the directory and file level is too coarse-grained to support finer-grained protected objects. The protection mechanism at the file level is for all users, that is, it is not possible to specify access rights for individual users or groups at the file level. In addition, AFS does not allow groups to contains subgroups to allow easy specification of subjects. It also does not support collaboration rights such as the right to couple with others. Finally, it does not support the notion of group ownership to allow multiple users to jointly own a directory or a file.

AFS supports automation at the file system level. Users do not need to worry about the implementation of protection of new files they create. However, it does not support extensibility. That is, it does not allow users to add new types of protected objects and access rights.

2.3.4 Hydra

Hydra is an example of a protection system that is flexible and general but offers low-level specification mechanisms. Hydra's design philosophy is to build a collection of facilities of "universal applicability", from which protection subsystems can be built flexibly. Therefore, unlike Unix, it does not assume any a priori knowledge of the protected objects, nor knowledge of the kinds of access that should be granted to them. Instead, Hydra uses a general notion of a "typed object" such as a "file" and a "page of memory" as a protected object. New types of objects can be defined by users, and each of these types can be protected. In particular, fine-grain protection of objects is supported in Hydra because the system supports protection of arbitrary objects. For instance, the system can treat a comment line in the software development example as a separate protected object. In addition, Hydra does not provide a fixed collection

of access rights, although a set of kernel rights is predefined for convenience. Instead, it defines a general notion of "applying an operation to an object", thereby providing a flexible way to protect type-specific operations.

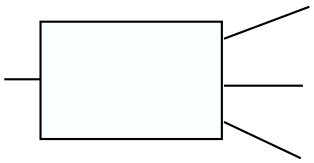

Figure 2.6

Hydra specification model: each capability indicates which operation a subject can do on an object.

Hydra implements the matrix data structure using the capability list (C-list) approach. For each individual subject, the system maintains a list of capabilities. A capability indicates the access privileges a user has over each object. The kernel of Hydra itself does not provide any mechanisms to facilitate the specification of the capability lists. As a result, the specification can be tedious if the number of capabilities a subject has is very large and has to be manually specified. Take the example of the software development application. Each time a user adds a comment line into the code, he has to explicitly give every user who can access the line an appropriate capability. In addition, it does not support exception as part of specification. For instance, it is not possible to support the specification "the line is readable to all `serc` users except user `abc`" directly. Thus the Hydra approach does not meet the requirement of high-level specification. In addition, it also does not directly support multiple, dynamic user roles.

The designers of Hydra, in constructing a generic protection system, observed that "while ownership is a useful, perhaps even an important, concept for certain 'security' strategies, to include the concept at the most primitive level would be to

exclude the construction of certain other classes of truly secure systems". As a result, Hydra discards the ownership concept and does not associate any ownership property with a protected object. It is up to individual applications to address the authorizer problem.

Hydra provides automation of the above described primitive protection mechanism: protection subsystems can be built on top of it without requiring detailed implementation of the management of matrix data-structures and checking of rights. All these are done automatically by the kernel of Hydra. It is also extensible in that new types of objects and type-specific rights can be defined in the model.

2.3.5 Bell-LaPadula Model

The Bell-LaPadula Model (BLM), also called the multi-level model, was proposed by Bell and LaPadula [9] for enforcing access control in government and military applications. In such applications, subjects and objects are often partitioned into different security levels. A subject can only access objects at certain levels determined by his security level. For instance, the following are two typical access specifications: "Unclassified personnel cannot read data at confidential levels" and "Top-Secret data cannot be written into the files at unclassified levels". This kind of access control is also called *mandatory access control*, which, according to the United States Department of Defense Trusted Computer System Evaluation Criteria [87], is "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (e.g., clearance) of subjects to access information of such sensitivity". The converse of mandatory access control is *discretionary access control*, which is defined by [87] as "a means of restricting access to objects based on the identity of subject and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) to any other subject".

The Bell-LaPadula model supports mandatory access control by determining the access rights from the security levels associated with subjects and objects. It also supports discretionary access control by checking access rights from an access matrix. With respect to specification, we can regard the multi-level model as adding higher-level mechanisms to the matrix model. In addition to supporting arbitrary access specifications to the access matrix, the model groups protected objects according to different security labels and decides user privileges by their authorized security clearance levels [2].

More formally, each object is associated with a security level of the form (classification level, set of categories). Each subject is also associated with a maximum and current security level, which can be changed dynamically. The set of classification levels is ordered by a $<$ relationship. For instance, it can be the set { `top-secret`, `secret`, `confidential`, `unclassified` } where `unclassified` $<$ `confidential` $<$ `secret` $<$ `top-secret`. A category is a set of names such as `Nuclear` and `NATO`. Security level A *dominates* B if and only if A's classification level $>$ B's classification level, and A's category set contains B's. For instance, (`top-secret`, {`Nuclear, NATO`}) dominates (`secret`, {`NATO`}) because `top-secret` $>$ `secret` and the set {`Nuclear, NATO`} contains {`NATO`}.

In the model, an access request (subj, obj, acc) is granted if and only if all of the following properties are satisfied:

- simple security property (no read up): if acc is read, then level(subj) should dominate level(obj).
- *-property (no write down): if acc = append, then level(obj) should dominate level(subj); if acc = write, then level(obj) should be equal to level(subj).
- discretionary security property: the (subj, obj) cell in the matrix contains acc.

[2] As shown in [91], it is awkward, though not impossible, to specify this kind of access definition using the matrix model.

By basing its mandatory component on hierarchical security categories, BLM provides high-level specification of mandatory access control. However, if a collaborative application, such as the software development tool, chooses to impose the discretionary access control, the multi-level model can support only the low-level specifications provided by the matrix model. It also does not allow exceptions as part of the specification. Therefore it only partly meets the high-level specification requirement. In addition, like other traditional models, the model does not address the issue of collaboration rights. It also does not address the issue of flexible access administration. Finally, it does not address the automation issue because it is up to individual systems to decide what kind of system support they will provide.

2.3.6 System R

Traditional database systems represent the approach of using the underlying structure of objects to facilitate the specification of the access matrix. In database systems, there are a large number of data objects that need protection and it is nearly impossible to specify the access rights for each object individually. We use System R, a well-known relational database system developed by IBM [45], as a model to illustrate the traditional database approach to solving the specification problem. In System R, protected objects are variable-grained data objects, which can be relations, fields of relations and relational views. Relational views can be defined through the SQL query language and used to partition the relations horizontally. Access rights are select (read), insert, delete and update. All these rights apply to entire relations and views except update, which applies to fields. The update privilege to a field of a relation implies the capability to update the field in all the records of the relation. The select, insert and delete right on a view or a relation implies the right to select, insert, or delete a record in the view or relation. In this way, System R is able to provide a relatively high-level specification model.

Like other conventional systems, System R supports only the read, insert, delete and update rights. Therefore, it does not meet the requirement of collaboration

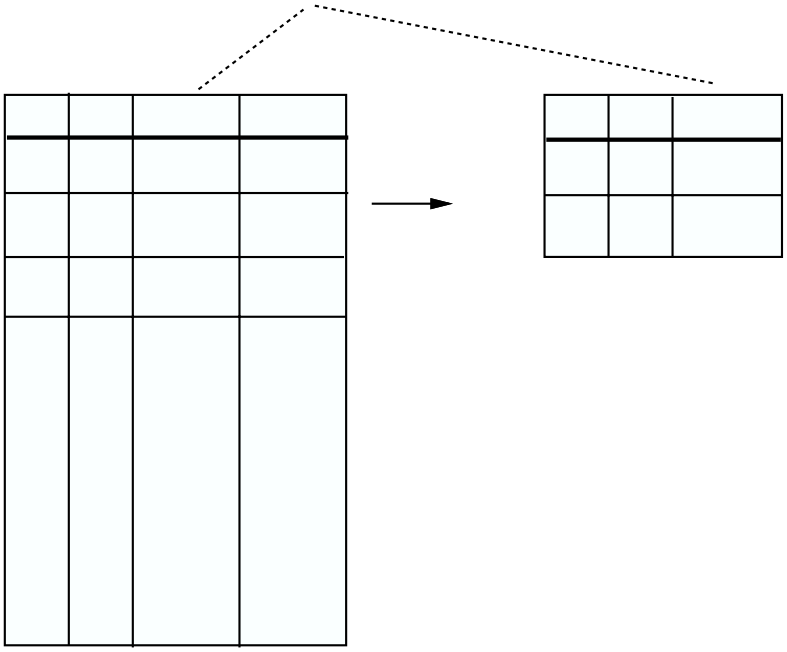

Figure 2.7

Specification in System R: Relational view inherits its access definitions from corresponding relations. For instance, the update right of field `f3` in the relational view is inherited from `f3` of the relation.

rights. In addition, it does not support role-based specification, nor does it support exception as part of the specification.

System R uses the *-right concept of the classical matrix model. It allows users to dynamically transfer the rights and allows these rights to be further transferable. It also supports both the shallow and deep revocation semantics. However, it does not support the notion of joint ownership.

System R supports automation. The access control mechanism is built within the database management systems (DBMS), on top of which individual databases are built. The system takes care of the maintenance of the access matrix and enforcement

of its semantics. However, it is not extensible in that only predefined rights and predefined type of objects (relations) are supported.

2.3.7 Rabitti's Model

The work described by Rabitti et al [95] is a recent model that endeavors to meet both the flexible and high-level specification requirements. It is proposed for the protection of object-oriented databases. This work is based on an earlier work by Fernandez [49] that uses inferences of access rights to facilitate access specification. Inferencing can be used, for instance, to assert that the write right implies the read right. If this assertion is true, then it is unnecessary to specify and store the read definition once the write definition is entered. Rabitti's work also uses data semantics by allowing inheritance of access definitions from the object class hierarchy, so that it is unnecessary to specify and store access definitions for each object. To facilitate specification and maintain consistency in the presence of a large amount of data objects, this work also uses the notion of *negative rights*, which are explicit denial of access privileges; *strong rights*, which are rights that cannot be overridden by other rights; and *weak rights*, which are rights that can be overridden by strong rights. By doing so, both the high-level and flexible specification goals can be achieved. In addition, exception-based specification is also supported in the model by the notion of negative rights.

The model also allows users to take multiple roles. It grants users a right if any of the roles they take has the positive right. In other words, users inherit all the positive rights of the roles they take.

Part of the model has been realized in the Orion Object-Oriented Database System [12], which provides system support through DBMS.

Like other conventional systems, the model discusses only the read, insert, delete, and write rights and does not address the issue of collaboration rights. It does not address the specification problem when there are a large number of rights. The

approach of allowing users to inherit all the positive rights of their roles is over-simplified. The issue of flexible authorizer is also not addressed in the work.

2.3.8 Current Work on Collaborative Access Control

Several existing collaborative systems also provide some access control. Among them are the GROVE outline editor [43], the DCS distributed conference system [86], the UCSD multimedia conference system [116], and the UNC Jointly-Owned Object System [63].

2.3.8.1 GROVE Outline Editor

GROVE is a group outline editor designed for use by a group of people interacting synchronously during a work session. The protected objects can be outline items, which are arranged in a structure hierarchy. They can also be sets of outline items called views. Users can define three types of views: private views, which are accessible only to the users themselves, shared views, which contain items accessible to an enumerated set of users, and public views, which contain items accessible to all users. If we classify users according to the privileges they have to the above three types of views, we get a Unix-like partition: self, which corresponds to private views, group, which corresponds to shared views, and others, which corresponds to public views. Within a shared view, an item is associated with an authorization form, which a user can use to interactively enter access definitions. GROVE also supports structure inheritance of protected items: a child item in the hierarchy inherits the access definitions from its parents.

By providing the above mechanism, GROVE is able to provide a high-level specification model similar to Unix. It also partially supports the flexibility requirement by allowing multiple-grained protected objects.

GROVE supports ownership based authorization. For shared view objects, the owner is responsible for specifying the access control list, i.e., the read and write rights of each user over the object.

The flexibility of GROVE is, however, limited. It currently supports only the read and write rights and does not support finer-grained rights such as insert and delete rights, and collaboration rights such as the right to change data format or couple with others. It also does not address the issue of multiple ownership. In addition, it does not support users taking multiple, dynamic roles. Finally, because GROVE is a specific application and not a system, it does not support extensibility. It supports only outline items. It is not possible to support other protected objects and non-predefined rights in the model.

2.3.8.2 DCS

DCS is a distributed conference system that provides a set of mechanisms for real-time conference management. Within a conference, users' rights are decided by the roles they take, which can be "voter", "non-voter", or "observer". A voting mechanism is provided to let voters control conference activities, such as merging and splitting of conferences, and changing of user roles. Both voters and non-voters have read and write access to all files within the conference. Observers are denied write access, but may view conference files and other objects.

Because DCS bases its protected objects on files and conferences and limits the operations to reading and writing of files and voting on conferences, the number of protected objects and rights in DCS are relatively small. Therefore, it is able to provide a high-level specification mechanism similar to Unix. DCS also supports automation in that the protection mechanisms it provides are generic and can be applied to any conferences under its management.

On the other hand, its flexibility is limited since it does not support fine-grained protected objects. In addition, it does not address the issue of collaboration rights. Because the application domain of DCS is restricted to conference control, the extensibility of the system is limited. It only supports a set of predefined rights, and a set of predefined roles related to conference control. It also does not address the issue of flexible authorizer schemes.

2.3.8.3 UCSD Multimedia Conference System

Vin et al [116] propose an access control mechanism for supporting multimedia collaborations. The mechanism uses the notion of a "stream" to represent media communication from a source to a set of destinations. A stream specifies, for each media, which users are allowed to transmit the media and which users are allowed to receive it, and at which time the transmission of media can be initiated. A session is used to model the exchange of media among a set of users. It consists of a sequence of semantically related streams, and the time at which the exchanges are allowed to occur. The notion of stream and session thus determines the access rights of users for a media by specifying who can exchange the media and when. In addition, the model defines a conference as a temporally ordered sequence of sessions to reflect the dynamic changes in the access rights of users during different times of collaboration. Therefore, the work supports a set of collaboration rights, such as the transmission rights and receiving rights. The specification model of the system includes a set of temporal logic rules so that time-related requirements can be specified easily. The notion of media data is abstract and can be used to represent multiple-grained protected objects, and therefore is flexible. The model also allows users to take multiple roles dynamically and infers the rights from their roles. In addition, the model supports automation since it provides system support for the implementation of the semantics of the model.

However, the model does not address the issue of high-level specifications in the presence of large amounts of data. It does not support the collaboration rights such as the right to change date formats. It also does not support exception-based specification. In addition, it does not allow dynamic definition of ownership semantics, or support group ownership. Finally, its extensibility is also limited because it does not allow users to define the rights of their own.

2.3.8.4 UNC Jointly-Owned Object System

Guan et al [63] propose an extension to the classical matrix model that supports the notion of joint ownership. In their work, they provide a mechanism that allows

multiple users to own an object. In addition, they associate conditions with objects that determine the access to them. A condition can be authority-based or quorum-based. An authority-based condition associates a user list (called control authorities) with an object, and specifies that before a right of an object is given out, the users appearing in the list must be present. A quorum-based condition, on the other hand, associates a number q with an object, and specifies that before a right is given out, at least q co-owners of the object must be present to meet the quorum q. The work also provides a set of primitives at the system level to support automation.

The work emphasizes joint-ownership of objects. It does not address other requirements related to collaborative systems, such as the issue of multiple, dynamic roles, collaboration rights, and flexible ownership semantics. It also does not address the issue of high-level specification. On the other hand, because the model is an extension of the basic matrix model, it meets the flexibility requirement.

2.3.9 Summary of Previous Work

In summary, previous work in access control does not fully address all the requirements we identified. The following table summaries the capabilities provided by the previous work. In the table, a blank entry stands for "not meeting the requirement".

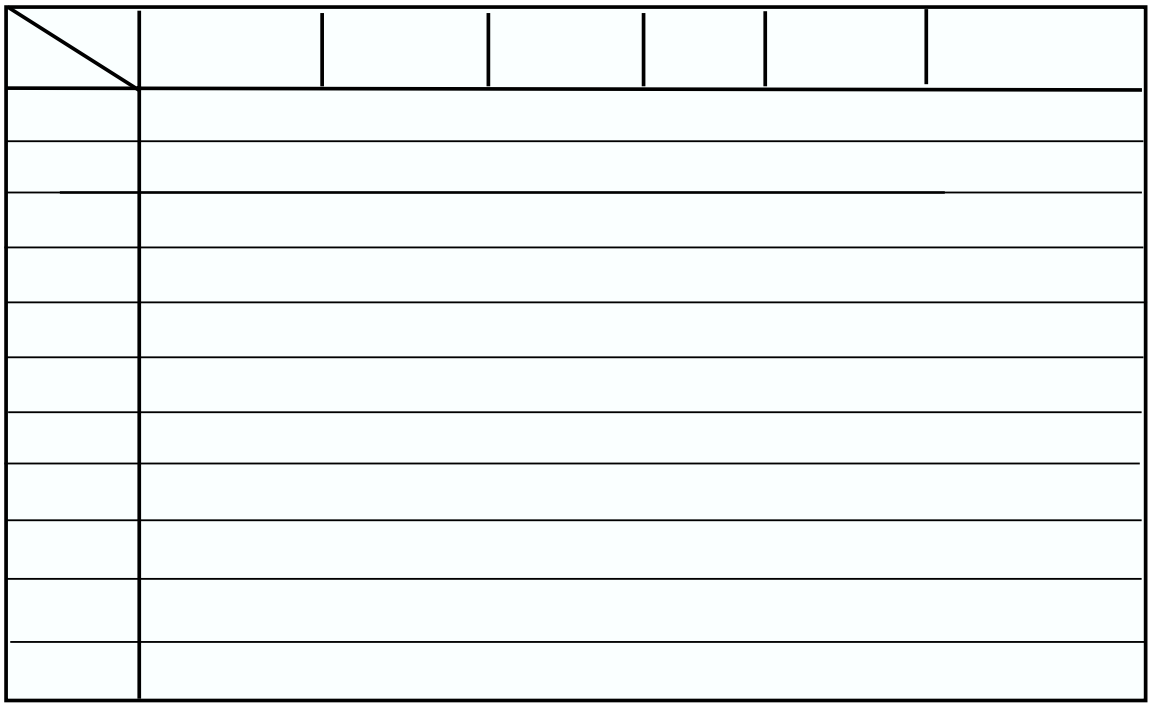

Figure 2.8 Summary of previous work.

3. THE ACCESS CONTROL MODEL

3.1 Overview

As mentioned in the first chapter, the components of our approach for meeting the requirements include a generic set of collaboration rights, variable-grained specification of access rights along the object, subject, and access right dimensions, and a set of inheritance and conflict resolution rules. In this chapter, we describe these components, motivate our design choices, and compare them with alternative design choices.

Before getting into the details of the model, we first give an overview of the model. Our model is an extension to the classic matrix model. Although the simple matrix model is inadequate for meeting the requirements as discussed in the previous chapter, it is the most basic one upon which other models are built. It supports the most primitive functionality of an access control model: with appropriate configuration of an access matrix, the access needs of any system can be defined. Therefore we choose the model as our extension base.

One of the disadvantages of the matrix model with respect to access specification is that it requires users to enter access definitions explicitly for all the granted rights in the matrix. To reduce the specification effort, our model extends the matrix model by allowing only part of the matrix to be specified explicitly and inferring the rest of the rights automatically.

We define the protection state of our model as a 4-tuple (S,O,A,F), where S is the set of subjects consisting of both individual users and their groupings, O is the set of objects consisting of individual objects and their groupings, A is an extended access matrix that stores both explicitly specified access definitions and those that are inferred automatically, and F is a function used to infer rights that have not

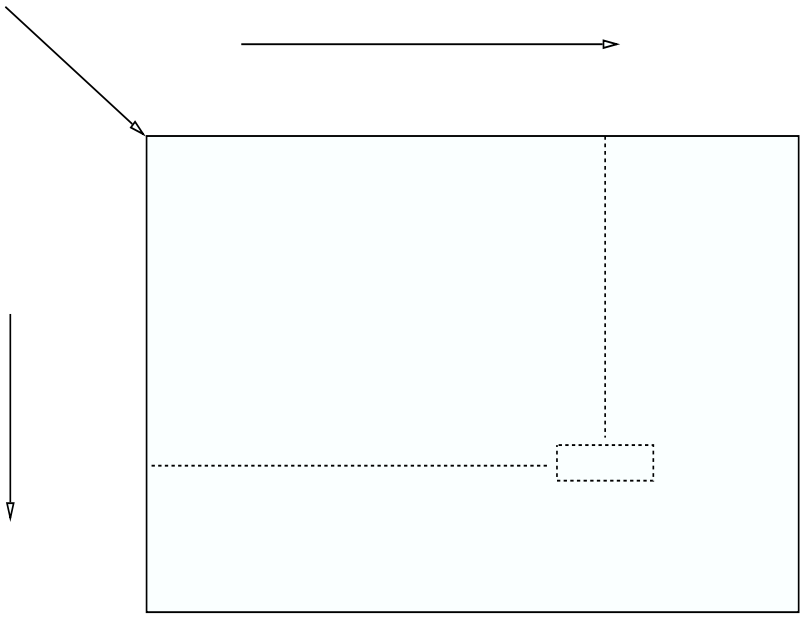

Figure 3.1 Definition of the new model.

been explicitly specified. As shown in figure 3.1, a cell A[s,o] in the access matrix represents the permission a subject s has over an object o. The permission can be an individual right corresponding to a single operation. It can also be a right group corresponding to a group of related operations. Moreover, the permission can take the form of a positive right or a negative right. A positive right is an explicit granting of a permission. A negative right is an explicit denial of a permission [95]. The inference function F takes as argument a (subject, object, right, A) tuple and returns the value {`True, False, Undecided`}.

To check the access privilege r of subject s over object o, the system first looks up the matrix entry A[s,o]. If A[s,o] contains an explicitly specified right, then access is granted or denied accordingly. Otherwise access is granted only if F(s,o,r,A) evaluates to true. In our model, F is defined by a set of inheritance and conflict resolution rules. These rules use the semantic relationships among the subjects, objects, and rights. They are defined for individual object, subject, and right dimensions and also for the cross product of these dimensions. By using these rules, the model allows a more specific definition to override a more general one.

To illustrate our approach, consider the example of the software development application. Assume that we want to associate the following access definition with a private comment line: "make the comment line readable to all members of the `suite` group except `hhs`". We will first define a group `suite` consisting of user `pd, rxc, hhs`, etc. We will then give a positive read right to the `suite` group and a negative read right to user `hhs`. Now if user `hhs` tries to read the comment line, his request will be rejected because a negative definition is explicitly defined in the matrix. On the other hand, all other members of the `suite` group will be able to read the line because, according to our inference rules, their rights can be inferred from the positive right defined for the `suite` group in the matrix.

The rest of the chapter is organized as follows. In section 3.2, we motivate the use of both positive and negative rights, and discuss how the access checking rule is defined. In sections 3.3, 3.4 and 3.5, we motivate and describe the basic elements of

the three dimensions of the extended access matrix and the associated access rules in detail. In section 3.6, we discuss access rules that involve multiple dimensions. Finally, in section 3.7, we describe our solution to the access administration problem, including how to support flexible ownership and flexible delegation and revocation of access rights.

