Access Control for Collaborative Environments

HongHai Shen and Prasun Dewan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
hhs@cs.purdue.edu and pd@cs.purdue.edu

ABSTRACT

Access control is an indispensable part of any information sharing system. Collaborative environments introduce new requirements for access control, which cannot be met by using existing models developed for non-collaborative domains. We have developed a new access control model for meeting these requirements. The model is based on a generalized editing model of collaboration, which assumes that users interact with a collaborative application by concurrently editing its data structures. It associates fine-grained data displayed by a collaborative application with a set of collaboration rights and provides programmers and users a multi-dimensional, inheritance-based scheme for specifying these rights. The collaboration rights include traditional read and write rights and several new rights such as viewing rights and coupling rights. The inheritance-based scheme groups subjects, protected objects, and access rights; allows each component of an access specification to refer to both groups and individual members; and allows a specific access definition to override a more general one.

KEYWORDS

CSCW, groupware, access control, protection, security, user interface.

1 INTRODUCTION

Recently, there has been much research done in computer applications for facilitating collaboration among multiple, distributed users. However, there has been relatively little work done in controlling access to the collaboration. Most collaborative systems give all collaborators the same rights to all objects and expect access to be controlled by social protocol. Thus, they do not provide computer support for preventing mistakes, conflicting changes, or unauthorized access. The major exception known to us is the GROVE outline editor [5]. A major reason for the lack of access control in collaborative applications is the absence of a generic access control model for collaborative systems that meets the requirements of collaborative systems.

Although generic access control models have been studied extensively in non-collaborative domains, none of them meets the requirements of collaborative environments. We have designed a new access control model to meet these requirements. The model is based on a generalized editing model of collaboration, which assumes that users interact with a collaborative application by concurrently editing its data structures. It associates fine-grained data displayed by a collaborative application with a set of collaboration rights and provides programmers and users with a multi-dimensional, inheritance-based scheme for specifying these rights. The collaboration rights include traditional access rights such as read and write rights and several new rights such as the right to change the shared view of a displayed object and the right to couple with other users [3]. The inheritance-based scheme groups subjects, protected objects, and access rights; allows each component of an access specification (subject, object, right) to refer to both groups and individual members; and allows a more specific definition to override a more general one. We have implemented the model as part of the Suite generic framework for supporting editor-based collaborative applications.

In this paper, we use the concrete example of Suite to describe our access model. Section 2 describes the new requirements for access control raised by collaborative systems. Section 3 gives an overview of our approach for meeting these requirements. Section 4 briefly outlines the Suite multi-user framework used as the basis of our access model. Section 5 gives the details of the model, showing how we have extended the conventional access control model to meet our requirements. Finally, Section 6 compares our work with

[1]This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD-8913133), and by National Science Foundation Grant IRI-9015442.

related work, gives conclusions, and presents directions for future work.

2 ACCESS REQUIREMENTS

We have identified several requirements that a generic access control model for collaborative environments should support:

Multiple, dynamic user roles: The model should allow users' access rights to be inferred from their roles [9]. Moreover, it should allow users to take multiple roles simultaneously and change these roles dynamically during different phases of collaboration.

Collaboration rights: Besides traditional operations such as read and write, all other operations whose results can affect multiple users should be protected by collaboration rights.

Flexibility: The system should support fine-grained subjects, objects, and access rights, that is, it should allow independent specification of each access right of each user on each object. For instance, it should allow independent protection of each line in a multi-user code viewer.

Easy specification: Users should be able to specify access definitions easily.

Efficient storage and evaluation: The storage of access definitions and evaluation of the access checking rule should be efficient.

Automation: The model should make it easy to implement access control in multi-user applications.

These requirements are motivated in more detail in the following sections along with our approach for meeting them.

3 OVERVIEW OF APPROACH

The classic access matrix model proposed by Lampson [8] and refined by Graham and Denning [7] provides the basic framework to describe protection systems. Conceptually, the model describes what a protection state is and how state transitions occur. A protection state is defined by the triple (S, O, A), where S is a set of *subjects* (entities wishing to access data), O is a set of *objects* (units of data that may be accessed), and A is a matrix with rows representing subjects, columns representing objects and A[s,o] denoting the *access rights* (privileges that are needed to do certain operations on objects) a subject s has over an object o. The model includes an access checking rule which ensures that a request for accessing object o by subject s is denied if A[s,o] does not contain the requisite right. The rule assumes that the subject requesting access is uniquely identified from the running process [1]. The model also contains a set of commands specifying how to make state transitions, i.e., to change (S, O, A).