3.2 Matrix Definition and Access Checking Rule

Positive and Negative Rights

In conventional systems based on the matrix model, it is generally assumed that it is only necessary to specify in the matrix the access rights a subject has. If a matrix entry is empty, then the subject is denied the access to the object. A right that explicitly grants permissions is called a positive right. Rabitti et al [95] introduce the notion of a "negative right" to facilitate access specification for object-oriented database systems [68, 65]. A negative right is an explicit denial of a permission and complements the notion of a positive right. In models supporting both positive and negative rights, a subject can be denied access either because it has no right over the object or has a negative right over the object. Rabitti et al have shown that the notion of a negative right is a powerful tool when permissions are inherited in object hierarchies. To illustrate its power, consider the software development application again. Assume a user `hhs` is to be denied read access to a comment line of a 1000-line-long program. Instead of granting him the positive read right over the 999 other lines, it is simpler to grant him a positive read right over the program and a negative read right over the comment line.

By using negative rights, exceptions becomes part of specifications. Therefore, the notion can be used to support easy specification not only in the object dimension as has been done in object-oriented databases[95], but also in the subject and right dimensions. To illustrate the use of negative rights in the subject dimension, consider a scenario in the software development application involving dynamic user

roles. Suppose user `rxc` adds a comment line to a program that criticizes some code another user `hhs` writes. User `rxc` would like to grant the read access of the comment line to all the users taking the `suite` role except `hhs`. If only positive rights were used, user `rxc` would have to identify all `suite` users other than `hhs` and grant them the read right individually. This approach suffers from two related disadvantages: First, it is painful for `rxc` to specify this definition if the list of `suite` members is long. Second, since users may take and relinquish the `suite` role dynamically, the above specification may become invalid later. With the notion of a negative right, the above can be achieved by granting a positive read right to `suite` and a negative read right to `hhs`. In addition, the use of negative rights can reduce the time required to search the inheritance space, as will be discussed in section 3.6.

Extended Matrix Definition and Access Checking Rule

An entry in our extended matrix takes three forms: positive, negative, or inferred. Positive and negative rights are explicitly specified. Inferred is the default setting of the matrix, and is represented by a blank entry. It designates inference from function F. The access checking rule of our model is defined as follows:

Access Checking Rule: *A subject s has access privilege r over object o if and only if*

a). A[s,o] contains a positive right r.

or

b). A[s,o] does not contain a negative right r, and F(s,o,r,A) evaluates to true.

While our rule makes inference the default setting, it is possible to define other rules that make denial or granting of rights as the default. We discuss below these alternatives and compare them.

Consider first the alternative of letting users explicitly specify granting and inference, and interpreting the default blank entry as the denial of a right. Accordingly, the access checking rule will be modified as follows:

Alternative Access Checking Rule: *A subject s has access privilege r over object o if and only if*

a). A[s,o] contains a positive right r.

or

b). A[s,o] specifies inference and F(s,o,r,A) evaluates to true.

This approach does not require the notion of negative rights, but still allows the deduction of implicit rights. However, it may be awkward to specify a matrix configuration using the approach. Our assumption is that with the help of F, the majority of rights can be inferred. Using the above approach, the users would have to explicitly specify for these rights an entry of "inferred", which is tedious. Continuing with example of the software development application, assume that user `hhs` is to be denied read access to a private comment line of a 1000-line-long program. It would be tedious if users have to specify for all lines except the comment line a matrix entry of "inferred".

The final alternative is to let users explicitly specify the denial and inference of a right, and interpret the default null entry as the granting of a right. It also suffers the same disadvantage with regard to specification since it requires explicit specification for "inferred".

Therefore, the best way to ease the specification problem is to let "inferred" be the default setting and allow users to specify overridings to this default setting. In fact, this is in the spirit of the approach used by the classic matrix model. The assumption of the matrix model is that users need specify only a few overridings (positive rights), and the rest of the matrix entries are the denial of rights. Therefore, the basic matrix model takes "denial" as the default setting and allows users to specify overriding (grant) to the setting. In our model, the default setting of the access matrix is "inferred", while overridings take the form of either grants (positive rights) or denials (negative rights), which are symmetric to the default setting.

3.3 Object Dimension

In this section, we address two issues related in the object dimension: What are the protected objects in a collaborative system, and how can we provide a specification model in the object dimension that meets the flexibility and ease of use goal? We will first list and discuss all the objects that need to be protected. We will then address the specification issue by defining a set of inheritance and conflict resolution rules. Finally, we will discuss how to handle exceptions to the inheritance rules.

Protection Objects

The set of protected objects depends on the collaboration model on which an access control model is based. As described in chapter 2, the collaboration model decides what the interactive objects of the system are, how users interact with them, and how users share these objects. From the total meditation principle, every object that users share should be protected since one user's operation on a shared object may affect others. We list below all the objects defined by our collaboration model that are shared and consequently need to be protected by the access control model.

- active variables

  As described in chapter 2, our collaboration model supports the notion of active variables, which are variable-grained program data structures that users can edit to trigger actions. They are shared in that multiple users can edit and change their values, and therefore are protected.

- interaction variables

  Interaction variables are copies of the active variables that users manipulate. They are associated with a set of attributes that represent their properties, such as their formatting and coupling settings. The coupling model allows the values and attributes of interaction variables of multiple users to be coupled. Therefore, we protect the manipulation of values and attributes of interaction variables.

- value groups

  Value groups are sets of related interaction variables of a user that share some common properties. Like interaction variables, value groups of multiple users can be coupled to share properties, and are therefore protected.

- object windows

  An object window is a screen area in which copies of active variables are displayed and manipulated. User's object windows are also protected objects since they can be coupled to share their cursor positions, scroll bars, window positions and window sizes.

- sessions

  A session is a series of interactions by multiple users with a shared application. It is shared in the sense that by joining a session, users are allowed to share the objects defined in the session. Activating, joining, leaving and terminating a session therefore has impact on other users and is controlled by making sessions as protected objects.

- roles

  A role has a unique id and a list of members who take the role. It is associated with a set of privileges. By taking and leaving roles users will acquire or abandon their access rights. Therefore, creating, joining, leaving, and deleting roles can affect others and is protected by making roles as protected objects.

- access matrix

  The access matrix is a protected object since it contains explicitly specified access definitions shared among all users.

Supporting the protection of the above shared objects introduces the problem of easy specification. In particular, fine-grained protection of active variables and interaction variables introduces a potentially large number of protected objects. In the

rest of this section, we will discuss how to define a high-level and flexible specification model for the protection of these variables.

Protection of Active Variables and Interaction Variables

The notion of active variables and interaction variables are closely related because the latter is the local buffer of the former. Therefore, one approach to address the specification problem is to treat both kinds of variables as one entity and associate the same access rights with them. Under this approach, if a user `hhs` is denied the right to modify a function, he will also be denied the right to modify his local buffer of the function. This approach, though simple, is inflexible since it does not meet the requirement of supporting optimistic access control described in chapter 1. An alternative, more flexible approach is to allow access definitions to be associated with both active variables and interaction variables, but allow the interaction variables to inherit their access definitions from corresponding active variables. Under this approach, even if users are denied the rights to do certain operations on active variables, they still have the option to do these operations on their local buffer. Returning to the above example, it is possible to give `hhs` the right to modify the interaction variable so that he can modify the local buffer of the function and communicate the changes to other users or save them to a local file, although he is not allowed to commit these changes to the active variables. In the above case, optimistic access control is supported. Alternatively, `hhs` can be denied the right to modify his local buffer because he does not have the right to modify corresponding active variables, that is, pessimistic access control is supported. A drawback with the above approach is that users have to worry about the specification for two different kinds of protected objects.

The approach we have taken is simpler than the second approach, but it is still able to support both optimistic and pessimistic access control. If interaction variables are isolated, access control on them is disabled. As a result, total autonomy of access control is granted for individual, isolated users, as long as their operations do not affect

each other. However, if users' interaction variables are in anyway coupled with other users' interaction variables, then their access to the interaction variables is controlled. Furthermore, we assume that the access control definitions are associated with the corresponding active variables to ensure that once coupled, interaction variables share the same access definitions so that consistency of access definitions can be maintained. Returning to the above example, user `hhs`, when isolated from other users, will be able to modify his own interaction variables. However, after he is coupled with another user `rxc`, he loses this ability since the access definition associated with the active variable will not allow him to do so. Compared with the second approach, our approach is simpler since users do not have to worry about the specification for two different kinds of access objects.

We discuss below how pessimistic and optimistic access control can be simulated by this approach. We define a `WriteR` right which determines whether users are allowed to modify their local buffers of an active variable. In addition, we introduce another right `UpdateR`, which determines whether users are allowed to update (commit) the changes of their local buffers to the active variable. Moreover, we predefine the relationship "`UpdateR` implies `WriteR`"[1], which specifies that the right to change an active variable implies the right to change its local buffer.

To simulate pessimistic access control, no access specification is made for the `WriteR` right. As a result, when users attempt to make changes to their local buffers, they succeed only if they have been given the `UpdateR` right because the `WriteR` right is automatically inferred from the `UpdateR` right by the relationship. To simulate optimistic access control, a positive access specification is explicitly defined for the `WriteR` right. In this way, users are allowed to change their buffers even if they do not have the `UpdateR` right.

In the rest of the thesis, we assume that users are all coupled with one another in some way, and therefore access control over their interaction variables is needed. Since, in such cases, we define that their interaction variables will have the same access

[1]The semantics of the relationship will be defined formally in section 3.4. Intuitively, the relationship specifies that the system can infer a positive `WriteR` from a positive `UpdateR`.

definitions as the active variables, we will not distinguish between active variables and the interaction variables. Therefore, from now on, we will treat only active variables as protected objects, and will not use the term "interaction variable" unless it is necessary to make the distinction.

Access Specification for Active Variables

As discussed before, active variables are variable-grained program data structures that multiple users can edit to trigger actions. These data structures include integers, real numbers, characters, subranges, enumerations, arrays, sequences (variable length arrays), records, unions, and pointers. To illustrate how we include active variables in our protection model, we use the example of the software development application. In this application, active variables are the name of the program being edited and tested, and the functions defined in the program. The declaration in figure 3.2 describe these active variables. In the figure, type `Text` is defined as a sequence of character strings. A sequence is an array of variable length consisting of two parts: a number specifying the length and an array holding the contents. Type `Func` is a structure consisting of a function name and its text body. The `Funcs` type is defined as a sequence of type `Func`. The active variables defined above are `program_name`, which is of `Sting` type, and `funcs`, which is of `Funcs` type.

To meet the requirement of flexibility, we allow the protection of not only the top-level active variables, but also their components, and the components of their components, and so on. For example, as shown in figure 3.3, it is possible for user `hhs` (right windows) to specify an access definition to prevent user `rxc` (left windows) from eliding a particular function `getvalue` in the sequence of functions. Even finer-grained protection is possible, as also shown in the figure where user `rxc` is denied read access to a comment line in the function `getvalue`.

Fine-grained specification requires users to specify and the system to store an access specification for each active variable, which is tedious and inefficient for most applications. This problem is similar to the problem encountered in specifying the

```
typedef char * String;

typedef struct {
    unsigned int num_of_lines;
    String *line_arr;
} Text;  /* Text = Sequence of String; */

typedef struct  {
    String name;
    Text text;
} Func;

typedef struct {
         unsigned int num_of_funcs;
         Func * func_arr;
} Funcs; /* Funcs = Sequence of Func; */

String program_name;

Funcs funcs;
```

Figure 3.2 Declaration of active variables

Figure 3.3

Fine grained specification. User `hhs` (right windows) denies `rxc` the elide right to the function `getvalue`. Thus, `rxc` (left windows) gets an error message when he tries eliding it. The figure also shows an even finer-grained specification: `hhs` protects a comment line from being read by `rxc`.

large number of format properties for active variables[32]. To support easy specification of these format attributes, Dewan [32] introduced the notion of value groups, which group values of active variables according to their types, contexts, siblings, ancestors, and other properties. The format attributes can be associated with not only active variables, but also their value groups, which are inherited by their members. This approach is also used for specifying the coupling attributes of active variables [37].

While the notion of value groups were initially defined to support easy specification of formatting and coupling definitions, we have found them also appropriate for specifying access definitions. We describe below various value groups defined in [32] that are useful for access specifications, and illustrate their usage in access control using examples from the software development application.

- **Generic Group:**

  The generic group includes all active variables of the application. The access definitions associated with this group are shared by all active variables. For example, if we deny "by-standers" the write right to the generic group, then they can change neither the program name, nor any functions in the application.

- **Type Groups:**

  Type groups include both predefined and user-defined types. An example of a predefined group is `integer`, which contains all integer values. An example of user-defined group is `Func`, which contains all values of that type and the descendents of these values.

  To illustrate the advantage of making type groups protected objects, assume that a `formatter` is allowed to change the fonts of all the text in a program. This specification can be achieved by giving the `formatter` the `FontR` right to the type group `String`.

- **Instance Groups:**

Instance groups are defined for individual values. A simple value is associated with a simple instance group, while a structure value is associated with a structure-instance group, and includes the value and its descendents. They can be used, for example, by `rxc` to associate a specification with a specific comment line that denies `hhs` the read right on it. It can also be used by `rxc` to associate a specification with a function that denies `hhs` the delete right of the function, and all the lines within the function.

- **Context Groups:**

  A context group is associated with variables appearing within a record, an array or a sequence, that is, in a particular "context". Among the context groups defined in [32], we find the record field context group particularly useful for access specifications. A record field context group contains all variables that appear as a particular field of a record of a particular type. To illustrate, consider the case when we want to deny `abc` the right to read the body of any function except its names. This can be easily specified by associating a negative read right with `abc` for the context group `Func.text`.

- **Child Groups**

  They are associated with the children of structure variables. These include fields of records and elements of sequences and arrays. In particular, a structure-type is associated with a type-child group that groups all children of structures of that type. An instance-child group is associated with a structure value that groups all descendents of that structure. As an example of using the child group, assume that `rxc` is to be granted the right to elide all the functions. This access intention can be specified by giving `rxc` the elide right over the instance child group of the variable `funcs`.

The above is only a partial list of all the value groups defined in [32]. Besides these value groups, the following value groups are also defined:

- **Constructor Groups:**

  They are associated with the constructors for creating arrays, sequence records, enumerations, and pointers. For example, "record" is a constructor group.

- **Array and Sequence Context Group**

  They include, for each sequence or array type, the list-first, list-last, and list-middle groups, which may have special formatting needs.

- **Pointer Reference Context Group**

  They include all variables referred to by pointers of a particular type.

While the above value groups are useful for specifying formatting properties, we have not found them useful for the specification of access definitions. For instance, the array and sequence context group divides an array into the first, middle, and last part, which is useful in situations where they may have distinctive formatting properties. However, we have not found situations in which it is necessary for them to have distinctive access needs. Because it is currently not clear to us whether all the above value groups will become useful for access specifications in practical situations, we choose to support them by default and will leave it open whether they are practical for access specifications.

Conflict Resolution

Because a value can belong to multiple value groups, and therefore can inherit attributes defined in these groups, it becomes necessary to resolve conflicting attribute definitions. Dewan ([32]) uses two kinds of relationships to arrange the inheritance hierarchy among value groups: IS-A is used to define the type inheritance and IS-PART-OF is used to define the structure inheritance among value groups. Figure 3.4 illustrates an inheritance hierarchy using an example from the software development application. In the figure, we assume that function 1 consists of three lines l1, l2, and l3, and function 2 consists of two lines l4 and l5.

l1 l2 l3 l4 l5

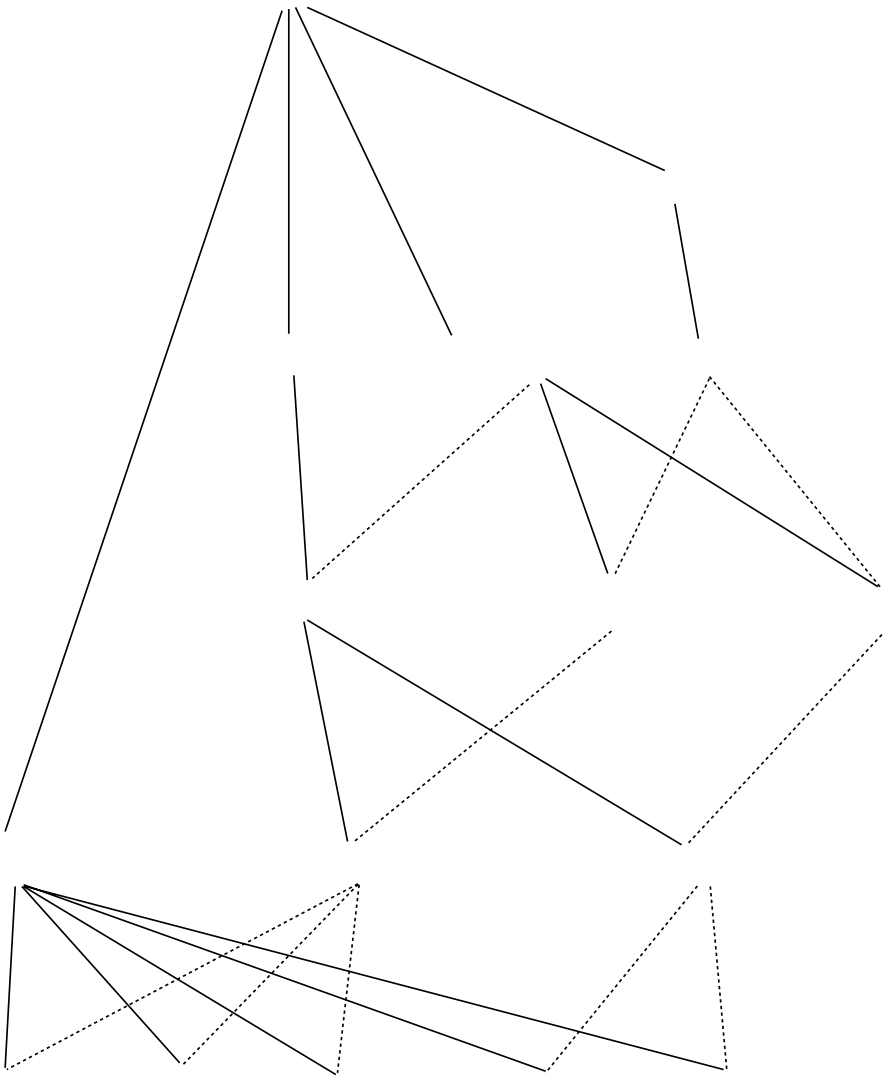

Figure 3.4 Inheritance hierarchy of value groups. In the figure, concrete lines are used to represent type inheritance and dotted lines structure inheritance.

As shown in the figure, it is possible for a value group to have two parents [2]. To resolve the multiple inheritance problem, the notion of an inheritance directive is introduced in [32], which determines from which of these groups an attribute should be inherited and the order in which it should be searched. The directive can be `none`, in which case the inheritance is not used, `structure-only`, in which case the IS-PART-OF link is used, `type-only`, in which case the IS-A link is used, `structure-first`, in which case the IS-PART-OF is given preference, and `type-first`, in which case the IS-A is given preference. The notion is also used in our access model since it provides a general and flexible way to resolve the multiple inheritance problem. In addition, we observe that in many applications, users create an object by inserting a new member into a structure variable, and they typically want the objects in the structure to have the same access right by default. In the software development application, for example, users create a line or a function by inserting a member into a function or a program that are structure variables. The inserted line and function should inherit their access information from their structure parents so that they have the same access definitions by default. In a file system, creating a file can be regarded as inserting a member into the directory structure. By default, the file should inherit its access list from its structure parent, i.e., the directory, to allow, for instance, directories of private files to be created easily. Therefore we have chosen `structure-first` as our default inheritance directive because we expect this default setting to be frequently used by applications. This default setting, however, is reconfigurable by each individual applications to tailor their access needs.

We summarize the above discussions by defining the following inheritance rule for the object dimension:

Object Inheritance and Conflict Resolution Rule: *A right r of subject s on object o is inherited from the value groups containing o that are chosen by the inheritance directive, i.e., $F(s, V(o, r), r, A) \rightarrow F(s, o, r, A)$ and $F(s, V(o, r), -r, A) \rightarrow$*

[2] It is shown in [32] that at most two parents are possible for a value group.

$F(s, o, -r, A)$ where V(o,r) is the set of value groups specified by the inheritance directive associated with o and r, r is either a positive or a negative right, and $\rightarrow$ denotes "imply". In case of conflicts, the access definition in the first value group chosen by the inheritance directive is used.

An alternative approach to resolve the conflict problem is to check the consistency of all the access definitions and disallow any conflicts. This alternative, though simpler, is inappropriate in an environment in which the set of access objects can be very large and dynamic. Whenever a new access definition is entered, potential conflicts could arise. Checking all these conflicts is costly [3] and sometimes even unnecessary when a particular user may need to access only a small portion of a large set of objects. Thus, it is not only too rigid, but also computationally expensive to adopt the approach of simply disallowing all conflicts. The above argument also applies to the resolution of conflict problems in the access and subject dimensions discussed later.

Exceptions to Structure Inheritance

The object inheritance rule specifies that a variable should inherit rights down from its structure or type parents. In some situations, it is also useful to synthesize rights up to a parent, that is, check the rights of the children before doing operations on the parents. To illustrate, consider the delete operation over structure variables. Assume that a user `hhs` decides to delete a function `getvalue,` which he can delete since he created it. Besides the code added by `hhs`, however, the function `getvalue` also contains some comments another user `rxc` has created. To prevent the comments from being erased accidently by others, `rxc`, as their owner, has specified that all users be granted a negative delete right to these comments. Therefore, even though user `hhs` has the delete right to the function, he should not be allowed to delete the comments added by `rxc`. In this situation, we notice that the right should not only be inherited down from the parent objects, but also be synthesized up from the child objects.

[3] As will be described later in section 3.5, allowing multiple, dynamic roles will also introduce conflicts and makes it even more expensive to check the consistency of all access definitions.

Similar observations also apply to those operations such as elide and hide, which, when applied to a structure variable, affect both the variable and its descendents.

To support such synthesis for the above operations, we define the semantics of these operations as follows. For a structure variable, the operation consists of two steps. The first step does the operation on each elements of the variable, with access rights checked each time the operation takes place. If the first step succeeds, that is, *all* operations on the children are successful, then the second step is taken to execute the operation on the whole structure, with the access right checked.

We use the above example to illustrate these semantics. If `hhs` wants to delete the function `getvalue`, the operation is first applied to all members (lines) within the function, which will be successful only if appropriate access right allows the deletion of each member. Since `hhs` does not have the delete privileges over the comment lines owned by `rxc`, these comment lines cannot be deleted at the first step. Therefore, the second step to delete the function is not allowed to be taken. These semantics are inspired by the rmdir command in Unix – a directory cannot be removed unless it is empty, i.e., all the files within it are removed.

3.4 Access Dimension

The requirement for supporting collaboration rights demands that all operations whose results can affect multiple users be protected by collaboration rights (section 1.1). In this section, we first identify these rights, and then address how an easy to use specification model can be provided in the access dimension.

We list below all the access rights on the protected objects identified in section 3.3. Each of these rights corresponds to an operation on an object defined by the Suite collaboration model. Thus, unlike systems such as Unix in which several operations such as insert, append, modify, and delete share the same right (the write right), each operation in our model has its own right to meet the flexibility requirement.

Rights on Active Variables

For active variables, the following access rights are defined:

- The `ReadR`, `WriteR`, `InsertR`, and `DeleteR` rights, which determine whether a user can read a variable, write a variable, insert an element into a structure variable, or delete an element from a structure variable.
- The `ElideR`, `HideR`, and `SelectR` rights, which determine whether a user can elide, hide, or select a variable.
- The `TitleR`, `IdentR`, `FontR`, and `ColorR` rights, which determine whether a user can change the title, indentation, font, or color of a variable.
- The `ValueCoupledR` right, which determines whether a user can share value changes to a variable with another user.
- The `ViewCoupledR` right, which determines whether a subject can share viewing properties of a variable with another user.
- The `FormatCoupledR` right, which determines whether a subject can share formatting properties of a variable with another user.
- The `TransmitRawR`, `TransmitParsedR`, `TransmitValidatedR`, and `TransmitCommittedR` rights, which determine whether a user can transmit raw changes, syntactically correct changes, semantically correct changes, or committed changes of a variable to another user.
- The `TransmitIncrementR`, `TransmitCompleteR`, `TransmitPeriodR`, `TransmitTimeR`, and `TransmitTransmitR` rights, which determine whether a user can transmit changes to a variable to another user incrementally, on completion, periodically, after a certain time, or explicitly.
- The `ListenRawR`, `ListenParsedR`, `ListenValidatedR`, and `ListenCommittedR` rights, which determine whether a user can listen raw changes, syntactically correct changes, semantically correct changes, or committed changes of a variable from another user.

- The `ListenIncrementR`, `ListenCompleteR`, `ListenPeriodR`, `ListenTimeR`, and `ListenTransmitR` rights, which determine whether a user can listen changes to a variable from another user incrementally, on completion, periodically, after a certain time, or explicitly.

Rights on Roles

For roles, the following access rights are defined:

- The `CreateRoleR` and `DeleteRoleR` rights, which determine whether a user can create a new role or delete an existing role. Creating a new role will allow users to join the role thereafter, while deleting a role will delete the role, including its member list.

- The `TakeRoleR, LeaveRoleR, AddMemberR` and `RemoveMemberR` rights, which determine whether a user can take (join) a role by adding itself to the member list of the role, leave (relinquish) a role by removing itself from the member list, add a user into the member list of a role, or remove a user from the member list of a role, respectively.

- The `ReadRoleR` and `ModifyRoleR` rights, which determine whether a user can read or modify the informations associated with a role, including its name and member list.

Rights on Sessions

The following access rights are defined for sessions:

- The `CreateSessionR` right, which determines whether a user can create a new session. After a session is created, it will be able to accept subsequent incoming connection requests from the participants.

- The `DeleteSessionR` right, which determines whether a user can delete an existing session.

- The `JoinSessionR` right, which determines whether a user can join a session.
- The `RemoveParticipantR`, which determines whether a user can remove a participant from a session.
- The `ReadSessionR` and `ModifySessionR` rights, which determine whether a user can read the information related to a session such as its current participants, or change the information of a session.

Rights on Object Windows

The following access rights are defined for object windows:

- The `CursorR` right, which determines whether a user can move the cursor around the object window.
- The `ScrollR` right, which determines whether a user can scroll the contents of the object window.
- The `ResizeR`, and `MoveR` rights, which determine whether a user can change the size or the position of the object window.

Introducing a large set of rights aggravates the problem of specification and storage of rights. To ease the problem in the access dimension, we introduce the notion of *right groups*. They are defined by classifying the individual rights into several logical categories. Each category consists of rights that users are likely to give or deny as a group. For example, the `TransmitRawR`, `TransmitParsedR`, `TransmitValidatedR` and `TransmitCommittedR` rights all specify the rights to transmit values according to their correctness level, therefore we put them in the right group `TransmitCorrectR`. The `ReadR`, `WriteR`, `InsertR`, and `DeleteR` rights are all included in the `DataR` right group, because they all correspond to the operations on the value of a variable. The `ElideR`, `HideR` and `SelectR` rights all correspond to the operations that change the view of an object, therefore they are included in the right group `ViewR`.

Because of the number of the rights involved, it is not sufficient to group rights within one level of hierarchy. Therefore, we also allow right groups to contain other right groups, thereby allowing the right groups to form a tree structure. For instance, the right group `TransmitR` includes the right groups `TransmitEventR`, `TransmitCorrectR`, `TransmitPeriodR`, and `TransmitTimeR`. Its member groups specify the rights of transmission based on the criteria of event, semantic correctness, period, and time. The `TransmitR` is in turn included as a member of the right group `CoupleR`. At the root of the tree is the right group `AllR`, which includes all the individual rights and right groups. Note that unlike an individual right, a right group does not correspond to any specific operation on an object.

To allow the rights to be granted or denied as a right group, we define the transitive *include* relationship among them. This relationship is used in the following rule:

Right Inheritance Rule: *A positive or negative right r of subject s on object o is inherited from the right groups it belongs, i.e.* $F(s, o, R, A) \rightarrow F(s, o, r, A)$ *and* $F(s, o, -R, A) \rightarrow F(s, o, -r, A)$ *where R includes r.*

A set of predefined *include* relationships are provided by our model, which are listed in appendix B. Some of these relationships are shown in figure 3.5.

We have restricted the *include* relationships to form a tree structure, instead of a more general lattice structure. That is, a right can belong to at most one right group instead of multiple groups. Conceptually, the semantics of the *include* relationship does not restrict the structure to be a tree, as long as there is no recursive definition of the relationships. However, while a lattice structured *include* relationship is more general and logically feasible, we have not found typical situations where it is useful for a right to be in multiple right groups. Therefore we currently do not support the lattice structured *include* relationship.

To illustrate the use of the above inheritance rule, consider the following scenarios in a classroom examination application. Assume that students `hhs`, `rxc` and `abc` are to be allowed to couple with the teacher `pd`, but not with one another. To prevent unauthorized coupling between user `hhs` and `rxc`, it is only necessary for user `pd` to

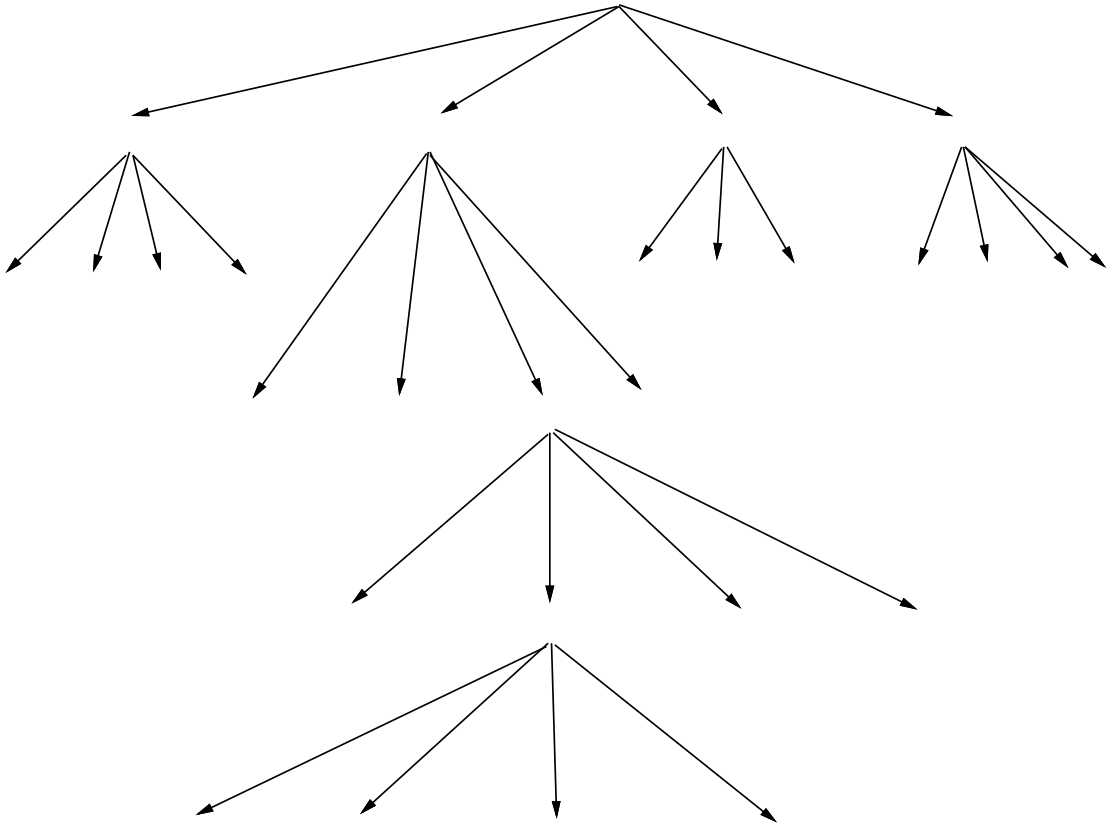

Figure 3.5 Some *include* relationships among access rights.

grant `rxc` and `hhs` a negative `TransmitR` right to each other. Without right groups, it would be necessary to grant the negative right for each individual transmission right, which is tedious.

The notion of a right group can be used in combination with negative rights to further simplify the specification task. To illustrate, assume that user `rxc` would like to grant all the data rights (Read, Write, Insert, and Delete) to user `hhs`, except the delete right. He can achieve this by giving `hhs` a positive `DataR`, and a negative `DeleteR`. Without these notions, it would be necessary to grant the read, insert, and write rights to `hhs` individually. Fine-grained access rights together with right groups gives users fine-control over collaboration while relieving them from the task of tediously specifying all the rights individually.

In addition to the *include* relationship described above, we also define the *imply* relationship between rights to support easy specification. This relationship was first proposed in [49] for protection of read, write, insert, and delete rights in traditional database system. It models the fact that the right to do a more powerful operation should guarantee the right to do a less powerful one. For instance, the right to do the write operation should guarantee the right to do the insert operation since the write operation is considered more powerful than the insert operation. This relationship is used in the following rule:

Right Inference Rule: *A right r of subject s on object o, if undecided, is inferred from the rights that imply r, i.e.,* $F(s, o, rx, A) \rightarrow F(s, o, r, A)$ *and* $F(s, o, -r, A) \rightarrow F(s, o, -rx, A)$ *where rx implies r.*

Though originally defined for database systems, the notion of right implication is crucial for the more complex collaborative systems, for two reasons. First, it reduces the problem of specifying a large number of rights necessary in a collaborative system. Second, in a complex system, it is more likely that users may specify conflicting access definitions because they have to specify, often dynamically, a large number of access rights. The *imply* relationship can be used by the system to detect some of these conflicts automatically. For example, if a write right is to be granted to a user who

has been previously denied the read right explicitly, the system can detect the conflict because the negative read right may imply the negative write right.

We have identified several useful *imply* relationships among the access rights we support. Previous work has defined the imply relationship among the `ReadR`, `WriteR`, `InsertR` and `DeleteR` as follows: `WriteR` *implies* `InsertR`, `InsertR` *implies* `ReadR` and `DeleteR` *implies* `ReadR`. We describe and justify below the additional relationships we have identified.

For the rights over the active variables, we further define:

- `HideR` *implies* `SelectR`, and `ElideR` *implies* `SelectR`

  because users have to select an object before they can elide or hide it.

- `TransmitRawR` *implies* `TransmitParsedR`, `TransmitParsedR` *implies* `TransmitValidateR`, and `TransmitValidateR` *implies* `TransmitCommittedR`

  because if users can transmit arbitrary raw changes to a variable, they can certainly transmit changes that are syntactically correct. Likewise, if users can transmit syntactically correct changes, they can certainly transmit the semantically correct changes, which must be syntactically correct. In addition, if users can transmit semantically correct changes, they can transmit the committed changes, which must be semantically correct [4].

- `ListenRawR` *implies* `ListenParsedR`, `ListenParsedR` *implies* `ListenValidateR`, and `ListenValidateR` *implies* `ListenCommittedR`

  because if users can receive arbitrary raw changes to a variable, they can certainly receive changes that are syntactically correct. Likewise, if users can receive syntactically correct changes, they can certainly receive the semantically correct changes, which must be syntactically correct. In addition, if users can receive semantically correct changes, they can receive the committed changes, which must be semantically correct.

[4] The observations on these operations are borrowed from the work on flexible coupling [37].

- `TransmitIncrementR` *implies* `TransmitCompleteR`, `TransmitCompleteR` *implies* `TransmitPeriodR`, `TransmitPeriodR` *implies* `TransmitTimeR`, and `TransmitTimeR` *implies* `TransmitTransmitR`

  because if users can transmit arbitrary changes incrementally, they can certainly transmit changes on completion, periodically, after a certain time, or explicitly.

- `ListenIncrementR` *implies* `ListenCompleteR`, `ListenCompleteR` *implies* `ListenPeriodR`, `ListenPeriodR` *implies* `ListenTimeR`, and `ListenTimeR` *implies* `ListenTransmitR`

  because if users can receive arbitrary changes incrementally, they can certainly receive changes on completion, periodically, after a certain time, or explicitly.

For the rights defined over user roles, we define:

- `AddMemberR` *implies* `TakeRoleR`

  because if users are allowed to add arbitrary subjects into the member list of a role, they should also be allowed to add themselves into the member list.

- `RemoveMemberR` *implies* `LeaveRoleR`

  because if users are allowed to remove arbitrary subject from the member list of a role, they should also be allowed to remove themselves from the member list.

- `ModifyRoleR` *implies* `CreateRoleR`, `ModifyRoleR` *implies* `DeleteRoleR`, `ModifyRoleR` *implies* `AddMemberR`, and `ModifyRoleR` *implies* `RemoveMemberR`

  because the modify operation allows all of the above operations.

- `CreateRoleR` *implies* `ReadRoleR`, `DeleteRoleR` *implies* `ReadRoleR`, `AddMemberR` *implies* `ReadRoleR`, and `RemoveMemberR` *implies* `ReadRoleR`

  because we allow users to look at the role information before they do the above operations.

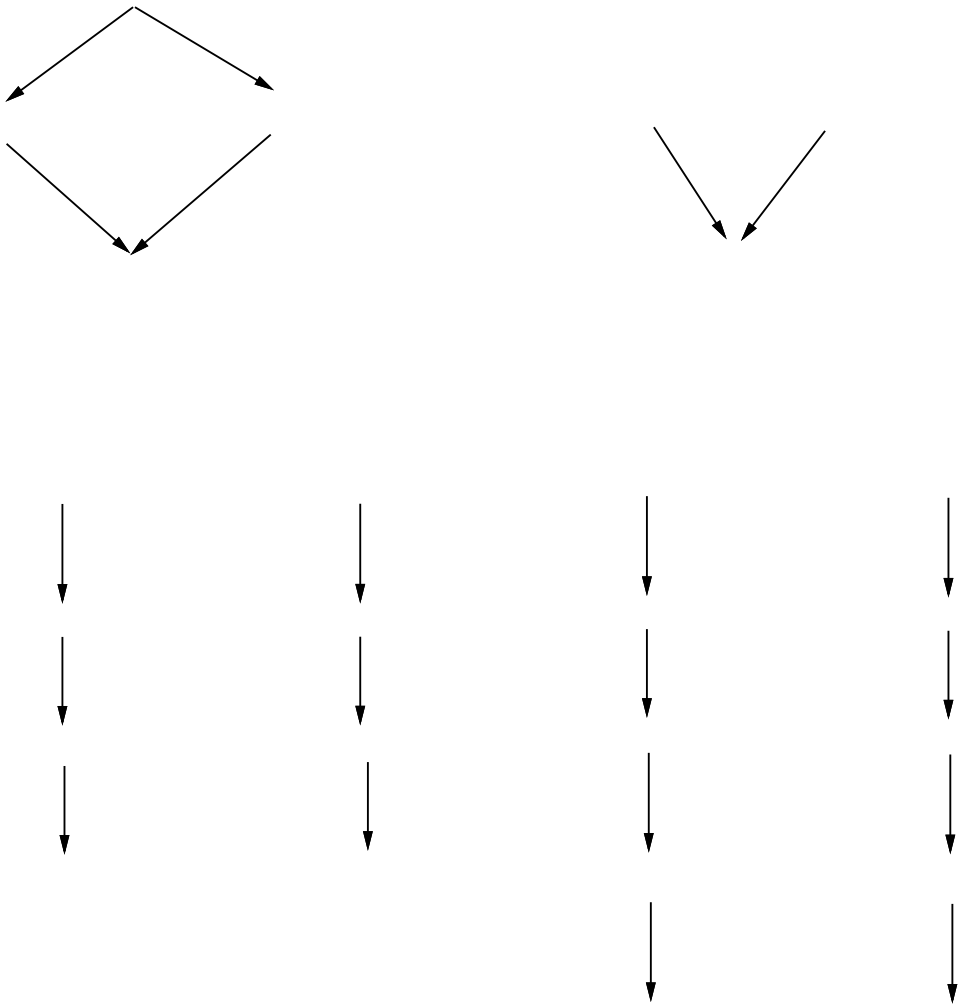

Figure 3.6 Some *imply* relationships among access rights defined over active variables.

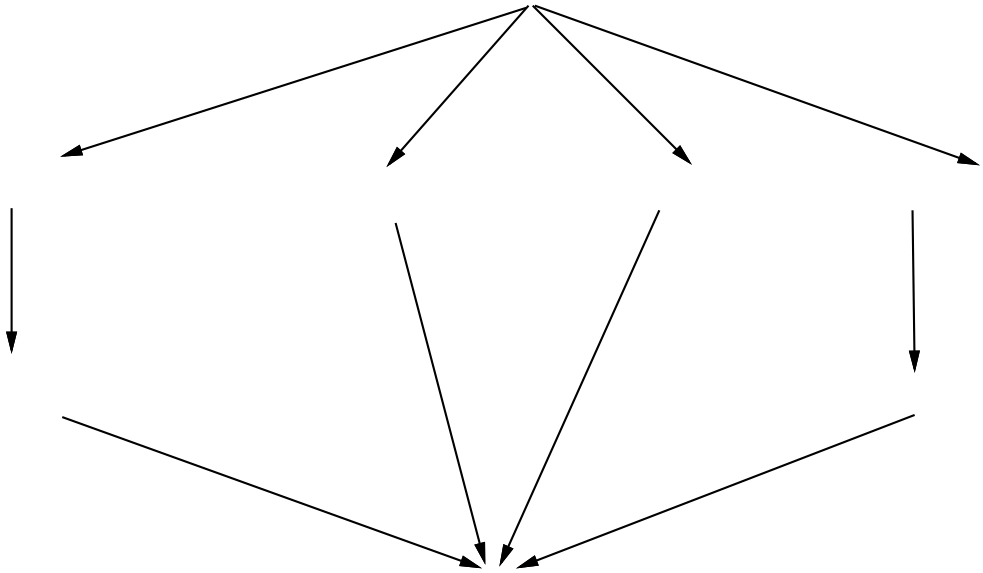

Figure 3.7 The imply relationships over role rights.

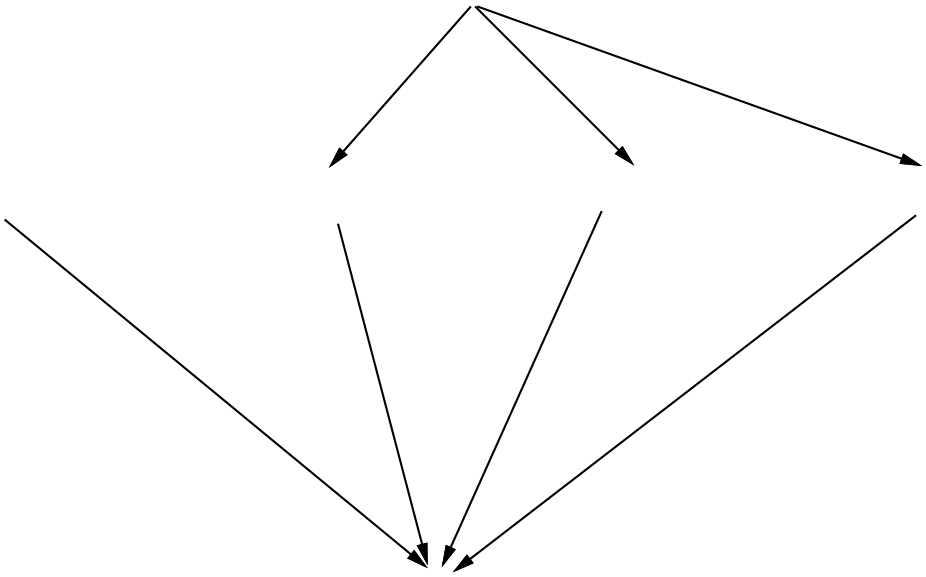

Figure 3.8 The *imply* relationships over session rights.

For the rights defined over sessions, we define:

- `ModifySessionR` *implies* `CreateSessionR`, `ModifySessionR` *implies* `DeleteSessionR`, and `ModifySessionR` *implies* `RemoveParticipantR`

  because the modify operation allows all of the above operations.

- `CreateSessionR` *implies* `ReadSessionR`, `DeleteSessionR` *implies* `ReadSessionR`, `JoinSessionR` *implies* `ReadSessionR`, and `RemoveParticipantR` *implies* `ReadSessionR`

  because we allow users to look at the session information before they do the above operations.

Conflict Resolution

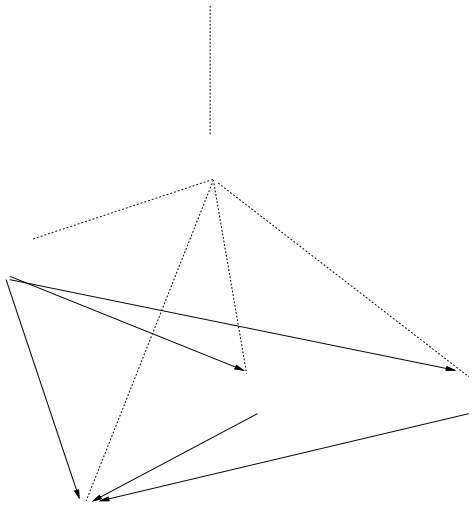

Figure 3.9 Conflict of *imply* and *include*.

Supporting both the *include* and *imply* relationships allows an access right to be inferred from multiple sources. To illustrate, consider a scenario in which a new user

`abc` joins the `serc` role and is not familiar with the system. To ensure security, `hhs` gives `abc` the right to insert lines (positive `InsertR`) into a function he just created, but denies `abc` the data rights (negative `DataR`) on the function so that he cannot delete or modify the existing text of the function. We have a conflict since the negative `ReadR` right can be inferred from the first definition while the positive `ReadR` right from the latter (figure 3.9). In such situations, the *imply* relationship is used first. This is because the *imply* relationships are defined over the individual rights and exist even if right groups are not defined. From another perspective, right groups can be regarded as mere macro conventions. Moreover, the *imply* relationship is introduced for both easy specification and right consistency, while the *include* relationship only for easy specification. Thus, rights are more tightly related by the *imply* relationship than the *include* relationship. Therefore, in the scenario above, the *imply* relation is used to grant `abc` the read right.

The following conflict resolution rule summarizes the above discussion:

Right Conflict Resolution Rule: *The imply relationship is used in preference to the include relationship in case of conflicts.*

To summarize this section, we define a set of generic collaboration rights based on a general collaboration model. The presence of a large set of access rights, however, brings the problem of easy specification, efficient storage, and potential inconsistencies. To cope with this problem in the access dimension, we support two kinds of inheritance relationships, *include* and *imply*, to deduce new rights from existing rights and make consistency checks. These relationships can conflict, and we have motivated and defined a conflict resolution rule to address the problem.