The above model does not specify the exact nature of the subjects, objects or access rights supported by the system. Moreover, it does not require a particular mechanism to specify elements of the access matrix and has been used, for instance, to support specifications based on both capability lists and access lists. However, it is not sufficient for meeting the collaboration requirement of (i) multiple user roles, since a process is always associated with a unique subject at any moment; (ii) easy specification, since it does not address specification of access rights, (iii) automation, since it does not offer an application mechanisms for managing and specifying the access matrix, and implementing the state transition commands and the access checking rule.

We have extended the conventional model in several ways to overcome these limitations of it:

1. Collaboration Rights: We define a new set of access rights designed for the Suite generic collaboration model.

2. Negative Rights: We support the notion of *negative rights* to allow explicit denial of access.

3. Inheritance-based Specification. We support an extended access matrix that supports not only individual subjects, objects, and rights but also groups of these entities. The value of an element of the access matrix does not have to be specified directly and can be inferred from values of other elements of the matrix.

4. Automation: Our model includes mechanisms that relieve an application of the task of implementing the details of access control. These mechanisms are based on the Suite multi-user framework, which relieves an application of the task of implementing the details of the Suite collaboration model.

We have implemented these extensions in Suite. In the following sections, we use the example of Suite to motivate, describe, and illustrate our access control model and compare it with related works.

4 SUITE MULTI-USER FRAMEWORK

As discussed above, our access control model is based on the Suite multi-user framework. The framework allows all interaction with the system to be modeled as concurrent editing of fine-grained data structures called *active variables*. Each active variable is associated with a set of *attributes*, which determine various interactive properties of the variable such as its format and which properties of it are shared among users. A system-provided structure editor called a dialogue manager handles a user's interaction with the system. The framework is described in more detail in [4].

To illustrate the framework, consider the example of a Suite application called Ctool, which allows multiple users to edit and test C functions. The active variables of Ctool are C programs, functions, and lines. Figure 1 shows two users, `rxc` and `hhs`, interacting with the tool. In this session, the users have set the `ValueCoupled` and `ViewCoupled` attributes [3] of the active variables to `True`, thereby sharing both their values and views. As a result, when user `rxc` changed the view of the `minit` procedure by "eliding" it[2], the result was shared by both users, as shown in the figure.

The model described above does not include access control. It allows, for instance, user `rxc` to elide arbitrary pro-

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

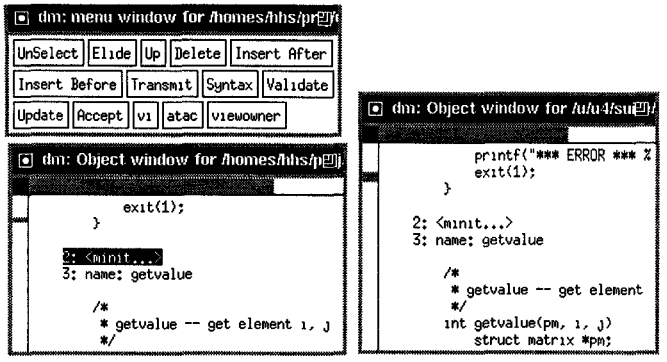

Figure 1: Two users in a collaborative editing session: when `rxc` (left) elides a function `minit` by pressing the `elide` button, `hhs`'s view is affected (right).

cedures being viewed by user `hhs`, which is not desirable if the former is required only to play the role of an "observer". It even allows `rxc` to change or delete arbitrary functions inserted by `hhs`. The model described in the next section overcomes this problem.