3.5 Subject Dimension

This section will address two issues that are analogous to the ones discussed in the previous section: which subjects should we support, and how should we define a specification model in the subject dimension to meet the flexible and easy specification requirements.

A subject is an entity that initiates an access request to a protected object. As in conventional systems, a user such as `pd` or `hhs` is treated as a subject in our model. In addition, a role is treated as a subject in our model. A role is an abstraction that bears a unique name (id) and is associated with a member list. Since it is also a subject, it is also associated with a set of access rights. For example, the role `suite` bears a unique name `suite`, has a member list of {`pd, rxc, hhs`}, and is associated with a set of access rights. A member of a role can be a user, or another role. We say that a user or role "takes" a role if it appears in the member list of the latter. Furthermore, we define that if s takes the role of S, then s inherits all the access definitions associated with S. Formally, we use the following rule to define the semantics of *take*:

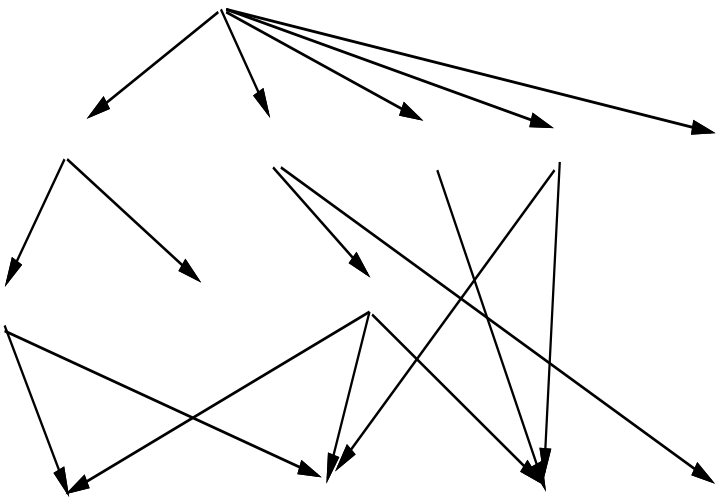

Figure 3.10 *Take*: inherit all rights of a role.

Subject Inheritance Rule 1: *Subjects inherit both positive and negative rights from the take relationship, i.e., if s takes the role of S, then* $F(S, o, r, A) \rightarrow F(s, o, r, A)$ *and* $F(S, o, -r, A) \rightarrow F(s, o, -r, A)$.

Because we allow multiple users to take a role, the notion of a role includes the concept of a user group supported by traditional systems such as Unix. Moreover, in accordance with the requirement of supporting multiple, dynamic user roles, we also allow a role to take multiple roles simultaneously, and change the taking of roles dynamically. For instance, during a code reviewing process, a person may take the roles of both chairman and narrator. After the inspection of one module, he may relinquish the chairman role and take the author role. Therefore, unlike traditional user groups that usually allow only static, simple grouping of users, roles allows grouping of users in a more flexible and dynamic way.

To illustrate the use of the above defined inheritance rule, we continue with the example of the software development application. If we allow the `faculty` role to insert new functions into the program but explicitly deny the `student` role from doing so, then `pd` gets the right when he takes the `faculty` role. On the other hand, `hhs` does not have the right because he takes the `student` role. Without the notion of roles, it would be necessary to associate an access definition for every user that takes the `faculty` or `student` role.

As discussed in the previous section, we support a set of coupling rights. For example, we support the `ValueCoupledR` right, which decides whether a subject (e.g. `hhs`) is allowed to couple the value of a function `getvalue` with another subject (e.g. `student`), and the `IncrementTransmitR` right, which decides whether a subject (e.g. `hhs`) is allowed to transmit the value of an object to another subject (e.g. `rxc`) incrementally. Unlike other conventional rights such as read and write, these rights involve two subjects instead of one. More precisely, one subject is also treated as an object. We refer to the second subject (`student` and `rxc` in the above examples) as the "passive subject". To ease the specification, we allow users to define the right to couple with a group of passive subjects rather than only individual subjects, and allow inheritance to be used to infer the right to couple with individual passive subjects. From another perspective, if we regard (obj, passive subject) as a complex object, then the grouping of passive subjects allows the grouping of the corresponding complex

objects. We use the *take* relationship to also group passive subjects, as defined by the following rule:

Subject Inheritance Rule 2: *If r is a coupling right involving passive subjects s and s2, and s2 takes the role of s, then* $F(s1,(o,s),r,A) \rightarrow F(s1,(o,s2),r,A)$ *and* $F(s1,(o,s),-r,A) \rightarrow F(s1,(o,s2),-r,A)$.

To illustrate, assume that a user `hhs` is granted the `ValueCoupledR` right to a complex object (`getvalue`, `teacher`), that is, `hhs` is allowed to couple the value of the function `getvalue` with the role `teacher`. According to the above inheritance rule, `hhs` also has the right to couple with user `pd` because `pd` takes the role of `teacher`. Likewise, the denial of `hhs` to value-couple the function with the `student` role implies the denial of `hhs` with user `rxc` because `rxc` takes the role of `student`.

We also support another inheritance relationship among subjects, which is useful in maintaining access privileges. It is often useful to allow a subject to take a subset of the positive access rights (privileges) of others. For example, a `PhDStudent` may have all the positive data rights of `MSStudent`, and `pd` (manager) may have all the positive data rights of `rxc` and `hhs` (employees), as illustrated in figure 3.11. We therefore define a relationship, *have*, which treats positive access rights as privileges and supports inheritance of selected privileges. Formally, *have* is defined as a mapping from $S \times R \times S$ to { *true, false*} where S is the domain of subjects and R is the domain of positive rights. We support inheritance of selected rights rather than all the rights because it reflects the "least privilege" principle [101] more faithfully. The following inheritance rule defines the semantics of *have*:

Subject Inheritance Rule 3: *If subject s1 has right r of subject s2, i.e., have(s1,r,s2) = true, then* $F(s2,r,o,A) \rightarrow F(s1,r,o,A)$ *where r is a positive right.*

Conflict Resolution

Allowing a user to take multiple roles in the above manner introduces potential conflicts. Continuing with the example of the software development application, assume that `serc` is granted the positive right and `student` is granted the negative

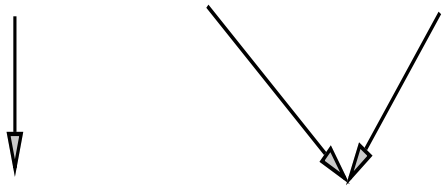

Figure 3.11 *Have*: inherit selected privileges.

right for some object. How should we decide the right for `rxc` who takes the role of both `student` and `serc`?

To motivate our solution, let us turn to some more examples. First, we observe that the *take* relationship actually defines a lattice structure which decides which role is more "specific". For instance, `PhDstudent` is more specific than `student`, and `student` is more specific than `all`. In the spirit of object-oriented languages, we use more specific roles in preference to less specific roles. As an example, assume that user `rxc` takes the roles of `all`, `student`, and `PhDstudent`. Further assume that we give a positive right to the roles `all` and `PhDstudent` and a negative right to the role `student`. Under these assumptions, `rxc` is granted the right because `PhDStudent` is the most specific role he takes. Thus, we derive our first rule in resolving conflict:

Subject Conflict Resolution Rule 1: *A more specific role defined by the take relationship should be used first.*

However, it is possible for a user to take two roles with different rights, neither of which is more specific than the other. As an example, assume that the `serc` role is granted a positive right and the `student` role is granted a negative right. Further assume that there is no *take* relationship between the two roles. As illustrated in figure 3.12, the effects of the above specification on subjects taking both roles can be interpreted as either positive or negative. In this situation, we require that users give a "hint" that indicates which roles should be used first. In particular, we provide an

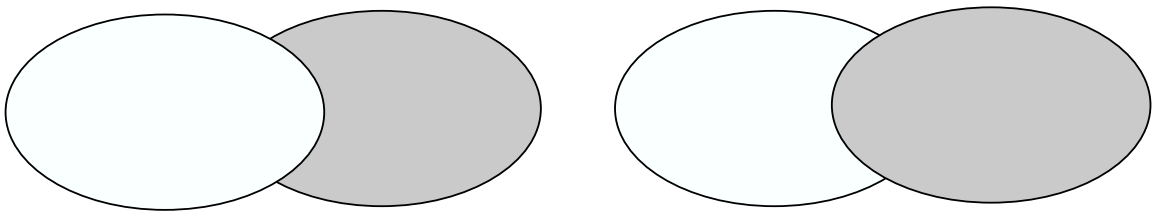

Figure 3.12 Conflicts due to multiple roles.

ordered access list of the form $(\omega_1 S_1 \omega_2 S_2 ... \omega_n S_n)$ where ω_i is either + or -, denoting positive right or negative right respectively, and S_i is a role. We interpret the list to be "position sensitive". That is, the position of the roles in the list is regarded as a "hint" indicating which roles should be used first. Following the heuristic of "putting important things first", we use roles appearing earlier in the list in preference to those appearing later. According to this rule, `-student+serc` will give negative rights to those taking both roles while `+serc-student` will grant them positive rights. This is our second rule in resolving conflicts:

Subject Conflict Resolution Rule 2: *In case of conflicts, a hint specified by users is used to determine which role should be used first. In particular, we choose the access definition that appears earliest in the access list.*

By adopting the above rules, our approach provides a simple yet flexible solution to the conflict problem. At one extreme, all the negative definitions can be put before the positive ones to ensure maximal security, as in the case of `(-student-capo+faculty+serc)`. At the other extreme, all the positive ones can be put before negative ones to ensure maximal sharing, as in the case of `(+faculty+serc-student-capo)`. Between these two extremes, users can adjust their access needs by putting subjects at different positions.

The above two rules specify how to resolve conflicts due to the *take* relationship. Conflicts can also arise when a negative right is inferred from the *take* relationship

and a positive right is inferred from the *have* relationship. In our algorithm, the *take* relationship is used in preference to the *have* relationship for the following two reasons: First, the *take* relationship can infer both positive and negative rights, which is safer than the *have* relationship as the latter can infer only positive rights. Second, the *take* relationship infers rights a user inherits *directly* from the roles he takes, while the *have* relationship infers rights a user gets *indirectly* from other roles which he may not take. Therefore, we define the following rule:

Subject Conflict Resolution Rule 3: *The take relationship is used in preference to the have relationship in case of conflicts.*

To summarize, allowing multiple, dynamic user roles introduces the problem of how to deduce users' rights naturally and efficiently from the roles they take. We address the problem by defining a set of inheritance rules that allow both the *take* and *have* inheritances and conflict resolution rules that are position sensitive and choose *take* over *have*.

3.6 Multi-dimensional Inheritance

In the above sections, we have described multiple inheritances in the object, access right, and subject dimensions. There is another important question we have to answer: in what order are the inheritances in the three dimensions used when resolving a right?

Conceptually, we can view the multi-dimensional inheritance from a single dimension by treating other dimensions as subdimensions of nodes in the viewed dimension. For example, we can view multi-dimensional inheritance within the framework of object dimensional inheritance. To decide the value of (s,o,r), we first find o and check whether it is possible to infer the value of (s,o,r) from the access control list associated with o, by using inheritance in subject and right dimensions if necessary. If not, we proceed by using inheritance in the object dimension. In this way, we regard the object dimension as the prime dimension, and the subject and right dimensions as

sub-dimensions. Similarly, we can also regard the subject dimension as the prime one and treat the object and right dimensions as sub-dimensions.

Figure 3.13 illustrates the above discussion by showing how the search looks like if the object dimension is treated as the prime dimension and the access and subject dimensions are treated as sub-dimensions. In the figure, we assume that each object is associated with an access control list, and that obj1 may inherit its access rights from obj2, while obj2 may inherit from obj3 and obj4, as shown in the upper graph. The lower graph shows that we start the search first from obj1 (object dimension). If we cannot infer the right by using the access control list associated with obj1 and inheritance within subject and right dimension (pane 1, within the subject-access sub-dimension), then we proceed by using inheritance in the object dimension. That is, obj2 is chosen and above process is repeated, and so on.

Figure 3.14 describes the search algorithm when we regard the object dimension as the prime one, access dimension as the sub-dimension, and subject dimension as the sub-sub-dimension. We can see that the most outside loop is for the inheritance of the object dimension, then the right dimension, and the inner-most loop is for the subject dimension. Moreover, if a right cannot be inherited from all three dimensions, we return a negative right. This is more conservative than returning a positive right and therefore safe. From the algorithm, we can also see that negative rights help short-circuiting the search space, since it is not necessary to do an exhaustive search of the inheritance space to infer denial of access.

Different inheritance orders have different impact on the efficiency of the lookup, which in turn depends on the access data structures used to represent the access matrix. For instance, if all of the access definitions are associated with objects, i.e., access control list is used, then starting the search from the object dimension will be the most efficient one because it is easy to find out all the subjects and rights in the inner two loops, such as pane 1 and pane 2 in figure 3.13. On the other hand, if access definitions are associated with subjects, and we use the same search algorithm as above to start the search from objects, it will be costly to find out all the subjects

(b). Search starts from pane 1, then pane 2.

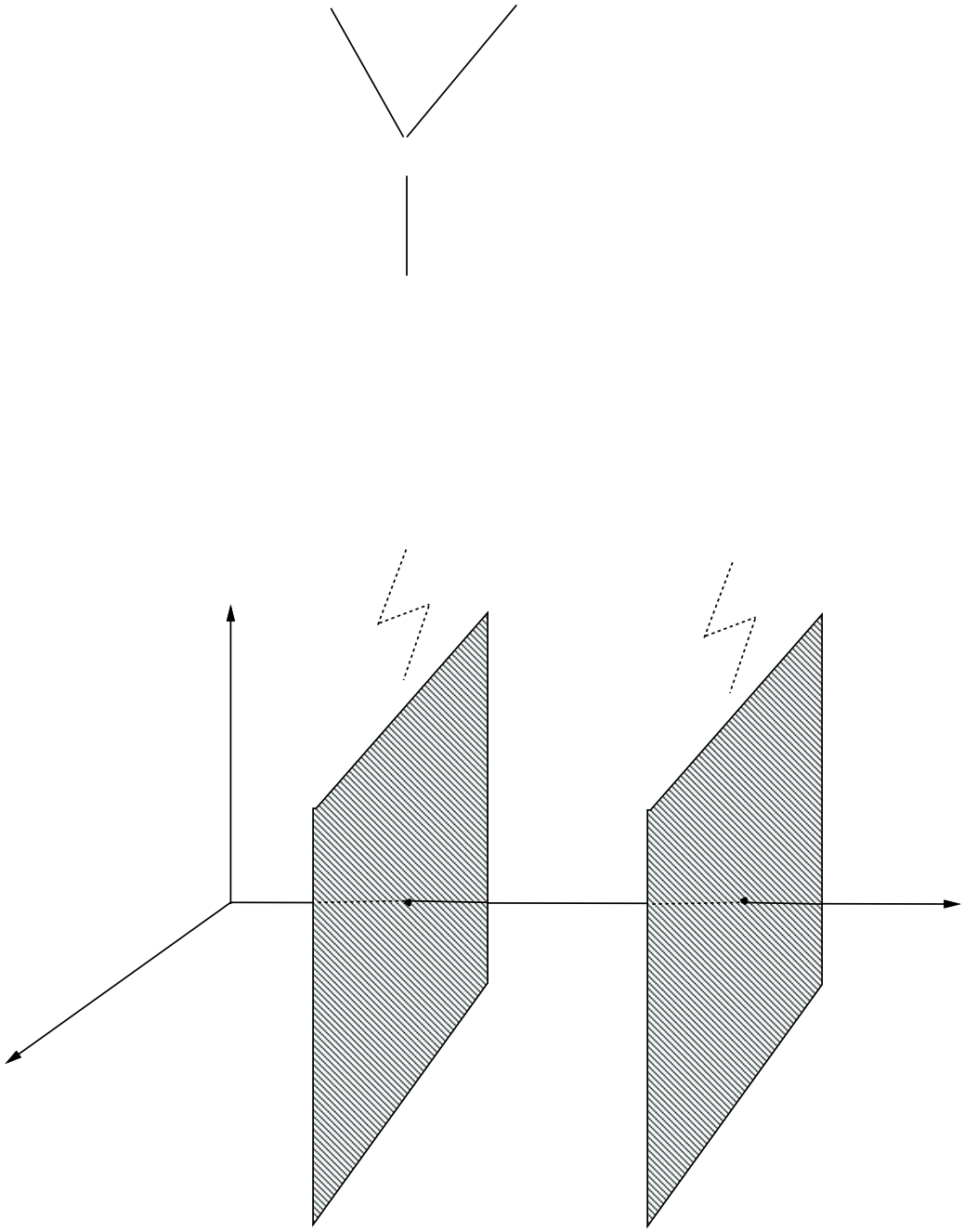

Figure 3.13 Multi-dimensional inheritance when the object dimension is treated as the prime dimension.

```
/* Multi-dimensional inheritance algorithm to check (s, o, r) = ? */

    loopO: for every object Ox such that F(s,Ox,r,A) -> F(s,o,r,A)
           as decided by the object inheritance rules and
           conflict resolution rules do {

       loopR: for every right Rx such that F(s,Ox,Rx,A) -> F(s,Ox,r,A)
           as decided by the right inheritance rules and
           conflict resolution rules do {

           loopS: for every subject Sx such that F(Sx,Ox,Rx,A) -> F(s,Ox,Rx,A)
           as decided by the subject inheritance rules and
           conflict resolution rules do {

               if A(Ox, Sx, Rx) = +/- return +/-;

           }
        }
    }
    return -;
```

Figure 3.14 Multi-dimensional inheritance algorithm

and rights in the inner loops associated with an object. It would require us to search the information associated with every subject to collect the access definitions related to a particular object, which would be inefficient. Although indexing could be used to relieve the problem to some extent, it will be far more straight-forward and convenient if we choose to search from the subject dimension in the first place. As an example, consider the simple case of capability systems that associate access definitions with subjects shown in figure 3.15. Assume that we want to find out whether subj1 has an R right over obj1. It would be inefficient to start the search from the object dimension, that is, to collect all the capabilities all the subjects have over obj1, and determine subj1's rights from these capabilities. Starting the search from subj1's capability list would be far more straight-forward and efficient. On the other hand, in systems that use access control lists, starting the search from the object dimension would be most efficient for similar reasons.

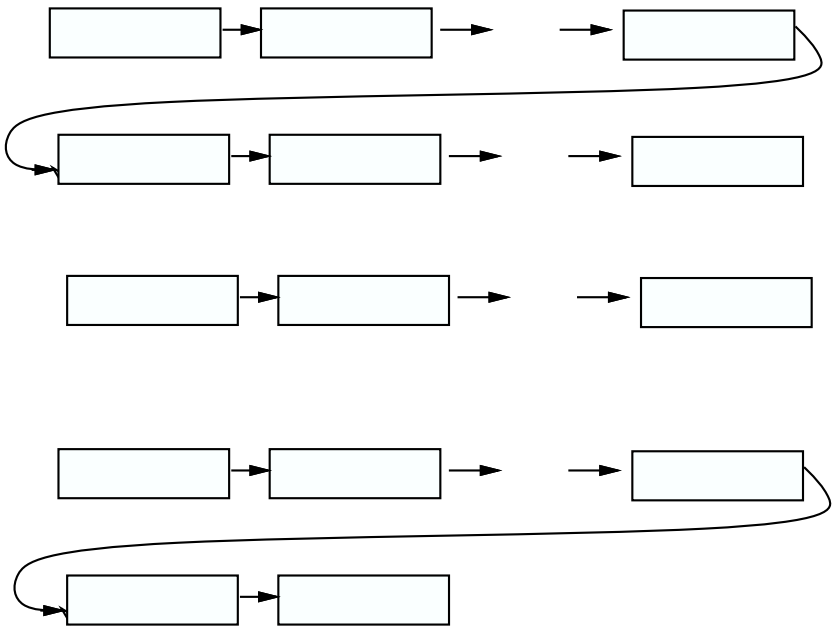

Figure 3.15 Example of capability lists.

In our model, we do not specify the exact multi-dimensional inheritance order. Instead, we give inheritance rules and conflict resolution rules in various dimensions, and let the system developers who use our model decide the prime and sub dimensions and data structures to represent the access matrix. As will be discussed in chapter 4, in the particular implementation of our model in the Suite system, we choose the object dimension as the prime dimension, access dimension as the sub-dimension, and subject dimension as the sub-sub-dimension, with the underlying storage mechanism of the access matrix being an extended access control list.

3.7 Access Administration

As discussed in section 3.3, the access matrix itself should be a protected object because otherwise users can penetrate the protection by changing the matrix arbitrarily. In this section, we address issues related to protection of the access matrix, which we refer to as access administration. In particular, we address the authorizer problem, that is, who should be responsible for specifying the access definitions. In addition, we discuss support for flexible access delegation and revocation.

3.7.1 Protection of Access Matrix

As mentioned in chapter 2, the classical matrix model uses the notion of *-rights for the protection of the access matrix. The *-rights are defined over the conventional rights. Only owners and *-right holders have the privilege to grant the corresponding rights. A simplified, coarser-grained version of this approach is adopted by systems such as Unix, where the administration of the access matrix entries is the sole responsibility of the owner. Compared with the more general approach, the Unix approach is easy to use, but is less flexible.

To meet the requirement of flexibility, we also use fine-grained *-rights and ownerships for the protection of access matrix entries. For each individual right R, we

define an R* right that protects the value of R [5]. Introducing a large number of *-rights brings the associated problem of easy specification and consistency checking. Therefore, we introduce the notion of *-right groups, and define a set of *include* and *imply* relationships over the *-rights to ease this problem.

To define the *include* and *imply* relationships among *-rights, we observe that the semantic relationships among conventional non-* rights also exist among their corresponding *-rights. For the *imply* relationship, the capability to grant others the right to do a strong operation such as write means the capability to grant others the right to do a weak operation such as read. Therefore from the "WriteR implies ReadR" relationship, we can draw a new relationship "WriteR* implies ReadR*". For the *include* relationship, we have grouped the *-rights into a tree hierarchy similar to the non-*-rights. Therefore the *include* relationship of non-*-rights is mapped into corresponding *-rights. For example, from "CoupleR includes TransmitR and ListenR", we can define "CoupleR* includes TransmitR* and ListenR*", and so on. We represent the above observation as the following two inheritance rules:

***-Right Inheritance Rule 1:** *The imply relationships among non-*-rights are mapped to the *-rights, i.e.,* $imply(R1, R2) \rightarrow imply(R1*, R2*)$ *where R1 and R2 are non-*-rights.*

***-Right Inheritance Rule 2:** *The include relationships among non-*-rights are mapped to the *-rights, i.e.,* $include(R1, R2) \rightarrow include(R1*, R2*)$ *where R1 and R2 are non-*-rights.*

[5]Currently, the * right is used to indicate the privilege to modify an access matrix entry. By incorporating additional *-rights, it is possible to extend the protection so that other operations such as read on the access matrix can be also controlled. For simplicity, we only discuss the modification *-rights in the rest of the chapter.

3.7.2 The Authorizer

The above discussion raises the question of "who should possess the *-rights?" This is called the authorizer problem. Although it is a policy issue specific to individual applications, we would like to provide system-level mechanisms to support these policies to meet the requirement of automation.

As surveyed in chapter 2, there are currently several approaches to defining the authorizer in conventional systems. The first approach uses the concept of ownership. The creator of an object becomes its owner and has all the rights to the object, including the right to grant these rights to other subjects. This is the approach defined by the classical matrix model. In another approach, a central administrator is responsible for specifying the access definitions, but does not have direct rights himself. This approach is used by some database systems such as INGRES in which the database administrator assumes the role of a security administrator. To meet the requirement of flexibility, we would like to provide mechanisms that are able to support all of the above authorizer policies. One possible approach is to follow Hydra's design philosophy. That is, we can discard the concept of ownership and let individual applications decide who are the authorizers. This approach, though flexible, does not provide any built-in mechanisms to assist application developers in constructing their protection subsystems.

Our solution to the authorizer problem is to support the concept of ownership, while at the same time providing mechanisms that allow applications to tailor their needs individually. This is in the spirit of the classical matrix model. However, as will be described in more detail below, the two works are different in that we support flexible definition of ownership semantics, and support dynamic change of ownership and multiple ownership.

In the rest of the section, we first define the semantics of ownership. We then discuss definition of owners of objects and support of dynamic change of ownership and

multiple ownership. Finally, we discuss simulation of various authorizer approaches by using the rules we devised.

Semantics of Ownership

To define the exact semantics of ownership, we observe that ownership of an object usually denotes a set of special privileges. Therefore, instead of treating the ownership as a concept totally alienated from other elements of the model, we treat the ownership as a special right. The exact meaning of the right is defined by a set of "imply" relationships reconfigurable by individual applications. We use the following rule to define the semantics of ownership:

Ownership Semantic Rule: *Ownership is a special right defined by associating a set of imply relationships with the right.*

As will be discussed shortly, this rule combined with other ownership rules will enable us to simulate existing authorizer schemes.

Owner Specification

We have devised a set of rules to define owners of protected objects. We use the following rule to define the initial owner of an active variable:

Ownership Initialization Rule: *A subject who creates an application owns all the initial variables defined within the application unless they have predefined owners. A subject who dynamically creates a new variable owns that variable.*

To illustrate, consider the software development tool. User `hhs`, who created the application, owns all the initial functions displayed by the tool since none of them have pre-designated owners. On the other hand, user `rxc`, who inserts new function `minit` into the program becomes the owner of the function.

The above rule associates a single owner with a variable since an operation such as insert that creates a variable dynamically is an atomic operation executed by a single user. To meet the requirements of dynamic change of ownership and joint-ownership, we associate an owner list with each object, whose contents can be changed dynamically. We use the following rule to define the semantics of an owner list:

Multiple Ownership Rule: *An initial owner of an object can add roles into or remove roles from its owner list. The owners of an object are users who take a role in the owner list.*

The above rule specifies that an initial owner as defined by the initialization rule can dynamically change the owner list. In particular, the initial owner can add more subjects into the owner list and therefore allow joint-ownership by multiple users. For example, user `hhs` can create a function `common` on behalf of the `suite` group. After creating the function and becoming the initial owner, he can add `suite` to the owner list associated with the function `common`. As a result, user `pd` also becomes an owner due to his membership in the `suite` group.

Because the owner list introduced above is defined over a shared object, it is a shared object itself and therefore should be a protected object according to the total mediation requirement. Therefore, we introduce a new right called `OlistR`, which is used to protect the owner list. A subject is allowed to change an owner list only if it possesses corresponding `OlistR`. In addition, we define a corresponding `OlistR*`. A subject is allowed to give the `OlistR` only if it possesses the `OlistR*`. We define the following rule to allow owners to change owner list dynamically:

Ownership Default Rule: *By default, ownership implies OlistR and OlistR*.*

Allowing dynamic change of ownership and joint-ownership aggravates the specification problem. To illustrate, suppose `hhs` owns the function `Getvalue`, and every line of it. Later, `hhs` would like to add `pd` as a co-owner of the function. If every line of the function is associated with an owner list consisting of `hhs`, it would be tedious and time-consuming for `hhs` to add `pd` to all these owner lists one by one. In such cases, we observe that although the ownership initialization rule specifies that the creator of a variable should own the variable, it is unnecessary to actually store the owner into the owner list of the variable if the creator also owns the variable's parent, such as its structure parent. Instead, inheritance mechanisms can be used to infer the ownership of the child from its parent. For the above scenario, the owner lists of all lines created by `hhs` will be set to empty, and their owner lists, being empty, will

be inherited from their structure parents, i.e., the function `Getvalue`. The advantage of using the above inheritance is the following: First, it makes it easy to change ownership and maintain consistency. For instance, it would only be necessary for `hhs` to change the owner list associated with the function `Getvalue`. Second, like other forms of inheritances we have seen before, it saves space by eliminating information redundancy. It is possible that the creator of a parent may also create many of its children, in which case the saving of space for the owner lists of the children could be substantial.

The above discussion shows the usefulness of supporting inheritance of ownership list from the structure parents. To ease the specification task, it is also useful to support inheritance from other value groups. For instance, we can define the owner list associated with the generic group of an application to contain only the creator of the application. Therefore by default, the creator owns all the initial variables defined within the application by the generic inheritance. Without inheritance, it would be necessary to define the owner list for each initial variables. Therefore, we support inheritance from all the value groups defined in section 3.3. That is, an owner list can be associated with any value group, and children of a value group can inherit its owner list from its parents, including structure parents.

Because an object can belong to multiple value groups, conflicts can arise from inheriting owner list from multiple value groups. To resolve the conflict, we adopt the same approach as used in section 3.3. That is, we use the notion of inheritance directives introduced by Dewan ([32]) to resolve the conflicts because of its generality and flexibility. In addition, for the same reason as discussed in section 3.3, we have chosen `structure-first` as the default inheritance directive for the owner list because we expect this default setting to be frequently used by applications. Nevertheless, this default setting can be changed by individual applications.

To summarize, we define the following ownership inheritance rule:

Ownership Inheritance Rule: *The owner list of an object o, if unspecified, is inherited from the value groups containing o that are chosen by the inheritance*

directive. In case of conflicts, the ownership list defined in the first value group chosen by the inheritance directive is used. By default, inheritance starts from structure value groups.

Supporting Existing Ownership Policies

The set of ownership rules defined above provides a flexible way to support existing ownership policies in addition to the new multiple ownership policy. We discuss below how these policies can be simulated.

To simulate the conventional usage of ownership such as Unix, we define "ownership implies DataR*" [6]. Therefore, the owner is entitled to change the access rights of the file he owns because of the privileges gained through the *-rights. In addition, depending on the value of the umask, we can set the corresponding access mode in the access list for the owner.

To simulate the INGRES-like centralized administration approach, the application can grant the administrator role a positive `AllR*` over the generic group. In this way, the central administrator is responsible for specifying all the *-rights over all the objects.

To simulate the classical matrix definition of ownership, we define "ownership implies AllR*".

To simulate the other extreme of the authorizer solution, i.e., the Hydra approach, we define no imply relationship for the ownership right. In this way, even though an owner is defined for each object, there is no semantics associated with the ownership and it entitles him no more and no less rights as the owner.

Supporting group ownership and letting the owner of an object change the owner list arbitrarily could be risky if an owner changes the owner list wantonly without considering its security consequences. Therefore, some applications may not want such feature. This can be realized in our model by disabling the rule "ownership

[6]Note that here we define the imply relationship on right group DataR*. However, it is to be interpreted strictly as a macro definition. That is, it is identical to the following specification: "ownership implies WriteR*, ownership implies InsertR*, ownership implies ReadR*", ..., and so on. Therefore, the discussion on the inheritance order of imply and include are still valid.

implies OlistR and OlistR*" using the programming primitives the system provided, which prohibits the owner from changing the owner list or granting the right to do the operation.

3.7.3 Delegation and Revocation

In the previous section, we described an approach of using flexible ownership for access administration. It allows owners of an object to act as authorizers and dynamically change access definitions depending on ownership semantics. In this section, we discuss how owners can distribute the above privileges to others. In our model, this capability is supported by a flexible access delegation and revocation mechanism.

As mentioned in chapter 1, delegation [1] is the ability of a user to distribute his privileges, including both *-rights and regular rights, to others. In the simplest form, a user A delegates all of his rights to another user B. To realize the above semantics in our model, we define a single-member role for each user. For every user X, its associated role(X) bears the same id as the user name, and by default has a member list containing only the user itself, that is, a user must take his own role. We further interpret the user id in the access matrix as its role id. To realize delegation, user A can add user B to the member list of Role(A), which initially contains only A itself. In this way, B will be able to inherit A's rights from the "take" inheritance rule. This kind of delegation, though simple, is not flexible. It does not allow users to give part of their rights to others. To meet the requirement of flexible delegation, we allow a user A to delegate selected parts of his access rights to trusted parties and the authority to further distribute these privileges on behalf of A. In our model, we support flexible access delegation in two ways. A simple way is for A to distribute his *-rights to others by changing the associated access lists. Another approach is to use the notion of indirect roles, which we will discuss shortly.

Related to access delegation is the concept of revocation, which is an operation that undoes the effects of delegation. It is important because it provides a way to

contain the spread of access rights. As mentioned in chapter 1, two possible semantics of revocation can be defined, namely shallow revocation and deep revocation, and both semantics might be useful depending on individual application needs.

Below, we first define the two semantics of revocation formally:

- **Shallow Revocation:**

  If X grants r* or r to Y, then shallow revocation just removes r* or r from Y. If Y has granted r* or r to Y1, Y1 to Y2, ..., and so on, then the r* and r rights of Y1, Y2, ... are not affected. This semantics is simple to understand, easy to implement, and is used by most existing systems.

- **Deep Revocation:**

  Deep revocation, on the other hand, tries to completely undo the effects of right delegation by revoking the r and r* rights of Y1, Y2, ... in the above case. That is, a system supporting it must recursively revoke authorization from those to whom the grantee granted the right. It has been defined by Griffiths and Wade [60] formally as follows: Let (G1, G2, ..., Gi-1, Gi, Gi+1, ..., Gn) be the sequence of grants of a specific privilege on a given object by any user, where each Gj is the grant of a single privilege. If i < j, then Gi occurred earlier than Gj. If Ri denotes the revocation of Gi, then the authorization state of the system after (G1, G2, ..., Gi-1, Gi, Gi+1, ..., Gn, Ri) should be identical to (G1, G2, ..., Gi-1, Gi+1, ..., Gn).

The implementation for supporting the total revocation semantics is non-trivial. Capability-based systems have used the notion of indirect keys to realize the semantics. Under this approach, subjects are given keys to the capabilities instead of the capabilities directly. Therefore, revocations can be realized by simply disconnecting the relationship between keys and capabilities. ACL-based protection systems usually do not support total revocation because of the complexity and expenses of the algorithm involved. The only exception known to us is System R, which records

each delegation with time-stamps and uses these stamps to realize the revocation. Although the above two approaches all support the deep revocation semantics, they are implementation dependent, that is, they depend on the particular data structures used to implement the access matrix. We describe below an approach to support the deep revocation semantics that is independent of the implementation of the access matrix.

Simulating Multiple Revocation Semantics

As a generic model, our model supports both semantics and lets the application decide which semantics it will use. By default, we expect applications to use shallow revocation since it is the simplest and most widely used one. Supporting the semantics of shallow revocation is simple and straight-forward in our model. The grantor can just remove the corresponding *-right or regular right from the revokee, or alternatively, grant a negative *-right or regular right to the revokee. On the other hand, supporting total revocation is more complicated. As described below, the approach we take requires users to use the notion of indirect roles to delegate rights rather than grant the rights directly.

Here is a brief description of the algorithm. Suppose X wants to give Y0 the R right on object Obj, and allow Y0 to distribute the right to other subjects, say Y1, Y2, ...Yn. Instead of giving Y0 the R and R* right directly, X first creates a role S, and grants S a positive R right to Obj. To give Y0 the R right, X can add Y0 to the member list of S, thereby allowing Y0 to inherit the R right from S. To grant Y0 the right to distribute the R right, X can give Y0 the `AddMemberRoleR` right of S, thereby allowing Y0 to add into S more members who can inherit the R right from S. Furthermore, if X wants to allow Y0 to distribute the right to distribute the R right, X can grant Y0 the `AddMemberRoleR*` right of S, so that Y0 can grant the `AddMemberRoleR` to other subjects.

Now, X can realize complete revocation by simply removing the R right from S. Because the R rights of Y0, Y1, Y2, ..., Yn all depend on the access rights of role S,

the above operation deprives Y1, ..., Yn's R rights too. In addition, it is also easy to realize shallow revocation by simply removing Y0 from the member list of S, and its `AddMemberRoleR` and `AddMemberRoleR*` rights of S. In this way, Y0's R right will be revoked, but Y1, Y2, ..., Yn's rights will not be affected.

To illustrate, let us return to the classroom examination application described in chapter 1. We mentioned that the teacher `pd` delegated his coupling rights to teaching assistant `rxc`, and later wants to revoke the delegation according to different revocation semantics depending on two possible scenarios. Instead of giving `rxc` the `CouplingR` and `CouplingR*` directly, `pd` can use the following process to realize the delegation of coupling rights. First, he creates a role `trustedPd` and grants `trustedPd` a positive coupling right. To give `rxc` the coupling right, he adds `rxc` to the member list of `trustedPd`, thereby allowing him to inherit the positive coupling right. Furthermore, to give `rxc` the right to transfer the coupling right to other parties, `pd` can grant `rxc` the right to add members into the role `trustedPd`. Consequently, `rxc` can further grant the coupling right to another user `hhs` by adding `hhs` to the member list of `trustedPd`. To realize the deep revocation semantics, `pd` can simply deny `trustedPd` the positive coupling right he previously issued, thereby denying both `rxc` and `hhs`'s coupling rights. On the other hand, `pd` can realize the shallow revocation semantics by simply removing `rxc` from the `trustedPd` member list.

We observe that although it is not necessary to realize our model using a capability list approach, the above indirect role approach is conceptually similar to the indirect capability approach in that they both use one level of indirection to grant a privilege rather than grant the privilege directly. Therefore, in both approaches it is easy to realize deep revocation by simply removing the link of indirection. With the indirect capability approach, a grantor gives users keys to a lock that guards some privileges. When the locks are changed, the original keys are rendered useless. Similarly, with the indirect role approach, a grantor gives users membership (analogue of key) to a trusted role that has the the privileges he wants to delegate. When the trusted role is

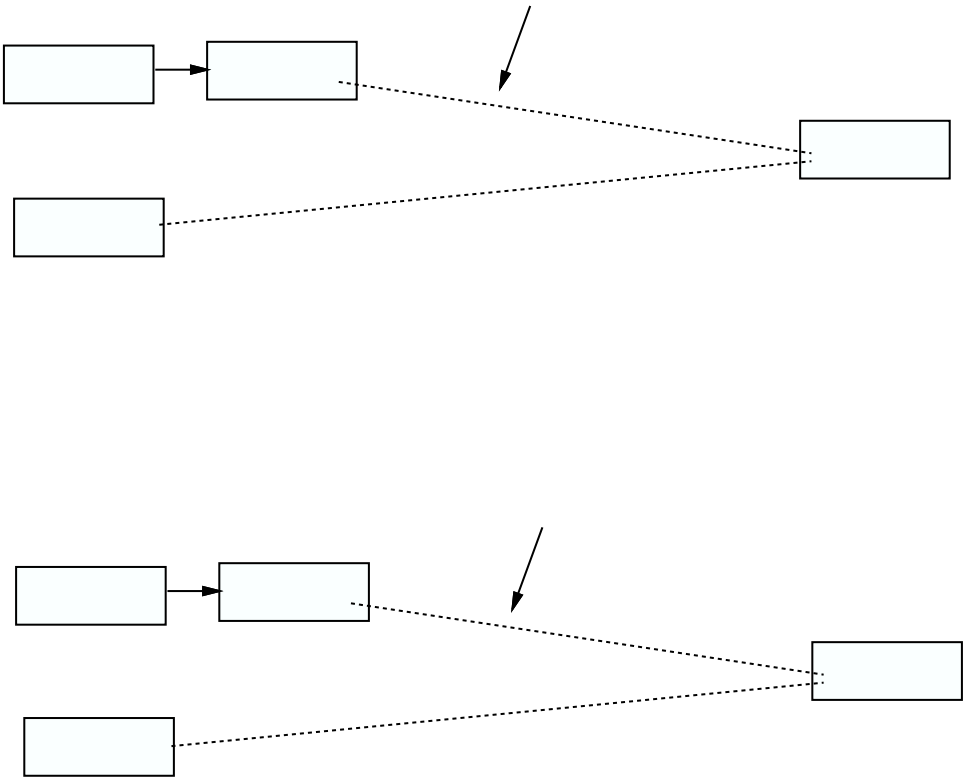

Figure 3.16

Comparison of indirect capabilities and indirect roles. Both support deep revocation by removing the indirection links.

deprived of the privilege, all the users that hold its membership also lose the privilege. The above discussion is illustrated in figure 3.16.

In our current implementation of the revocation mechanism, an authorizer has to follow the above process manually in order to realize the total revocation. It is possible to automate the above process to a single operation so that authorizers can have a simple and easy to use interface.

3.8 Programming Interface

In previous sections, we have discussed the semantics and specification of access control from the end-user's point of view. In this section, we discuss the specification from the programmer's point of view by describing how programmers can specify various aspects of access control for a particular application.

We have provided the following language constructs for programmers to specify access control. These language constructs are needed to meet the requirement of automation and extensibility.

- `Dm_AddRight` and `Dm_DelRight`, which add a new right, or delete an existing right.
- `Dm_AddInclude` and `Dm_DelInclude`, which add or delete a particular *include* relationship among rights.
- `Dm_AddImply` and `Dm_DelImply`, which add or delete a particular *imply* relationship among rights.
- `Dm_SetDirective`, which sets an inheritance directive associated with a particular right of a particular object.
- `Gm_MakeGroup`, `Gm_GetGroup` and `Gm_DeleteGroup`, which create a new role, get the definition of a role, or delete an existing role.
- `Gm_AddUser` and `Gm_RemoveUser`, which add or remove role members, i.e., change a *take* relationship.

- `Gm_AddHave` and `Gm_DelHave`, which add or remove a particular *have* relationship.
- `Dm_GetACL` and `Dm_SetACL`, which get or set the ACL of an object.
- `Dm_GetOwnerList` and `Dm_SetOwnerList`, which get or set the owner list of an object.
- `Dm_GrantRight` and `Dm_RemoveRight`, which grant an explicit right (either positive or negative) of an object to a subject, or remove an explicit right of an object from a subject.
- `Dm_CheckRight`, which registers a callback that checks the right a subject has over an object.

By providing the above primitives, we allow application programmers to tailor the access control model according to the access needs of individual applications. To illustrate, consider the software development application. Assume that we want to add a new right `CheckSpellingR` into the software development application. The right is checked by the spelling-check callback. The application can use `Dm_AddRight` to add the new right `CheckSpellingR` and `CheckSpellingR*`, use `Dm_AddInclude` to add the rule "`DataR` *includes* `CheckSpellingR`", use `Dm_AddImply` to add the rules "`WriteR` *implies* `CheckSpellingR`" and "`CheckSpellingR` *implies* `ReadR`", and use `Dm_CheckRight` to register the spelling-check callback that checks the right. The `CheckSpellingR` right then becomes an integral part of the right inheritance space. Therefore, when the right is checked at the beginning of the spelling-check callback, the system will automatically use the inheritance model to infer the `CheckSpellingR` right.

However, there are still situations when the access needs of applications cannot be met by the mechanism described above. To illustrate, let us add the following additional access requirements to the software development application: The code reviewing process cannot begin until the chairman and the author put their signatures

in the signature fields. Moreover, the author can sign only after the chairman has signed. This requirement cannot be represented by our model directly since the input parameters deciding the access rights contains temporal elements besides the subject, object and access right. As another example, consider situations in which controlling of access is related to the real time. Assume that users can access an object only during a certain time period, say working hours in weekdays. This requirement cannot be directly specified by the specification model we have described.

To support the above access needs, we use the notion of access callbacks. If the access requirement of a right R of an object O cannot be represented by the access model directly, the programmer can use the `Dm_SetCallback` primitive to define that an access callback is associated with the right R of the object. The callback, written in some conventional programming languages such as C, is a procedure that defines the access requirements of right R of object O. When the access right R for object O is checked, its associated callback is invoked and the return value is used as the access rights.

To illustrate, again consider the software development application discussed above. Assume that `Signature1` and `Signature2` are the objects on which the chairman and author put their signatures respectively, and the chairman is granted a positive `WriteR` right on `Signature1`. To represent the requirement that the author can sign only after the chairman signs, the programmer associates a callback `check_signature2` with the `WriteR` right of the `Signature2` field, which is called whenever the `WriteR` of the object is checked. The callback returns a positive right only if the `Signature1` object is signed and the caller takes the role of "author".

We have defined access callbacks as attributes of protected objects. Therefore, it is inherently embedded in our inheritance and coupling scheme. Consequently callbacks themselves can be inherited and shared (coupled). If corresponding objects of users are access coupled, their associated callbacks are also shared. In addition, a callback will be invoked whenever the associated value is needed by the inheritance algorithm. Consider the software development tool again. Although the callback is defined only

for the write right, it will be invoked whenever an *imply* relationship needs to access its value. As a result, the performance of the model will be affected. Moreover, if distributed workstations have to request the invocation of a callback remotely to get its value, the degradation of performance will be more obvious [7].

To solve the above performance problem, we use the following "cache and hint" approach. A cache is associated with every access callback. The `Dm_SetHint` and `Dm_ResetHint` primitives are provided to allow applications to specify whether a specific value associated with (Subj, Obj, Right) evaluated from the callback needs to be cached in the local workstation, and whether the cached value needs to be discarded. If the value is cached in the local workstation, the callback does not need to be invoked when the object is accessed in the corresponding mode. The `Dm_ResetHint` also allows an application to turn off the caching mechanism, in which case the cache is refreshed, and all the subsequently evaluated values are not stored in the cache. This is the default setting by the system, since by default the system does not know a priori when the cached value will be obsolete. To ensure correctness, the cache is not used by default.

[7] As will be shown in next chapter, in our implementation of the model, distributed workstations have to request the invocation of callbacks in a remote application object.

4. IMPLEMENTATION AND PRELIMINARY EXPERIENCE

We have described the semantics of a general access control model in the previous chapters. In this chapter, we discuss a particular implementation of the model in the Suite multi-user infrastructure and how we support the automation requirement. We also describe some experiments that measures the performance of this implementation, and some initial experience in using the model to support several diversified access policies.

4.1 Implementation

Realization of the Access Matrix

As described in chapter 2, there are traditionally two popular approaches to represent the access matrix: access control lists and capability lists. Both approaches have their advantages and disadvantages. From the perspective of search efficiency, if the number of protected objects is large, it will be inefficient to search a long capability list associated with a subject in the capability systems. Similarly, if the number of subjects is large, it will be costly to search a long access control list associated with an object in the access control list systems. From the perspective of access administration, it is often desirable to offer an authorizer a global view of those users that can access a specific object, and a global view of all the capabilities that a specific user has. In capability systems, it is easy to present all the capabilities that a specific user has, but it is expensive to construct a view of those users that can access a specific object, since the C-list of all users have to be searched to find those holding a capability to the object. Similarly, in access control list systems, it is straight-forward to present a view of those users that can access a specific object, but it is expensive to

construct a view of all the capabilities that a specific user has since doing so requires a search of all the access control lists associated with all the objects.