5 THE SUITE ACCESS MODEL

5.1 Positive and Negative Rights

Like conventional systems, Suite supports positive rights, that is, allows explicit granting of a right. In addition, it also supports negative rights, that is, explicit denial of a right. This notion is borrowed from a previous work on access control for database systems [10]. In this work, it was used along with the notion of strong and weak authorization to facilitate the access specification in the presence of a large set of data objects, which also applies to our environment. To illustrate this use of negative rights, assume a user is to be denied read access to one line of a 1000-line-long program. Instead of granting him the positive read right over 999 other lines, it is simpler to grant him a (weak) positive read right over the program and a (strong) negative read right over the line. We have found that negative rights can also be used to ease the specification task for CSCW applications supporting dynamic roles. To illustrate this use, consider the scenario where a user `hhs` wants to grant the read access of a private comment line to all users taking the `suite` role except `rxc`. Supporting only positive rights would require that `hhs` identify all `suite` users and explicitly grant each one of them except `rxc` the positive read right. This approach, however, suffers from two related disadvantages. First, it is painful for `hhs` 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 negative rights, the above can be achieved by granting a positive read right to `suite` and a negative read right to `rxc`. The use of negative rights can also reduce the time required to search the inheritance space, as discussed in section 5.6.

5.2 Protection State

The Suite protection state is represented by a 4-tuple (S,O,A,F), where S is the set of subjects consisting of both individual users and their groups, O is is the set of objects consisting of individual objects and their groups, A is the extended matrix with A[s,o] representing *explicitly* specified rights subject s has over object o, and F is an inference function allowing inference of rights that have not been explicitly specified. F takes as arguments a (subject, object, right, A) tuple and returns the value {`True, False, Undecided`}. Our access checking rule is the following modification of the conventional access rule:

Access Checking Rule: *To check the access privilege r of subject s over object o, A[s,o] is first consulted. If A[s,o] contains a positive or negative r, then access is granted or denied, respectively. Otherwise access is granted only if F(s,o,r,A) evaluates to true.*

Sections 5.3,5.4 and 5.5 describe the three dimensions of the extended access matrix and the associated inference rules in detail.

5.3 Object Dimension

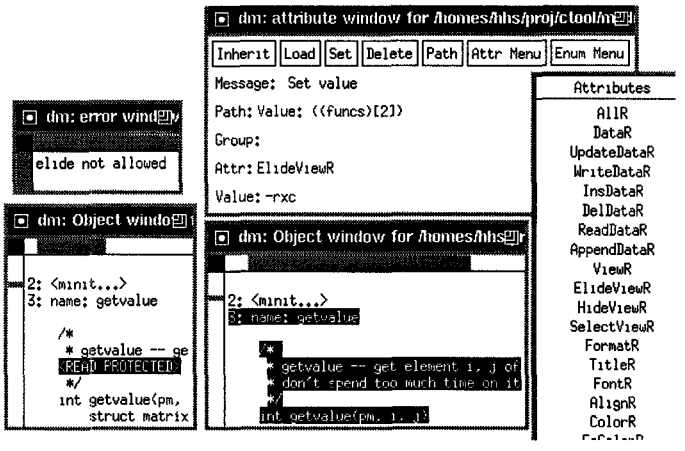

Figure 2: Fine grained specification: user `hhs` (right windows) denies `rxc` the elide right to the function `getvalue`. Thus, `rxc` (left windows) gets an error message when he tries eliding. The figure also shows an even finer-grained specification: `hhs` protects a comment line from being read by `rxc`.

Suite supports protection of fine-grained active variables. Continuing with the example of Figure 1, it allows user `hhs` to prevent user `rxc` from eliding a procedure he is currently working on (Figure 2). To specify this access definition, `hhs` first selects the function `getvalue` from the object window; presses the `Path` button in the attribute window, which fills out the `Path` field of the selected object in the window; selects the `ElideViewR` right from the menu; fills the `Value` field with "-rxc" and presses the `Set` button, which enters the access definition into the matrix. The figure also shows the result of a finer-grained specification, which

disallows user rxc from reading a comment line.

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. To illustrate, assume that the user hhs does not want rxc to delete any line in a procedure he has just added. In this situation, he would have to specify an access definition for each of these lines and the system would have to store all of these definitions.