Because it is currently not clear to us as to which mechanism is the best for realizing access matrix for collaborative systems, we leave it to the implementator of our model to decide the implementation mechanisms. In our particular implementation of the access model in Suite, we use an extended access control list (XACL) as the underlying data structure to store the explicitly specified access rights. An XACL consists of a list of access blocks, which specify, for each right, the kind of access mode (positive or negative) a role has. Depending on the rights, an access block can take two different forms. For those rights that involve only one subject, such as the conventional read and write right, an access block is in the form of (R: ($\omega_1 S_1 \omega_2 S_2 ... \omega_i S_i$)), where R is a right defined over the object, ω_i is either + or -, denoting positive right or negative right respectively, and S_i is a role. For those rights that involve complex objects consisting of one object O and one passive subject, such as the coupling rights, the access block is in the form of (R: $S_a(\omega_1 S_1 \omega_2 S_2 ... \omega_i S_i)$), where $S_1, S_2, ..., S_i$ are passive subjects. It indicates the kind of R right (positive or negative as defined by ω_j) S_a has on the complex object (O, S_j).

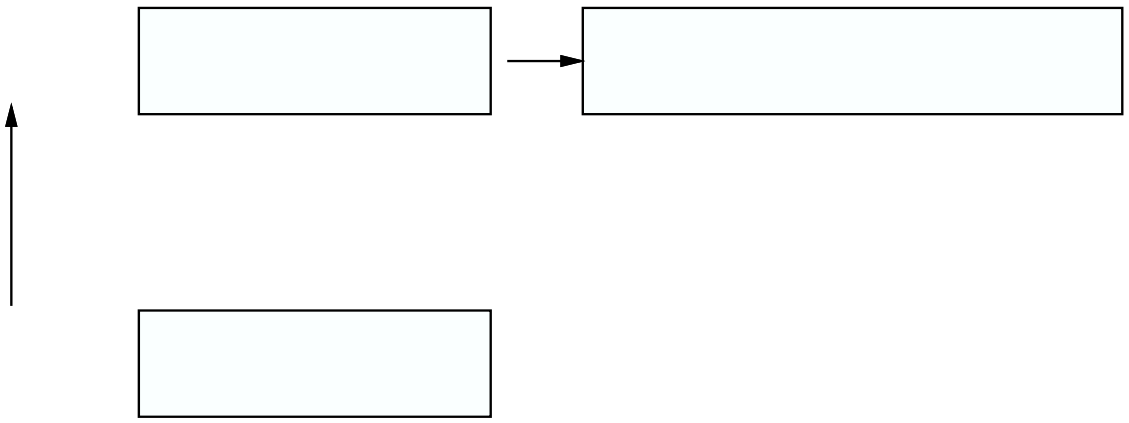

Figure 4.1 XACL of two objects.

Figure 4.1 illustrates the XACL of two protected objects in the software development tool: a comment line L1 and a function F1 containing L1. The XACL of L1 has only one access block specifying that user `hhs` can write the line. F1, the structure parent of L1, has an XACL consisting of two access blocks. The first access block specifies that the `serc` role has a positive read right and the `student` role has a negative read right over F1. The second access block of the XACL involves a coupling right. It specifies that the `student` role cannot couple with the `student` role but can couple with the `teacher` role.

In our implementation, we choose the object dimension as the prime dimension as described in section 3.6. That is, we view multi-dimensional inheritance within the framework of object dimension by attaching other dimensions to each node in the object dimension. To decide the value of (s,o,r), we first find o and check whether the information associated with o is enough to infer the value of (s,o,r), by using inheritance in subject and right dimensions if necessary. If not, we proceed by using inheritance in the object dimension. This is also in the spirit of the implementation of Unix. Unix uses the ACL approach and therefore chooses object dimension as the prime dimension. In addition, we choose the right dimension as the sub-dimension, and the subject dimension as the sub-sub-dimension. That is, we start the search first from object dimension, then from right dimension, and finally from subject dimension. The detailed search algorithm is presented in section 3.6.

We use the example of figure 4.1 to illustrate the search process. To check whether user `rxc` has the `ReadR` right to L1, the system first finds L1 and tries to decide the right from its associated XACL. Because there is no access block associated with the `ReadR` right, the system uses the access block of `WriteR` right because of the right implication rule "`WriteR` implies `ReadR`". It then uses subject dimension inheritances to infer whether user `rxc` can inherit the positive right from the role `hhs`. Since no definite right can be inferred, the system starts over from the structure parent of L1, namely F1, and tries to decide the right from XACL of F1. It accesses the access block of `ReadR` right first, and then uses subject inheritance rules to decide whether

`rxc` can inherit the positive `ReadR` right the `serc` role has. This time it succeeds because `rxc` takes the `serc` role and inherits all the rights of `serc`. Therefore, the algorithm returns a positive right.

Extensions to the Suite Framework and Automation

We have implemented the access control model by extending the Suite framework [38]. The original implementation of the framework consists of a dialogue compiler, which builds a symbol table from the type declarations of active variables; dialogue managers, each of which manages the interaction objects created for a particular user, including interaction variables, attributes, value groups, and object windows; a group manager, which manages all groups defined by users and programs; and a set of library routines used by applications to communicate with dialogue managers and group managers.

We have embedded the access control mechanism within the dialogue managers to control all editing commands entered by end-users. We have also built a session manager, which manages the activation of applications, connection and disconnection of dialogue managers to and from applications, and other session related activities discussed in chapter 3. In addition, we have built a new role manager based on the original group manager. It manages roles and notifies all the other objects, including dialogue managers and session managers, about changes to user roles. Finally, we have implemented the access control language interface described in chapter 3, which is used by application programmers.

Because all of the above mechanisms are provided by the system, application programmers are relieved from implementing the details of access control for each application and therefore automation is provided by the framework.

4.2 Performance

As discussed in chapter 1, one of the requirement we want to meet is efficient storage and evaluation. By using the inheritance mechanism, it is clear that the

storage of explicit rights can be reduced. However, the advantage is not gained for free. The tradeoff is the possible loss of time in evaluating the rights not explicitly specified. In this section, we discuss some experiments we conducted to analyze the performance of the model.

The purpose of the experiments was to determine how performance is degraded when inheritance is used to evaluate access rights. We compared the time spent when an access right is evaluated using the inheritance algorithm with the time spent when it is fetched directly from the access matrix.[1] The following factors affect the performance of the inheritance-based evaluation:

- the speed of the workstation that evaluates the access rights.

  The speed of the workstation decides how fast an access value can be evaluated. In our experiments, we used two popular modern workstations (Sun Sparc 4 and IBM RS6000) to conduct the experiments.

- the complexity of access requests.

  The complexity of an access request determines how many inheritance rules have to be used to evaluate the rights. In simple cases, single dimension inheritance may be sufficient to get the result, while in complex cases, three dimensional inheritance may have to be used, which would increase system overhead. In the experiments, we varied the complexity of the access request so that various inheritance schemes can be combined. We first specified a set of access definitions and access rules to the system. We then issued access requests that required no inheritance, i.e., when the access definition was directly accessible from the XACL of the object. We then added more complexity to the requests by requiring inheritance searches of more dimensions. We added simple inheritance in each of the three dimensions, then two dimensional inheritance in

[1] It is unfair to compare the two approaches in this way from a human factors perspective. Our approach reduces the number of specifications an end-user would have to enter interactively otherwise. In this perspective, the gain of specification time in our model is better than the latter approach. Nevertheless, we use the experiments to demonstrate that even if we compare the performance using the above criteria, the loss can still be kept low.

all combinations of two of the three dimensions, and finally three dimensional inheritance.

- the implementation of the access algorithm.

  The implementation of the model on a specific system has impact on the evaluation. For example, the performance of the model would have been better if we had used the hashed table or binary bit storage method instead of the linked list approach to store the access matrix. In the experiments, however, we only used the particular implementation in Suite to assess the model.

Table 4.1 gives the performance data when we varied the above factors to evaluate the model. The time is measured in milliseconds. To offset the network noise, we repeat each part of the experiments five times, and each time we executed a loop that repeated the computation 1000 times. The data given in the table is therefore the mean time of evaluating the rights 5000 times.

Table 4.1 The time needed to evaluate an access right. The time is measured in milliseconds.

	AIX/IBM-RS6000	SunOS/Sun-Sparc4
Without inheritance	0.6	1.0
simple right inheritance	0.6	1.0
simple object inheritance	0.7	1.4
simple subject inheritance	0.9	1.7
right/subject inheritance	1.5	1.8
subject/object inheritance	1.1	2.1
right/object inheritance	4.1	5.6
3 dimensional inheritance	5.1	6.3

From the table, we see that computing access definitions from multiple inheritances is not a CPU-bound operation. The increase of time in the worst case, in our experiments, is about 6 milliseconds. This performance penalty is not considered to be high [81, 82].[2]

4.3 Experience

Throughout this thesis, we have used the software development application to motivate and illustrate our design. In this section, we describe several additional examples to show how our access control model can be used to support the access policies of different classes of multi-user applications. Specifically, we will use a simple text editor, an outline editor, a two-person talk program, a simple mail program, and a multi-user command interpreter [38], and enhance the code of these applications using our access control primitives. We will also use a code inspection tool, a file protection program similar to Unix, an AFS simulation program, a programming contest tool, and the multi-level model to illustrate how our model can be used to support more complex application needs.

Figure 4.2 shows the code of a simple text editor allowing users to concurrently edit a text string. The first line of the program is a special C comment telling Suite the types of the active variables of the program. The `Load` procedure defines the callback invoked by a dialogue manager when the user starts a session with the application. It invokes `Dm_Submit` to create the interaction variable `Text` in the dialogue manager. The arguments of `Dm_Submit` specify the location, name, and type, respectively, of the associated active variable. It then uses `Dm_GrantRight` to specify the access policy of the application, which is simple and straight-forward: the string is treated as the protected object, and all users are granted every right of the string. This is supported by giving the role `all` a positive `AllR` right to the variable. Finally, the load callback invokes the `Dm_Engage` call to ask Suite to display `Text` in the object window.

[2]The studies show that for interactive systems, response time for intermediate steps in a process should not exceed 2 seconds (2000 ms). Furthermore, any delay between input action and output response under 8.3 ms is undetectable by human eyes.

```
/*dmc Editable String */
String text = "";
Load () {
    Dm_Submit(&text, "Text", "String");
    Dm_GrantRight("Value: Text", AttrAllR, "all", '+');
    Dm_Engage("Text");
}
```

Figure 4.2 A string editor.

Figure 4.3 shows a more complex example of a multi-user data-structure editor, namely an outline editor. Similar to GROVE described in section 2.3.8.1, it allows multiple users to edit outline items synchronously. The protected objects of the application are fine-grained outline items and their children. Each outline item is associated with an access control list. The owner of an item can change its access list, and decides whether the item is accessible to the owner only, to an enumerated set of users, or to all users. By default, the owner of an outline item has all its data rights (read, write, etc.), while others have only the read right. In addition, the access definition of any child of an outline item is by default inherited from its structure parent. Since most of the above access needs are supported in our model by default, we need to add only one access specification for the application by granting all users a positive `ReadR` right over the generic group.

Figure 4.4 shows the code of a two-person talk program. The program allows two users to talk with each other using different windows. In the program, the `Load` callback keeps a count of the number of times it was called and allows only two users to connect to the program. It creates separate text buffers for each user. In addition, it prevents one user from doing incorrect operations, such as typing, in the wrong window. It does so by granting the user all the rights to his own talk window, but only the read right over the other window.

```
typedef char *String;
typedef struct {unsigned num; struct section *sec_arr; } SubSection;
typedef struct section {
    String Name; String Contents; SubSection Subsections;
} Section;
typedef struct {unsigned num; Section *sec_arr; } Outline;
Outline outline;

Load ()
{
    /* All users can read any item by default */
    Dm_GrantRight("Type: Default", AttrReadR, "all", '+');
    Dm_Submit( &outline, "Outline", "Outline" );
    Dm_Engage("Outline");
}
```

Figure 4.3 An outline editor.

```
typedef char *String;
String UserA = "", UserB = "";
int talkers = 0;
char *uid;

Load ()
{
    if (talkers < 2) { /* only two users are allowed to talk */
        talkers++; /* keep count of the users */
        Dm_Submit(&UserA, "UserA", "String");
        Dm_Submit(&UserB, "UserB", "String");

        uid = strdup(Dm_GetUserId());
        if (talkers == 1) { /* user A connects */
            Dm_GrantRight("Value: UserB", AttrReadR, uid, '+');
            Dm_GrantRight("Value: UserA", AttrAllR, uid, '+'); }
        else { /* user B connects */
            Dm_GrantRight("Value: UserA", AttrReadR, uid, '+');
            Dm_GrantRight("Value: UserB", AttrAllR, uid, '+'); }

    Dm_Engage_Specific("UserA", "UserA", "Text");
    Dm_Engage_Specific("UserB", "UserB", "Text");
    }
}
```

Figure 4.4 Two-Person Talk.

Figure 4.5 shows portions of code of a simple mail program that are related to access specification. The program has two active variables that are used for outgoing and incoming messages, respectively. While users are allowed all the operations over the outgoing messages, they can only read the incoming messages. This policy is supported by granting users all the rights over the outgoing message, but only the read right over the incoming messages.

```
/* dmc Editable Msgs */
typedef struct {String From, To, Text; } Msg;
typedef struct { unsigned num; Msg *msg_arr; } Msgs;
Msg Outgoing; Msgs Incoming; struct passwd *pwd;

......

Load ()
{
    Dm_Submit(&Outgoing, "Outgoing", "Msg");
    Dm_Submit(&Incoming, "Incoming", "Msgs");

    ......

    Dm_GrantRight("Value: Outgoing", AttrAllR, Dm_GetUserId(), '+');
    Dm_GrantRight("Value: Incoming", AttrReadR, Dm_GetUserId(), '+');
    Dm_GrantRight("Value: Incoming", AttrDataR, Dm_GetUserId(), '-');
    Dm_Engage("Outgoing");
    Dm_Engage("Incoming");
}
```

Figure 4.5 Mail Program.

Figure 4.6 shows portions of code of a multi-user command interpreter that is related to access control. It allows users to enter new commands, and displays the output of these commands. Users cannot change the output of the commands, nor any

commands already executed. The application supports this access policy by granting users a negative `WriteR` right over the output type `CmdRec.Result`, and a positive `AllR` right over its parent `CmdRec`. Furthermore, after a new command is executed by the `Execute` procedure, it is given a negative `WriteR` right so that it is no longer editable.

Figure 4.7 shows portions of code in the software inspection tool that is related to access control. The access control policy of the application is the following:

1. The creator of an object becomes its owner and has the data rights (read, insert, delete, and modify) over the object, and the rights to further transfer these rights. Multiple ownership is supported so that a group of users can jointly-own portions of a program. This requirement is supported by defining a new implication rule: "ownership implies `DataR` and `DataR*`", and by using the default ownership rules of the model.

2. Only the chairman can change the name of the program to be inspected. Others can only read it. This is supported by giving the chairman the owner right of the program name, and others the `ReadR` right.

3. Only the formatter can change the color, font, or title of the data in the program. Other participants can do so only under the permission of the formatter. This is supported by giving the formatter role the positive `FormatR` and `FormatR*` rights.

4. By default, users can read the whole program, which is supported by giving the `ReadR` right to all participants over the generic group.

Figure 4.8 shows how Unix file protection policy is supported using our model. As described in section 2.3.2, in Unix, users are grouped into owner, group, and others, objects are grouped into files, and operations are grouped into read, write and execute. To simulate the Unix model, we create Unix files as textual active variables. In addition, we remove the default implication rule "write implies read", which is not supported in Unix. We define "OwnerR implies DataR*" to simulate its ownership semantics. We also set the default XACL for the file according to the umask value.

```
/* dmc Editable CmdSeq */
typedef struct {String Cmd, Result; } CmdRec;
typedef struct { unsigned num; CmdRec *msg_arr; } CmdSeq;
CmdRec Init = {"", ""}; CmdSeq CmdList = {1, &Init };

void Execute (cmd_name, val)
    char *cmd_name; String *val;
{   char cmd_path[1024];

    /* execute the command and display output to object windows */
    ......


    /* previous cmd cannot be changed */
    sprintf(cmd_path, "Value: %s", cmd_name);
    Dm_GrantRight(cmd_path, AttrWriteR, "all", '-');
    ......
}

Load ()
{
    Dm_Submit(&CmdList, "CmdList", "CmdSeq");
    ......
    Dm_GrantRight("Type: CmdRec.Result", AttrWriteR, "all", '-');
    Dm_GrantRight("Type: CmdRec", AttrAllR, "all", '+');
    Dm_Engage("CmdList");
}
```

Figure 4.6 Command Interpreter.

```
String program_name;

......

Dm_AddImply(AttrOwnerR, AttrDataR); /* OwnerR implies DataR */
Dm_AddImply(AttrOwnerR, AttrAttrDataR); /* OwnerR implies DataR* */

Dm_SetOwnerList("Value: program_name", "+chairman");
Dm_GrantRight("Value: program_name", ReadR, "all", '+');

Dm_GrantRight("Type: Default", AttrFormatR, "formatter", '+');
Dm_GrantRight("Type: Default", AttrAttrFormatR, "formatter", '+');

Dm_GrantRight("Type: Default", AttrReadR, "all", '+');
```

Figure 4.7 Access code of the Software Inspection Tool.

```
String f;
char subj[3];
int right[3] = {AttrReadR, AttrWriteR, AttrExecR};

Dm_DelImply(AttrOwnerR, AttrOlistR);  /* disable multiple ownership */
Dm_DelImply(AttrWriteR, AttrReadR);  /* disable ``write implies read'' */
Dm_AddImply(AttrOwnerR, AttrAttrDataR);  /* owner can change ACL */

/* assume that f is the active variable corresponding to file f */
subj[0] = Dm_GetOwnerList(f); /* get owner */
subj[1] = Gm_Getgroup(0); /* get group */
subj[2] = "all";

for (s = 0; s <= 2; s++) {
    for (r = 0; r <= 2; r++) {
/* set +/- right depending on umask value */
if (UmaskSet(subj[s], right[r]))
          Dm_GrantRight("Value: f", right[r], subj[s], '+');
else Dm_GrantRight("Value: f", right[r], subj[s], '-');
    }
}
```

Figure 4.8 Supporting Unix file protection.

Figure 4.9 shows how the AFS protection policy is supported using our model. In the simulation, we define a directory to be a structure consisting of a sequence of files and a sequence of directories. The `'a', 'l', 'i'` and `'d'` rights of AFS defined over a directory is equivalent to the `AllR*`, `ReadR`, `InsertR`, and `DeleteR` of our model. The 'r', 'w', and 'l' rights, though associated with a directory, actually apply only to the files within the directory. Therefore in our simulation, when users specify these rights, they should associate them with the file sequence of the directory. The 'r' right corresponds to the `ReadR` right of our model. The 'w' right is not equivalent to the `WriteR` right of our model because according to its semantics in AFS, it is used for both the `WriteR` and `AllR*` rights. Therefore, we define a new 'w' right and add a new *include* relationship: "`w` includes `WriteR` and `AllR*`". In addition, we disable the rule "`WriteR` implies `InsertR` and `DeleteR`" according to the AFS semantics. Moreover, when a new directory is created in AFS, its ACL is copied from its parent directory. This feature is implemented in the callback associated with the directory insert operation. Similarly, when a new file is created, its ACL is defined according to the protection bits decided by the umask value of the process. To support the AFS semantics, we grant the `all` role a negative right if the file protection bit is off and do nothing if the bit is on. In this way, if the file protection bit is off, then the access will be rejected because of the negative right in the ACL. On the other hand, if the file protection bit is on, then by inheritance, the access is determined by the rights associated with the directory. In addition, to support the AFS semantics of disallowing positive rights on a file, we associated a callback with files that is invoked before any update to the ACL of a file. If the ACL contains a positive right, then the update is disallowed.

Figure 4.10 shows how the software development tool can be changed to support the programming contest application. In the application, participants form several teams to compete each other. Assume that at the startup, the proctor creates, for each team Ti, a function Fi as the workspace for the team. The competition has two stages. In the first stage, each team is required to work out its solution to the contest

```
typedef char *String; typedef char *File;
typedef struct {unsigned num; File *arr; } FileList;
typedef struct {unsigned num; struct dir *arr; } SubDir;
typedef struct dir { String Name; FileList Files; SubDir Subd; } Directory;
Directory directory;
int right[3] = {AttrReadR, AttrWriteR, AttrExecR}, w, l;

void InsertDirCbk(parent, child, index) /* called when new dir is created */
    char *parent, *child; int index;
{   char vpath[1024], pvpath[1024], *acl; int r;
    sprintf(vpath, "Value: (%s.Subd)", child);
    sprintf(pvpath, "Value: %s", parent);
    /* copy ACL from parent directory */
    for (r = AttrFirstRight; r <= AttrLastRight; r++)
        if ((acl = Dm_GetACL(pvpath, r)) != 0) Dm_SetACL(vpath, r, acl);
}
void InsertFileCbk(parent, child, index) /* called when new file is created */
    char *parent, *child; int index;
{   char vpath[1024]; int r;
    sprintf(vpath, "Value: %s", child);
    for (r = 0; r <= 2; r++) /* set right depending on Unix umask value */
        if (!UmaskSet("owner", right[r]))
            Dm_GrantRight(vpath, right[r], "all", '-');
}
/* called before acl is updated, a non-zero return value will disallow update */
char *FileAclVerifyCbk(path, acl)
    char *path, *acl;
{   return (strchr(*acl, '+'));
}
Load ()
{   Dm_Submit( &directory, "Directory", "Directory" );
    w = Dm_AddRight("w");
    l = Dm_AddRight("l");
    Dm_AddInclude(w, AttrWriteR);
    Dm_AddInclude(w, AttrAttrAllR);
    Dm_DelImply(AttrWriteR, AttrInsertR);
    Dm_DelImply(AttrWriteR, AttrDeleteR);
    Dm_SetAttr ("Type: SubDir", AttrSequenceInsertUpdateProc, InsertDirCbk);
    Dm_SetAttr ("Type: File", AttrSequenceInsertUpdateProc, InsertFileCbk);
    Dm_SetAttr ("Type: File", AttrAclVerifyProc, FileAclVerifyCbk);
    Dm_Engage("Directory");
}
```

Figure 4.9 Supporting AFS protection.

problem without any collaboration outside of the team. In the second stage after all the teams have submitted their programs, all the participants collaborate together to inspect and evaluate the solution of each team.

Specifically, the following access policy needs to be supported for the application.

In the first stage:

1.1. Each team member can read, insert, delete, and modify the contents of their program. This is supported by granting a positive `DataR` right to the team.

1.2. Each team member can change the format of their program such as the font, color, etc. This is supported by granting a positive `FormatR` right to the team.

1.3. Members within the same team Ti can couple with each other and with the proctor, but may not couple with other team members. This is supported by giving Ti a positive coupling right with respect to Ti, but a negative coupling right with respect to others.

1.4. The proctor can read the programs of each team. In addition, the proctor is in charge of the access administration. This is supported by giving the proctor role a positive `ReadR` and `AllR*` right over the generic group, and by removing the `AllR` and `AllR*` right from the owners.

In the second stage, after all the teams have submitted their solutions:

2.1. All the participants can read the programs of other teams. This is supported by giving all users a positive `ReadR` right over the generic group.

2.2. All the participants can couple with each other, which is supported by giving all users a positive `CoupleR` right over the generic group.

2.3. All the participants can insert comments into others' program, which is supported by giving all users a positive `InsertR` right.

2.4. A participant who inserts a comment line becomes its owner and is responsible for specifying the access definitions of the comment line. This is supported by adding a new rule "Ownership implies AllR and AllR*".

Finally, we discuss how the mandatory access control policy of Bell-LaPadula Model can be supported in our model. As described in section 2.3.5, mandatory access

```
/* 1.1 */
Dm_GrantRight(F[i], AttrDataR, Team[i], '+');
/* 1.2 */
Dm_GrantRight(F[i], AttrFormatR, Team[i], '+');
/* 1.3 */
/* CoupleR: Team[i](+Team[i]+proctor-all) */
sprintf(acl, "%s(+%s+proctor-all)", Team[i], Team[i]);
Dm_SetACL(F[i], AttrCoupleR, acl);
/* 1.4 */
Dm_GrantRight("Type: Default", AttrReadR, proctor, '+');
Dm_GrantRight("Type: Default", AttrAttrAllR, proctor, '+');
Dm_DelImply(Ownership, AttrAllR);
Dm_DelImply(Ownership, AttrAttrAllR);
/* 2.1 */
Dm_GrantRight("Type: Generic", AttrReadR, "all", '+');
/* 2.2 */
Dm_SetACL(F[i], AttrCoupleR, "all(+all)");
/* 2.3 */
Dm_GrantRight("Type: Text", AttrInsertR, "all", '+');
/* 2.4 */
Dm_AddImply(Ownership, AttrDataR);
Dm_AddImply(Ownership, AttrAttrDataR);
```

Figure 4.10 Access specification for the programming competition application.

control decides accessibility of subjects to objects according to the security labels associated with subjects and objects. Recall from section 2.3.5 that the mandatory part of BLM has two components: 1) the no read-up property, which specifies that a subject can read only objects with equal or lower security levels; and 2) the no write down property, which specifies that a subject can append information only to objects with equal or higher security levels, and a subject can write only object with the same security level. To simulate these two components, we need to set up rules such that for the read, append, and write rights, the above two properties can be satisfied. The approach we take is to specify as few explicit rights as possible, and use the "take", "have", and "imply" relationships to derive the unspecified rights according to these two properties. To be more specific, we explicitly specify that a subject can write only objects with the same security level. By defining "imply" relationships between read, append, and write, the subject can also read and append the objects of the same level. In addition, we define a set of rules using the "have" relationship that specify that a subject inherits the read right only from the subjects with lower security levels (read-down-only) and the append right only from the subjects with higher security levels (append-down-only), thereby satisfying the two properties.

Formally, let $l_1, l_2, ..., l_n$ be the security levels as defined in BLM, with a smaller number indicating higher security ($l_1 > l_2 > ... > l_n$). Let $O_1, O_2, ..., O_n$ be sets of objects with O_i denoting the set of objects with security label being l_i. The simulation consists of the following steps:

1). For each security level l_i, we define a role l_i.

2). We explicitly grant l_i a positive write right to object set O_i (i = 1, 2, ..., n).

3). We define the following "imply" relationships: "WriteR implies AppendR" and "WriteR implies ReadR". Therefore, role l_i has the read and append rights to O_i.

4). We define the "have" relationship for i = 1, 2, ..., n-1: have(l_i, ReadR, $l_{(i+1)}$) = true, which specifies that role l_i inherits the positive read right of role $l_{(i+1)}$. In this way, we allow subjects to read objects with lower security labels.

5). We define the "have" relationship for i = 1, 2, ..., n-1: have($l_{(i+1)}$, AppendR, l_i) = true, which specifies that role $l_{(i+1)}$ inherits the positive append right of role l_i. In this way, we allow subjects to append information to objects with higher security labels.

6). To give user U_j a security level l_i, we define the following "take" relationship: "U_j takes role l_i".

From the above specification, users taking the l_i role can inherit the read right only from $l_{(i+1)}$, ..., l_n. Therefore, the "no read up" property of BLM is satisfied. Similarly, users taking the l_i role can inherit the append right only from $l_{(i-1)}$, ..., l_1. Therefore, the "no append down" property is satisfied. As for the write right, the only way a user taking the l_i role can get it is through the *take* relationship on l_i. Therefore the simple property and *-property of BLM can both be satisfied.

The code to simulate the BLM mandatory access policy is shown in figure 4.11.

Table 4.2 gives the number of lines of code that are related to access control in each of these applications. These numbers measure how much effort an application programmer has to put in specifying the access needs of an application. As indicated by the table, the high-level mechanisms in our model make it easy for programmers to specify access control.

```
/* step 1 and 2 */

for (i = 1; i <= n; i++) {
    Gm_MakeGroup(l[i]);
    Dm_GrantRight(O[i], AttrWriteR, l[i], '+');
}

/* step 3 */

/* Dm_AddImply(AttrWriteR, AttrAppendR) is a system default rule */
/* Dm_AddImply(AttrWriteR, AttrReadR) is a system default rule */

/* step 4 and 5 */

for (i = 1; i < n; i++) {
    Dm_AddHave(l[i], AttrReadR, l[i+1]);  /* read down only */
    Dm_AddHave(l[i+1], AttrAppendR, l[i]); /* append up only */
}

/* step 6 */

/* to give user U[j] a security level of level(U[j]) */
for (j =1; j <= MAX_USERS; j++)
    Gm_AddUser(U[j], level(U[j]));
```

Figure 4.11 Simulating mandatory access control policy.

Table 4.2 Lines of access control code (comments excluded).

	# of access code
String editor	1
Outline editor	1
Two-person talk	7
Mail	3
Command interpreter	4
Software inspection tool	7
Unix	15
AFS	30
Programming contest	13
Mandatory control	10

5. COMPARISON WITH OTHER WORK

In this chapter, we compare our work with previous work. We discuss the similarities between our model and other models, point out the features our model supports that others do not, and wherever possible, show how other models can be easily simulated using our model.

5.1 Matrix Model

Since our model is an extension of the classical matrix model described in section 2.3.1, it shares much with the matrix model. Like the classical matrix model, our model is a general, application-independent abstract model. Both models support flexible specification of access rights and use the notion of *-rights and ownership to support flexible access administration. In its most primitive form with inheritance rules and negative rights disabled, our model is the matrix model. Therefore, it can be as flexible as the matrix model and support arbitrary access control policies.

The matrix model, however, does not address high-level specification and automation. Our work extends the basic matrix model by using exception based, multiple inheritance mechanisms to ease the specification task. In addition, our model defines and supports a generic set of collaboration rights besides the traditional read, delete, insert, write, and execute rights of the matrix model, and supports deep revocation and automation. We also allow dynamic change of ownership semantics depending on application needs and support the notion of joint-ownership.

5.2 Bell-LaPadula Model

BLM addresses the issue of flexibility by containing a basic access matrix in its discretionary access control component. Like BLM, our model can simulate the basic matrix model as discussed in the previous section. BLM also supports easy specification of mandatory access policy by assigning ordered security labels to subjects and objects, and determining access rights through the labels. This multi-layered model is a special case of our model and can be simulated using the inheritance paradigm of our model, as shown in section 4.3.

A major difference between our work and BLM is that we have addressed issues related to the collaborative domain. We support a set of generic collaboration rights, role-based specification and a flexible administration scheme to meet the requirements we have identified, for which there are no analogues in BLM.

5.3 Unix

As discussed in chapter 2, Unix supports easy specification by making some simplifying assumptions about the protection domain. Like Unix, our model also makes use of the knowledge of the underlying protection domain to simplify the specification task. However, unlike Unix, our model does not sacrifice flexibility for easy specification. By using exception-based inheritance mechanisms, our model achieves both flexibility and easy specification at the same time.

Compared with the Unix protection model which supports only the read, write and execute rights, our model defines and supports an additional set of collaboration rights. Moreover, unlike Unix, our model supports access control based on multiple, dynamic roles. In addition, Unix uses the single ownership approach to address the authorizer problem, while our model supports the multiple ownership paradigm and allows flexible semantic definitions of the ownership, thereby providing more flexibility.

As discussed in section 4.3, the Unix model can be easily simulated by our model.

5.4 AFS

Like Unix, AFS trades flexibility for easy specification. Unlike our model that supports fine-grained object protection, it does not allow protection of objects at a finer-grained level than directories and files. A similarity between AFS and our model is that both use inheritance to support high-level specification. Nevertheless, they have several important differences. AFS supports inheritance only in the object and subject dimension, while our work also supports right dimensional inheritance to further ease the specification task. In the object dimension, though AFS allows a file to inherit the access definitions from its parent directory, it does not allow a file to have its own fine-grained access control list. Specifically, the access definitions associated with individual files apply to all users and cannot be tailored for specific users. When a directory is created, it copies its ACL from its parent directory instead of inheriting it dynamically. Therefore the inheritance mechanism of AFS in the object dimension is not as flexible as our model. In the subject dimension, AFS does not allow groups to contain subgroups, and makes a user's rights the union of the rights that his groups have. Compared with our role-based schema, it is less-flexible because our model allows groups to be defined over existing groups and users, and allows inheritance of selected positive rights through the *have* relationship, and inheritance of both positive and negative rights through the *take* relationship. In addition, AFS addresses only the traditional rights such as the read, write, insert, and delete, while our work also includes a set of collaboration rights. Finally, our model supports flexible access administration, which AFS does not address.

As discussed in section 4.3, the AFS model can be easily simulated by our model.

5.5 Hydra

As discussed in chapter 2, the Hydra model emphasizes on flexibility. It protects objects of arbitrary type, and allows users to define their own access rights. Like Hydra, our model also allows programmer-defined objects and types. On the other

hand, because we have designed our model specifically for the collaborative domain, we are able to use the underlying domain knowledge to facilitate the specification process. In particular, we predefine a set of collaboration rights and a set of relationships among subjects, objects and rights that are useful in collaborative domains, and use them to provide a higher-level specification model.

Unlike our model, Hydra does not have the notion of negative authorization and therefore does not support exception based specification. Moreover, in order to construct a universal access control model, Hydra does not support the notion of ownership. Instead, it leaves it to individual applications to decide and implement their own authorization scheme. In contrast, we support the notion of multiple ownership and allow flexible definition of the semantics of ownership.

Hydra supports automation. However, because it does not have any domain knowledge to use, the facilities it provides are low-level. Since our model is based upon a high-level collaboration model, it is able to provide high-level automation.

To simulate the Hydra approach, we disable the support for negative rights and all the default rules. In particular, we do not define any imply relationship on ownership so that ownership does not have any semantics.

5.6 Database Systems

5.6.1 System R

Like System R, our model supports both flexible and high-level specification by using the underlying semantics of protected objects. Our three-dimensional inheritance scheme, however, is more powerful than the single relational inheritance paradigm supported by System R. Furthermore, unlike System R, our model supports exception to ease the specification task. On the other hand, System R provides a powerful query mechanism for users to define the protected objects, which allows, for instance, content-specific objects to be identified and protected. This feature is not supported in our model.

Like System R, our model supports fine-grained access administration by using *-rights. In addition, our model supports flexible access administration by allowing flexible definition of ownership semantics and joint-ownership. We also support a set of collaboration rights in addition to the the read, insert, delete and write rights supported by System R.

To provide fine-grained protection similar to System R, we can map a relational structure to a record type in our model, a relation to a sequence variable, and a relational tuple to a member of the sequence. Consequently, we can apply the protection model we defined on the active variables to relations and provide both flexible and high-level protection.

5.6.2 OODB

Like our model, the OODB approach proposed by Rabitti et al [95] addresses the problem of how to provide both flexible and high-level specification. Both work use the notion of implicit rights to ease the specification task and use negative rights to support exception. Although both models allow inheritance in all the three dimensions of access control, they have several important differences. Rabitti's work only allows object dimension inheritance in the class hierarchy, subject dimension inheritance in the role hierarchy, and right inheritance in the imply hierarchy. More specifically, in the subject dimension, Rabitti's model specifies that users be granted a right if any of the roles they take have the positive right, and thereby supporting the inheritance of only positive rights. Our model, on the other hand, allows inheritance of *selected* positive rights through the *have* relationship, and inheritance of both positive and negative rights through the *take* relationship in the subject dimension. In this way, the "least privilege" principle is obeyed more faithfully. In the right dimension, Rabitti's model does not allow the grouping of access rights through the "include" relationship, which our model supports. Moreover, our work has taken the approach of allowing certain conflicts to exist, and has devised a set of rules to resolve these conflicts. Rabitti's work, on the other hand, partitions access rights into two

categories: a strong right set, and a weak right set. Conflicts of access rights within the same set are strictly disallowed. As mentioned in chapter 3, this approach may not apply to an environment where the set of access objects is very large and dynamic, and a user may take dynamic, multiple roles. Whenever a new role is assigned, a *take* or *have* relationship is changed, or there is a change of explicit access rights of a role to an object, potential conflicts could arise. Checking all these conflicts is costly and sometimes even unnecessary when a particular user may need to access only a small portion of a large set of objects. Moreover, as discussed in chapter 3, devising rules to resolve the conflicts instead of detecting and disallowing all the conflicts allows flexibility of specification. Thus, it is not only too rigid, but also computationally expensive to adopt the approach of simply disallowing conflicts [1]. In addition, Rabitti's model addresses only the traditional rights such as the read, write and append rights, while our work also includes a set of collaboration rights. Finally, our model presents a solution to support flexible access administration, which Rabitti's model does not address.

5.7 Other Collaborative Systems

As discussed in chapter 2, several existing collaborative systems also address the issue of access control. Among them are the GROVE outline editor, the DCS distributed conference system, the UCSD multimedia conference system, and the UNC Jointly-Owned Object System.

GROVE

Like the GROVE outline editor, our model also supports fine-grained protected objects. Moreover, both works provide high-level specification by exploiting the underlying domain knowledge and both use inheritance as the major mechanism for achieving the above goal. Unlike GROVE, however, our model supports rights other

[1] In fact, Rabitti's work does not give solutions to the detection of all the conflicts in their model.

than the read and write rights supported by GROVE. We also support multiple, dynamic roles to allow dynamic change of user rights. GROVE addresses protection over application-specific objects, namely outlines, while our model addresses protection over general, application-independent objects. GROVE uses only structural inheritance in the object dimension, while our model uses multiple inheritance in all three dimensions of access control to ease specification. Finally, unlike GROVE, we support exception-based specification, and flexible access delegation and revocation.

We can simulate GROVE by treating outline items as active variables and disabling all the inheritance rules of our model except structure inheritances. In this way, we can provide a protection model similar to GROVE, as was shown in the previous chapter.

DCS

A major difference between our model and DCS is that our model addresses the issue of protection over fine-grained protected objects, while DCS limits its application domain to coarse grained objects such as files and conferences. In addition, DCS supports only a set of predefined roles and rights specific to conferences, while our model allows dynamic roles, a set of application-independent collaboration rights, and programmer-defined rights. Finally, unlike DCS, our model also addresses the issue of flexible access administration.

UCSD Multimedia Conference System

Like our system, UCSD multimedia conference system also supports the rights to exchange information among users. In addition, it includes temporal logic into its specification model so that time-related requirements can be specified. In this respect, it provides a higher-level mechanism for specifying time-related access definitions than our model because in our model, temporal dependencies have to be programmed into callbacks using conventional programming languages rather than using system-provided high-level temporal primitives. The shortcoming of the UCSD approach is that the representation power of the model is limited by the temporal logic the model

uses. There are situations when it is impossible to represent the access needs using temporal logic. For instance, if the access right of a particular object depends on the existence of a signature file, then the model will not be able to meet this requirement directly. Our model, on the other hand, provides extensibility through callbacks and is able to handle the above access needs.

In addition, our model addresses several important issues that the UCSD system ignores. We address the the issue of how to provide high-level specification if the number of protected objects is large by providing an exception-based, multiple inheritance specification model. We also address the issue of flexible access administration by supporting joint ownership and flexible ownership semantics.

UNC Jointly-Owned Object System

Like the work of Guan et al [63] described in section 2.3.8.4, our work also supports joint ownership of protected objects by a group of users. Both models are application-independent and both address the issue of providing support at system level so that automation can be achieved. Compared to their work, our work does not directly associate quorum-based or authority-based conditions with objects. Instead, users have to program these conditions manually into corresponding callbacks in our model. On the other hand, Guan et al do not address ownership and other access specification when the number of objects is large. As a result, they do not support inheritance of access specification. Finally, unlike our model, their work does not allow flexible definition of ownership semantics and does not address the issue of collaboration rights.

5.8 Summary of Comparison

The following table summarizes the comparison of our work with other work. In the table, a blank entry stands for "not meeting the requirement" and N/A stands for "not available".

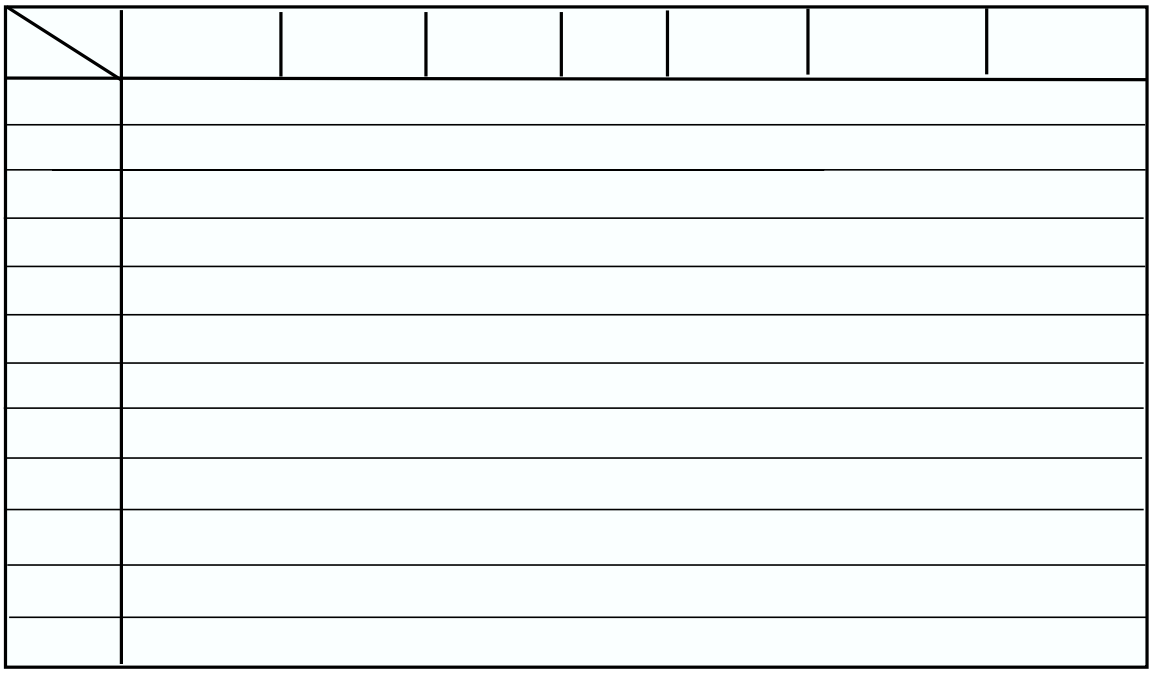

Figure 5.1 Summary of comparison.

6. CONCLUSIONS AND FUTURE WORK

Access control is important for collaborative systems. Because of its complexity, however, it is missing in many current systems. It has been observed in [44] that "Groupware's requirements can lead to complex access models, a complexity that must be managed ..., there must be lightweight access control mechanisms that allow end-users to easily specify changes. User interfaces should smoothly mesh the access model with the user's conceptual model of the system. Changing an object's access permissions should, for example, be as easy as dragging the object from one container to another".

Based on the above observation and other general requirements for conventional access control, we identified several new requirements in the context of collaborative environments. Some of the important issues we addressed to meet these requirements include: What are the basic protection elements for the environments? How can we provide a flexible and high-level specification model for collaborative access control? How can we support flexible access administration? And how can we support automation? As shown in this dissertation, existing work does not address all these problems completely.

We have addressed these issues by providing a new framework for designing and implementing collaborative access control. The main components of the framework are:

- a generic set of collaboration rights;
- multi-grained specification of access rights;
- inheritance in the subject dimension based on the *have* and *take* relationships;
- inheritance in the right dimension based on the *include* and *imply* relationships;

- inheritance in the object dimension based on both type and structural information;
- methods of inheriting definitions along multiple dimensions and resolving conflicts among definitions;
- flexible ownership semantics through the *imply* relationship;
- dynamic, multiple ownerships and inheritance of ownerships based on type and structural information.
- flexible delegation and revocation mechanisms based on indirect roles;
- a set of language constructs that allow application programmers to develop their own protection subsystems;
- generic support for implementing collaborative access control.

We believe that any flexible model for collaborative systems will be complex, as is our model. Therefore we have provided several methods for using and learning it incrementally. In particular, we have provided multiple dimensions of inheritance, which can be learned and used independently as illustrated by our examples.

Our inheritance rules and conflict resolution rules are defined by considering various scenarios in which they might be used. These rules are by no means the only suitable rules for various applications, since the access needs vary widely depending on applications and it is difficult to devise rules for these cases. By studying more applications, more general rules can be devised to enhance the model.

In this dissertation, several collaborative applications, including software development application and classroom examination application have been used to motivate and illustrate our design. Further applications need to be developed to test the usability of the model.

Distributed software architectures for access control is another area that needs further research. Because collaborators are distributed, there can be at least two

ways to realize the access model: centralized and distributed, in which each access to the protected objects is checked centrally or distributedly. Each approach has its advantages and disadvantages. The issue is related to the overall system architecture for collaboration, which can be centralized, distributed, or hybrid [35, 78]. Further research is needed to find out which access control architecture works better for various situations.

Access verification [64, 110, 80, 17] is another area for extending our work. It would be useful if the security of our model were proved formally.

As stated in chapter 2, access control addresses only one aspect of a secure system. It would be useful to combine our work with advances in other related areas, including cryptographic control, information flow control, inference control, user authentication, secure communication channel establishment, and concurrency control. It is not until then that we can provide a general solution for a secure collaborative computing environment.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Martin Abadi, Michael Burrows, and Butler Lampson. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] H. Abdel-Wahab, Sheng-Uei Guan, and J. Nievergelt. Shared workspaces for group collaboration: An experiment using Internet and UNIX interprocess communications. *IEEE Communications Magazine, Communications for Distributed Applications and Systems*, 26(11):10–16, November 1988.

[3] H.M. Abdel-Wahab and M.A. Feit. XTV: A framework for sharing X window clients in remote synchronous collaboration. In *Proc. IEEE Conf. on Communications Software: Communications for Distributed Applications and Systems*, pages 159–167, April 1991.

[4] A.F. Ackerman, F.H. Bushwald, and F.H. Lewski. Software inspections: An effective verification process. *IEEE Software*, pages 31–36, May 1989.

[5] S.R. Ahuja, J.R. Ensor, D.N. Horn, and S.E. Lucco. The Rapport multimedia conferencing system: A software overview. In *Proc. of 2nd IEEE conf. on Computer Workstations*, pages 52–58, March 1988.

[6] S.R. Ahuja, J.R. Ensor, and S.E. Lucco. A comparison of application sharing mechanisms in real-time desktop conferencing systems. In *Proc. of IEEE conf. on Computer Office Information Systems*, pages 238–248, April 1990.