Therefore, Suite allows a user to specify an access definition for a group of active variables called value groups, and allows a specific definition to override a more general definition. Values are grouped on the basis of several properties including their structural parents and types. Because an active variable can belong to multiple value groups, an inheritance directive is associated with each active variable defining from which of these groups the specification should be inherited and the order in which they should be searched. This approach was first proposed for specifying display formats and other attributes of active variables by Dewan [2] and is described in detail there. We use this approach in the following inheritance rule for the object dimension:

Object Inheritance and Conflict Resolution Rules: *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.,* $F(s, V(o, r), r, A) \rightarrow F(s, o, r, A)$ *where V(o,r) are the value groups specified by the inheritance directive associated with o and r. In case of conflicts, the access definition in the first value group chosen by the inheritance directive is used.*

To illustrate this rule, assume that user hhs wants to disallow user rxc from deleting any line of the function getvalue. He can achieve this control by granting rxc a negative delete right on the function getvalue, which causes any line within the function to inherit this definition. In this example, access rights were structurally inherited. They can also be inherited from types. For instance, if hhs denies rxc the eliding right to type function, all objects of this type (that is, all functions) inherit this access specification.

5.4 Access Dimension

The Suite coupling model allows users to share the results of all operations on an active variable [3]. To meet the requirement of collaboration rights, our access model associates each of these operations with its own access right. For example, it defines the ElideViewR right, which determines whether a user can elide a variable; the ColorR right, which determines whether a user can change the color of a variable ; the RawListenR right, which determines whether a user can receive raw changes to a variable made by another user; the ParsedListenR right, which determines whether a user can receive syntactically checked changes to a variable made by another user; the RawTransmitR right, which determines whether a user can transmit raw changes to a variable to another user; the ValueCoupledR right, which determines whether a user can share value changes to a variable with another user; the FormatCoupledR right, which determines whether a subject can share formatting properties of a variable with another user, and so on. In all, Suite currently supports over 50 rights. We call these *individual rights* as opposed to the right groups described below.

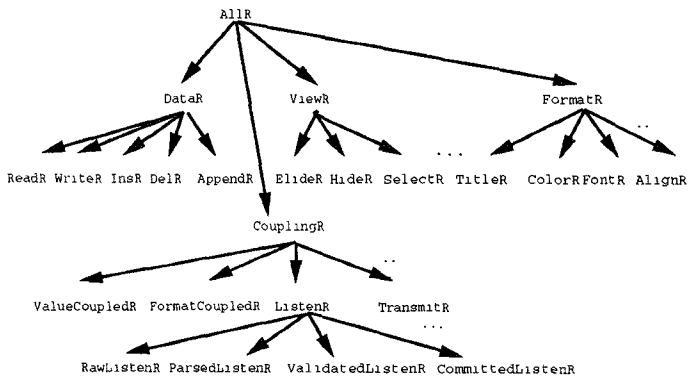

Figure 3: *Include* relationship among access rights

Introducing a large set of rights brings the associated problems of specification and storage of rights. To solve the problem, we introduce the notion of *right groups*, which are defined by classifying the individual rights into logical categories. Figure 3 gives part of the tree hierarchy of the grouping of rights where a transitive relationship *include* is defined. For example, RawListenR and ParsedListenR are included in the right group ListenR, RawTransmitR and ParsedTransmitR are included in the right group TransmitR, and ListenR and TransmitR are included in the right group CouplingR. Note that unlike an individual right, a right group does not correspond to any specific operation on an object.

The inheritance rule below defines the semantics of the *include* relationship.

Right Inheritance Rule 1: *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.*

Continuing with the example, assume that after some collaboration, hhs thinks he can trust rxc with several rights including the DeleteR, InsertR, ElideR, and HideR rights, but he still wants to restrict the coupling rights of rxc. He only has to grant rxc the corresponding right groups DataR and ViewR. Now rxc can invoke the insert, delete, elide, etc., operations. But he still cannot change the coupling. Fine-grained access rights together with right groups gives users fine-control over collaboration while relieving him from the task of tediously specifying all the rights.