[7] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[8] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–318, 1992.

[9] D.E. Bell and L.J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp., Bedford, Mass., March 1976.

[10] P. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] T.A. Berson and G. L. Barksdale. KSOS–development methodology for a secure operating system. *Proc. NCC AFIPS Press*, 48:365–371, 1979.

[12] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Access controls in object object-oriented database systems – some approaches and issues. *Lecture Notes in Computer Science, LNCS 759, Springer Verlag, edited by Nabil R. Adam and Bharat K. Bhargava.*, pages 17–44, 1993.

[13] M. Bishop and L Snyder. The transfer of information and authority in a protection system. *ACM Operating Systems Review*, pages 45–54, December 1979.

[14] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: Groupware for code inspection. In *Proc. of ACM Conf. on CSCW*, pages 169–181, October 1990.

[15] J. Cash, W.D. Haseman, and A.B. Whinston. Security for the GPLAN system. *Information Systems*, 2:41–48, 1976.

[16] IBM T.J. Watson Research Center. *AFS Fast Facts.* IBM, April 1993.

[17] M.H. Cheheyl, M Gasser, G.A. Huff, and J.K. Millen. Verifying security. *ACM Computing Surveys*, 13:279–339, 1981.

[18] Rajiv Choudhary and Prasun Dewan. Multi-user undo/redo. Technical Report SERC-TR-125-P, Software Engineering Research Center, Purdue University, August 1992.

[19] D. E. Comer. *Internetworking with TCP/IP, Volume I - Principles, Protocols, and Architecture.* 2nd edition, Printice-Hall, Englewood Cliffs, NJ, 1991.

[20] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume II - Design, Implementation, and Internals.* Printice-Hall, Englewood Cliffs, NJ, 1991.

[21] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume III - Client-Server Programming and Applications: BSD socket version.* Printice-Hall, Englewood Cliffs, NJ, 1993.

[22] R.W. Conway, W.L. Maxwell, and H.L. Morgan. On the implementation of security measures in information systems. *Communications of the ACM*, 15:211–220, 1972.

[23] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. MMConf: An infrastructure for building shared multimedia applications. In *Proc. of ACM Conf. on CSCW*, pages 329–341, October 1990.

[24] T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams.* Dorset House Publishing Co., New York, 1987.

[25] R.A. DeMillo. Combinatorial inference. In *Foundations of secure computation*, pages 27–35. ed. R.A. DeMillo et al., Academic Press, New York, 1978.

[26] R.A. DeMillo and D. Dobkin. The foundations of secure computation. In *Foundations of Secure Computation*, pages 1–12. ed. R.A. DeMillo et al., Academic Press, New York, 1978.

[27] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[28] D.E. Denning. A review of research on statistical data base security. In *Foundations of secure computation*, pages 15–25. ed. R.A. DeMillo et al., Academic Press, New York, 1978.

[29] D.E. Denning. *Cryptography and Data Security.* Addison-Wesley Publishing company, 1982.

[30] D.E. Denning and P.J. Denning. Data security. *ACM Computing Surveys*, 11(3):227–249, September 1979.

[31] Prasun Dewan. A tour of the Suite user interface software. In *Proc. of the 3rd ACM SIGGRAPH Symp. on User Interface Software and Technology*, pages 57–65, October 1990.

[32] Prasun Dewan. An inheritance model for supporting flexible displays of data structures. *Software – Practice and Experience*, 21(7):719–738, July 1991.

[33] Prasun Dewan. Coupling the user interfaces of a multi-user program. In *ACM SIGGRAPH Video Review, also in ACM CSCW'92*, November 1992.

[34] Prasun Dewan. Designing and implementing multi-user applications: A case study. *Software – Practice and Experience*, 22(12), December 1992.

[35] Prasun Dewan. Tools for implementing multi-user applications. *Trands in Software: Issue on User Interface Software*, 1:149–172, 1993.

[36] Prasun Dewan and Rajiv Choudhary. Experience with the Suite distributed object model. In *Proc. of IEEE Workshop on Experimental Distributed Systems*, pages 57–63. ACM, New York, 1990.

[37] Prasun Dewan and Rajiv Choudhary. Flexible user interface coupling in collaborative systems. In *Proc. of ACM CHI'91 Conf.*, pages 41–49, April 1991.

[38] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems*, 10(4), October 1992.

[39] Prasun Dewan and Rajiv Choudhary. Coupling the user interfaces of a multiuser program. In *ACM Transactions on Information Systems, to appear*, 1994.

[40] Prasun Dewan and John Riedl. Toward computer-supported concurrent software engineering. *IEEE Computer*, 26(1):17–27, January 1993.

[41] Prasun Dewan and Marvin Solomon. An approach to support automatic generation of user interfaces. *ACM Transactions on Programming Languages and Systems*, 12(4):566–609, October 1990.

[42] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 399–407. ACM, New York, May 1989.

[43] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Design and use of a group editor. In *Proc. of IFIP WG 2.7 Working Conf. on Engineering for Human-Computer Interaction*, pages 13–28, 1989.

[44] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[45] M. M. Astrahan et al. System R: relational approach to database management. *ACM TODS*, 1(2):97–137, June 1976.

[46] J. D. Eveland and T. K. Bikson. Work group structures and computer support: A field experiment. *ACM Transactions on Information Systems*, 6(4):354–379, October 1988.

[47] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[48] R. Fagin. On an authorization mechanism. *ACM TODS*, pages 310–320, December 1978.

[49] E. B. Fernandez, R. C. Summers, and T. Lang. Definition and evaluation of access rules in data management systems. In *Proc. of 1st International Conf. on VLDB*, pages 268–285, 1975.

[50] E. B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity*. Addison-Wesley, 1981.

[51] R.A. Finkel. *An Operating Systems Vade Mecum*. Prentice Hall, 1988.

[52] Ellen Francik, Susan Ehrlich Rudman, Donna Cooper, and Stephen Levine. Putting innovation to work: Adoption strategies for multimedia communication systems. *Communications of the ACM*, 34(12):53–63, December 1991.

[53] C. W. Fraser. A generalized text editor. *Communications of the ACM*, 23(3):154–158, March 1980.

[54] R.S. Gaines and N. Z. Shapiro. Some security principles and their application to computer security. In *Foundations of secure computation*, pages 223–236. ed. R.A. DeMillo et al., Academic Press, New York, 1978.

[55] Simson Garfinkel and Gene Spafford. *Practical UNIX Security.* O'Reilly & Associates, Inc., 1991.

[56] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. of 12th National Computer Security Conference*, pages 305–319, October 1989.

[57] G.S. Graham and P.J. Denning. Protection – principles and practice. *Proc. Spring Jt. Computer Conf.*, 40:417–429, 1972.

[58] R. Graubart. On the need for a third form of access control. In *Proc. of 12th National Computer Security Conference*, pages 296–304, October 1989.

[59] Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Information Systems*, 5(2):187–211, April 1987.

[60] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM TODS*, 1(3):242–255, September 1976.

[61] Jonathan Grudin. CSCW: An introduction. *Communications of the ACM*, 34(12):31–34, December 1991.

[62] Jonathan Grudin. Groupware: Eight challenges for developers. *Communications of the ACM*, 37(1):93–105, January 1994.

[63] Sheng-Uei Guan, Hussein Abdel-Wahab, and Peter Calingaert. Jointly-owned objects for collaboration: Operating-system support and protection model. *Journal of Systems Software*, 16:85–95, 1991.

[64] M. A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *CACM*, 19:461–471, 1976.

[65] A.R. Hurson, Simin H. Pakzad, and Jia bing Cheng. Object-oriented database management systems: Evolution and performance issues. *IEEE Computer*, pages 48–60, February 1993.

[66] Hiroshi Ishii and Naomi Miyake. Toward an open shared workspace: Computer and video fusion approach of teamworkstation. *Communications of the ACM*, 34(12):37–50, December 1991.

[67] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 4(6):58–67, November 1987.

[68] Won Kim and F.H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading, Mass., 1989.

[69] Michael J. Knister and Atul Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proc. of ACM Conf. on CSCW*, pages 343–356, October 1990.

[70] Henry Korth and Gregory D. Speegle. Long duration transactions in software design projects. In *Proc. of the 6th International Conf. on Data Engineering*, pages 568–574, February 1990.

[71] B.W. Lampson. Protection. *ACM Oper. Syst. Rev.,*, 8(1):18–24, 1974.

[72] B.W. Lampson, M. Abadi, M Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *DEC SRC Research Report*, February 1992.

[73] Thomas G. Lane. *User Interface Software Structures*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-101.

[74] Keith A. Lantz. An experiment in integrated multimedia conferencing. *Computer-Supported Cooperative Work: A Book of Readings*, pages 533–552, 1988.

[75] Keith A. Lantz and William I. Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, 3(1):23–51, January 1984.

[76] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer. An empirical study of distributed application performance. *IEEE Transactions of Software Engineering*, 11(10):1162–1174, October 1985.

[77] J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proc. of ACM CHI'90*, pages 303–312, April 1990.

[78] J.C. Lauwers and K.A. Lantz. Software architectures for collaborative systems. In *Unpublished manuscript*, 1991.

[79] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1992.

[80] R.J. Lipton and T.A. Budd. On classes of protection systems. In *Foundations of Secure Computation*, pages 281–296. ed. R.A. DeMillo et al., Academic Press, New York, 1978.

[81] I. Scott MacKenzie and Colin Ware. Lag as a determinant of human performance in interactive systems. In *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, pages 488–493, April 1993.

[82] Deborah J. Mayhew. *Principles and Guidelines in Software User Interface Design*. PTR Prentice Hall, 1992.

[83] Brad A. Myers. Creating user interfaces using programming by example, visual programming and constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, April 1990.

[84] R.M. Needham and R.D.H. Walker. The cambridge CAP computer and its protection system. *ACM Operating System Review*, 11(5):1–10, 1977.

[85] Christine M. Neuwirth, David S. Kaufer, Ravinder Chandok, and James H. Morris. Issues in the design of computer support for co-authoring and commenting. In *Proc. of ACM Conf. on CSCW*, pages 183–195, October 1990.

[86] R.E Newman-Wolfe and et al. A brief overview of the DCS distributed conferencing system. In *Summer Usenix Conference*, 1991.

[87] United States Department of Defense. Department of defense trusted computer system evaluation criteria. Technical Report DOD 5200.28-STD, Department of Defense, United States, December 1985.

[88] M. Tamer Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[89] John F. Patterson. Comparing the programming demands of single-user and multi-user applications. In *Proc. of the 4th ACM SIGRAPH Conf. on User Interface Software and Technology*, pages 79–86, November 1991.

[90] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and W. Scott Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *Proc. of ACM Conf. on CSCW*, pages 317–328, October 1990.

[91] P. A. Pittelli. The Bell-LaPadula computer security model represented as a special case of the harrison-ruzzo-ullman model. *Proc. of 10th national computer security conference*, pages 118–121, September 1987.

[92] Larry Press. Personal computing: Collective dynabases. *Communications of the ACM*, 35(6):26–32, June 1992.

[93] IBM Technical Publications. *Introduction to System and Network Security: Considerations, Options, and Techniques.* International Technical Support Centers, IBM Corporation, GG24-3451-01, January 1990.

[94] IBM Technical Publications. *Security Overview of Open Systems Networking.* International Technical Support Centers, IBM Corporation, GG24-3815-00, December 1992.

[95] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM TODS*, 1(16):88–131, March 1991.

[96] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11(3):276–285, March 1985.

[97] D. M. Ritchie and K Thompson. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[98] R.J.Brachman. On the epistemological status of semantic networks. *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50, 1979.

[99] T. Rodden, J.A. Mariani, and G. Blair. Supporting cooperative applications. *Computer Supported Cooperative Work, An International Journal*, 1(1):41–67, 1992.

[100] Mark Roseman and Saul Greenberg. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proc. of ACM Conf. on CSCW*, pages 43–50, November 1992.

[101] J.H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[102] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, pages 9–19, November 1993.

[103] Sunil Sarin and Irene Greif. Computer-based real-time conferencing systems. *Computer-Supported Cooperative Work: A Book of Readings*, pages 397–420, 1988.

[104] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[105] K Schmidt and L Bannon. Taking CSCW seriously: Supporting articulation work. *Computer Supported Cooperative Work, An International Journal*, 1(1):7–40, 1992.

[106] J.A. Scigliano and et al. A real-time Unix-based electronic classroom. In *Proc. of the IEEE Southeastcon '87*, April 1987.

[107] HongHai Shen and Prasun Dewan. Access control for collaborative systems. In *Proc. of ACM Conf. on CSCW*, pages 51–58, November 1992.

[108] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.

[109] John B. Smith and F. Donelson Smith. ABC: A hypermedia system for artifact-based collaboration. Technical Report TR91-021, Department of Computer Science, The University of North Carolina at Chapel Hill, April 1991.

[110] L Snyder. On the synthesis and analysis of protection systems. *ACM Operating Systems Review*, pages 141–150, November 1977.

[111] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Information Systems*, 5(2):147–167, April 1987.

[112] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[113] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Winter Usenix Conference*, pages 191–202, 1988.

[114] W. R. Stevens. *UNIX Network Programming*. Printice-Hall, Englewood Cliffs, NJ, 1990.

[115] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[116] H.M. Vin, P.V. Rangan, and M.S. Chen. System support for computer mediated multimedia. In *Proc. of ACM Conf. on CSCW*, 1992.

[117] William E. Weihl. Linguistic support for atomic data types. *ACM Transactions on Programming Languages and Systems*, 12(2):178–202, April 1990.

[118] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, July 1974.

APPENDICES

Appendix A: Glossary

Computer Supported Cooperative Work (CSCW, groupware, collaborative systems): Computer based systems that support groups of people engaged in a common task and that provide an interface to a shared environment.

Access Control: a mechanism for protection of data objects against unauthorized access. It ensures that all direct accesses to objects are authorized.

Subject: an entity that initiates an access request.

Object: an entity of the system over which a set of protected operations is defined.

Access right (access type): the privilege to do certain operation(s) over objects.

Access matrix: a matrix with rows representing subjects and columns representing objects, and A[s,o] denoting the access rights s has over o.

Access list: a representation of access matrix by columns, i.e., a list of pairs (subject, rights) associated with each object.

Capability list: a representation of access matrix by rows, i.e. a list of pairs (object, rights) associated with each subject.

Access policy: a configuration of an access matrix.

Access mechanism: a set of facilities used to implement access policies.

Negative right: explicit denial of an access right.

Authorizer: a subject who specifies the access rights.

Role: an abstraction that bears a unique name, and is associated with a member list and a set of access rights.

Authentication: Verification of the identity of a subject that makes an access request.

Coupling: sharing of data and its properties (including interactive properties such as views and formats), perhaps simultaneously, among multiple users.

Flexible coupling: applications can decide which properties of objects are shared and how they are shared.

WYSIWIS coupling: all properties of the objects are shared so that "What You See Is What I See".

Dialogue manager: a structure editor that manages all interactions between a user and an application.

Inheritance: the acquisition of certain properties from parents if the child does not have them.

Multi-inheritance: inheritance from multiple sources. It may cause conflicts.

Appendix B: Access Rights and Groupings

```
AllR includes DataR
AllR includes ViewR
AllR includes FormatR
AllR includes CoupleR
AllR includes WindowR
AllR includes UserDefinedR
AllR includes RoleR
AllR includes SessionR

DataR includes UpdateR
DataR includes WriteR
DataR includes InsertR
DataR includes DeleteR
DataR includes ReadR

ViewR includes ElideR
ViewR includes HideR
ViewR includes SelectR

FormatR includes TitleR
FormatR includes FontR
FormatR includes IdentR
FormatR includes ColorR

WindowR includes CursorR
WindowR includes ScrollR
WindowR includes ResizeR
WindowR includes MoveR

CoupleR includes TransmitR
CoupleR includes ListenR
CoupleR includes ValueCoupleR
CoupleR includes ViewCoupleR
CoupleR includes FmtCoupleR

TransmitR includes TransmitEventR
TransmitR includes TransmitCorrectR
TransmitR includes TransmitPeriodR
TransmitR includes TransmitTimeR
TransmitEventR includes TransmitIncrementR
TransmitEventR includes TransmitCompleteR
```

```
TransmitEventR includes TransmitPeriodR
TransmitEventR includes TransmitTimeR
TransmitEventR includes TransmitTransmitR
TransmitCorrectR includes TransmitRawR
TransmitCorrectR includes TransmitParsedR
TransmitCorrectR includes TransmitValidatedR
TransmitCorrectR includes TransmitCommittedR
ListenR includes ListenEventR
ListenR includes ListenCorrectR
ListenR includes ListenPeriodR
ListenR includes ListenTimeR
ListenEventR includes ListenIncrementR
ListenEventR includes ListenCompleteR
ListenEventR includes ListenPeriodR
ListenEventR includes ListenTimeR
ListenEventR includes ListenTransmitR
ListenCorrectR includes ListenRawR
ListenCorrectR includes ListenParsedR
ListenCorrectR includes ListenValidatedR
ListenCorrectR includes ListenCommittedR

RoleR includes ReadRoleR
RoleR includes ModifyRoleR
RoleR includes CreateRoleR
RoleR includes DeleteRoleR
RoleR includes InsertMemberR
RoleR includes RemoveMemberR
RoleR includes TakeRoleR
RoleR includes LeaveRoleR

SessionR includes ReadSessionR
SessionR includes CreateSessionR
SessionR includes DeleteSessionR
SessionR includes ModifySessionR
SessionR includes RemoveParticipantR
SessionR includes JoinSessionR
```

Appendix C: *Imply* Relationships

```
UpdateR implies WriteR
WriteR implies InsertR
WriteR implies DeleteR
InsertR implies ReadR
DeleteR implies ReadR

ElideR implies SelectR
HideR implies SelectR

TransmitIncrementR implies TransmitCompleteR
TransmitCompleteR implies TransmitPeriodR
TransmitPeriodR implies TransmitTimeR
TransmitTimeR implies TransmitTransmitR
TransmitRawR implies TransmitParsedR
TransmitParsedR implies TransmitValidatedR
TransmitValidatedR implies TransmitCommittedR
ListenIncrementR implies ListenCompleteR
ListenCompleteR implies ListenPeriodR
ListenPeriodR implies ListenTimeR
ListenTimeR implies ListenTransmitR
ListenRawR implies ListenParsedR
ListenParsedR implies ListenValidatedR
ListenValidatedR implies ListenCommittedR

OwnerR implies OListR
OwnerR implies OListR*
OwnerR implies AllR
OwnerR implies AllR*

AddMemberR implies TakeRoleR
RemoveMemberR implies LeaveRoleR
ModifyRoleR implies AddMemberR
ModifyRoleR implies RemoveMemberR
InsertMemberR implies ReadRoleR
RemoveMemberR implies ReadRoleR

ModifySessionR implies CreateSessionR
ModifySessionR implies DeleteSessionR
ModifySessionR implies RemoveParticipantR
JoinSessionR implies ReadSessionR
```

```
CreateSessionR implies ReadSessionR
DeleteSessionR implies ReadSessionR
RemoveParticipantR implies ReadSessionR
```

Appendix D: Access Rules

Access Checking Rule: *A subject s has access privilege r over object o if and only if*

a). A[s,o] contains a positive right r.

or

b). A[s,o] does not contain a negative right r, and F(s,o,r,A) evaluates to true.

Object Inheritance and Conflict Resolution Rule: *A right r of subject s on object o is inherited from the value groups containing o that are chosen by the inheritance directive, i.e.,* $F(s, V(o,r), r, A) \rightarrow F(s, o, r, A)$ *and* $F(s, V(o,r), -r, A) \rightarrow F(s, o, -r, A)$ *where V(o,r) is the set of value groups specified by the inheritance directive associated with o and r, r is either a positive or a negative right, and* $\rightarrow$ *denotes "imply". In case of conflicts, the access definition in the first value group chosen by the inheritance directive is used.*

Right Inheritance Rule: *A positive or negative right r of subject s on object o is inherited from the right groups it belongs, i.e.* $F(s, o, R, A) \rightarrow F(s, o, r, A)$ *and* $F(s, o, -R, A) \rightarrow F(s, o, -r, A)$ *where R includes r.*

Right Inference Rule: *A right r of subject s on object o, if undecided, is inferred from the rights that imply r, i.e.,* $F(s, o, rx, A) \rightarrow F(s, o, r, A)$ *and* $F(s, o, -r, A) \rightarrow F(s, o, -rx, A)$ *where rx implies r.*

Right Conflict Resolution Rule: *The imply relationship is used in preference to the include relationship in case of conflicts.*

Subject Inheritance Rule 1: *Subjects inherit both positive and negative rights from the take relationship, i.e., if s takes the role of S, then* $F(S, o, r, A) \rightarrow F(s, o, r, A)$ *and* $F(S, o, -r, A) \rightarrow F(s, o, -r, A)$.

Subject Inheritance Rule 2: *If r is a coupling right involving passive subjects s and s2, and s2 takes the role of s, then* $F(s1, (o, s), r, A) \rightarrow F(s1, (o, s2), r, A)$ *and* $F(s1, (o, s), -r, A) \rightarrow F(s1, (o, s2), -r, A)$.

Subject Inheritance Rule 3: *If subject s1 has right r of subject s2, i.e., have(s1,r,s2) = true, then* $F(s2, r, o, A) \rightarrow F(s1, r, o, A)$ *where r is a positive right.*

Subject Conflict Resolution Rule 1: *A more specific role defined by the take relationship should be used first.*

Subject Conflict Resolution Rule 2: *In case of conflicts, a hint specified by users is used to determine which role should be used first. In particular, we choose the access definition that appears earliest in the access list.*

Subject Conflict Resolution Rule 3: *The take relationship is used in preference to the have relationship in case of conflicts.*

***-Right Inheritance Rule 1:** *The imply relationships among non-*-rights are mapped to the *-rights, i.e.,* $imply(R1, R2) \rightarrow imply(R1*, R2*)$ *where R1 and R2 are non-*-rights.*

***-Right Inheritance Rule 2:** *The include relationships among non-*-rights are mapped to the *-rights, i.e.,* $include(R1, R2) \rightarrow include(R1*, R2*)$ *where R1 and R2 are non-*-rights.*

Ownership Semantic Rule: *Ownership is a special right defined by associating a set of imply relationships with the right.*

Ownership Initialization Rule: *A subject who creates an application owns all the initial variables defined within the application unless they have predefined owners. A subject who dynamically creates a new variable owns that variable.*

Multiple Ownership Rule: *An initial owner of an object can add roles into or remove roles from its owner list. The owners of an object are the set of users that take a role in the owner list.*

Ownership Default Rule: *By default, ownership implies OlistR and OlistR*.*

Ownership Inheritance Rule: *The owner list of an object o, if unspecified, is inherited from the value groups containing o that are chosen by the inheritance directive. In case of conflicts, the ownership list defined in the first value group chosen by the inheritance directive is used. By default, inheritance starts from structure value groups.*

VITA

VITA

HongHai Shen was born on June 24, 1964, in Shanghai China. He received a B.S. and M.S. degree in computer science from Fudan University, Shanghai, China, in July 1985 and July 1988. He received a M.S. degree in computer science from Purdue University, West Lafayette, Indiana, in May 1991. He is a member of ACM and a member of Upsilon Pi Epsilon honorary society of computer science. He is currently a member of the technical staff at the IBM Corporation.