We motivated and described above a relationship called *include* for grouping rights. This relationship alone is not

sufficient for supporting easy specification of and maintaining consistency among access rights. Consider the following scenario. A new user `abc` joins the `serc` role and is not familiar with the system. To ensure security, `hhs` denies `abc` the data rights on the function he just created. Also assume that after some time, `hhs` decides that it is now safe to allow `abc` to insert new lines into the function though he still does not want him to delete anything in it. Naturally, this implies that `abc` should get the right to read the function. In some systems such as Unix, `hhs` has to grant the read right to `abc` explicitly, which is unnecessary. In our model, however, `hhs` does not have to do so explicitly since `read` is implied automatically by `insert`. This is an example of the *imply* relationship over individual access types, which is used to model the fact that the right to do a more powerful operation guarantees the right to do a less powerful one. It is similar to the *include* relationship in that it is also used to infer rights. The differences between the *include* and *imply* relationships are as follows: The *include* relationship is defined over individual rights as well as right groups while the *imply* relationship is defined only over individual rights. Moreover, unlike the *include* relationship, only positive authorizations are inherited from the *imply* relationship while negative authorizations are not. For example, `write` implies `read`, but negative `write` does not imply negative `read`, although negative `read` does imply negative `write`. Fernandez et al [6] suggested this approach for database systems and identified *imply* relationships among traditional access rights. We believe this approach is crucial for the more complex collaborative systems for two reasons. First, it reduces the problem of specifying the large number of rights necessary in a collaborative system. Second, in a collaborative system, users are prone to specifying conflicting rights because they have to specify, often dynamically, a large number of access rights. The *imply* relationship can be used by the system to detect 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 negative `read` implies negative `write`.

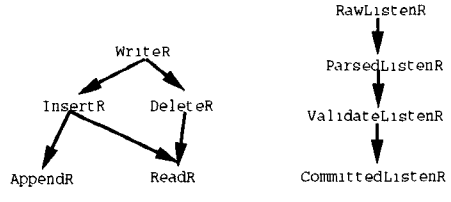

Figure 4: Some *imply* relationships among individual access rights

Therefore, we support the notion of implication of rights and have identified and implemented the *imply* relationships among the Suite access rights. Figure 4 gives some of these relationships. These relationships allow, for instance, the `RawListenR` to imply the `ParsedListenR` since a user in Suite listening for raw values of active variables can also listen for parsed values [3].

The following inheritance rule summarize the above discussion:

Right Inheritance Rule 2: *A right r of subject s on object o, if undecided, is inherited from the rights that imply r, i.e.,* $F(s, o, rx, A) \rightarrow F(s, o, r, A)$ *where rx implies r.*

Supporting both the *include* and *imply* inheritances allows an access right to be inherited from multiple sources. To illustrate, consider the situation above where user `abc` was granted a negative `data` right and a positive `insert` right. We have a conflict since the negative `read` right can be inferred from the first definition while the positive `read` right from the latter. 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. Furthermore, 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. Thus, in the scenario above, the imply relation is used to grant `abc` the read right. Our conflict resolution rule is summarized below:

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

5.5 Subject Dimension

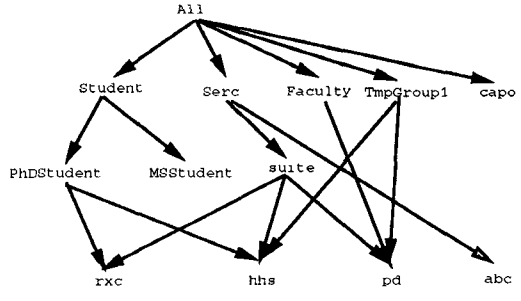

Figure 5: *Take*: inherit all rights of a role

Like conventional systems, Suite allows a subject to be a specific user. In addition, it allows a subject to be a *role* and allows a user or role to "take" a role. A subject takes a role if a *take* mapping is defined between the subject and the role, which can be changed dynamically. A subject can take multiple roles simultaneously. Moreover, multiple subjects can take a role. As a result, the notion of a role includes the concept of a user-group supported by traditional systems.

The take relationship allows a subject to inherit the rights of its roles according to the following inheritance rule:

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* $F(S, o, -r, A) \rightarrow F(s, o, -r, A)$.

Continuing with the Ctool example, 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 by taking the `faculty` role while `rxc` is denied the right by taking the `student` role.

We also support another inheritance relationship among subjects which is useful in maintaining access privileges. There are certain inheritance relationships among subjects which take only a subset of the positive access rights (privileges) of others. We 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. Figure 6 illustrates some *have* relationships of Ctool, which allow a `PhDStudent` to have all the data rights of `MSStudent` and `pd` (manager) to have all the data rights of `rxc` and `hhs` (employees). We support inheritance of selected rights rather than all the rights because it reflects the "least privilege" principle [11] more faithfully. The following inheritance rule reflects the semantics of *have*:

Subject Inheritance Rule 2: *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.*

Allowing a user to take multiple roles in the above manner introduces potential conflicts. Continuing with the Cool example, if `serc` is granted the positive right and `student` is granted the negative right for some data object, how should we decide the right for `rxc` who takes the role of both `student` and `serc`?

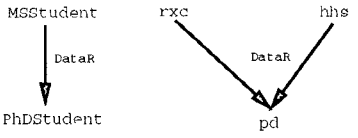

Figure 6: *Have*: inherit selected privileges

One approach to resolve this problem is to check the consistency of all of the access definitions and disallow any conflicts. However, this approach is inappropriate in a collaborative environment where the set of access objects can be very large and dynamic, and a user may take dynamic, multiple roles. Whenever a new role is assigned, or 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 result. Checking all these conflicts is costly and sometimes even unnecessary when a particular user may need to access only a very small part of a large set of objects. Thus, it is not only too rigid, but also computationally expensive to adopt the approach of simply disallowing conflicts.

To illustrate our solution, let us turn to some more examples. First, we observe that the *take* relationship actually defines an inheritance hierarchy which decides which role is more "specific". For instance, if `rxc` is checked against the access list `(+all-student+PhDStudent)`, although he takes all of the three roles in the list, he should be granted the right because `PhDStudent` is the most specific role he takes in the list. 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, there are cases when it is impossible to decide which role is more specific as in the case of the access list `+serc-student` where there is no *take* relationship between the two roles, and the effects of `+serc-student` on subjects taking both roles can be interpreted as either positive or negative. In this situation, we interpret the access list to be "position sensitive", which corresponds to the heuristic of "putting important things first". According to this rule, `-student+serc` will grant negative rights to those taking both roles while `+serc-student` positive rights. This is our second rule in resolving conflict:

Subject Conflict Resolution Rule 2: *In case of conflicts, the access definition that appears earliest in the access list is chosen.*

By adopting the above rules, our approach provides a simple but flexible solution to this conflict problem. In one extreme case, all the negative definitions can be put before the positive ones to ensure maximal security `(-student-capo+faculty+serc)`. In the other extreme case, all the positive ones can be put before negative ones to ensure maximal sharing `(+faculty+serc-student-capo)`. Between these two extremes, users can adjust their needs by putting subjects at different positions. In conventional systems such as Unix and [10], a user is given the union of the rights that all of his roles can have. In Multics [11], a user is asked to supply upon login which role and compartment he will take and his rights in the session is subsequently checked against the `(individual, project, compartment)` triple. This approach, however, actually does not allow multiple roles of a user in a session.

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 the algorithm we use, the *take* relationship is used in preference to the *have* relationship because of the following reasons: First, the *take* relationship can infer both positive and negative rights, which is safer than the *have* relationship as the latter can only infer 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 users or 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.*

5.6 Multi-dimensional Inheritance and Extended Access Lists

In the above sections, we have described multiple inheritances in the object, type, and subject dimensions. There is yet another important question we have to answer: in which order are the inheritances in the three dimensions used when calculating a right? The answer determines both the semantics of the access model and the efficiency of the lookup. It is not currently clear to us which conflict resolution method is the most semantically meaningful. Therefore, we choose the method based on the efficiency considerations, which are related to the access data structures used to represent the access matrix. Traditionally, two main data structures have been chosen to represent the access matrix: capability lists and access lists. The first stores the matrix by rows, i.e., each subject is associated with a list of pairs (object, rights) called capabilities. The second approach stores the matrix by columns, i.e., each object is associated with a list of pairs (subject, rights) [7]. If we use the capability list mechanism, we should start the search from the subject dimension because it is the most efficient way to search the underlying data structures. If we use the access list mechanism, we should start the search from the object dimension for the same reason.

We did not choose the capability list mechanism because we found it difficult to devise a simple way using the capability mechanism to resolve conflicts arising from supporting multiple roles. Consider the `+serc-student` example again in the preceding section. Assume that the capability approach is adopted. Also assume the `serc` role has positive rights to procedure f1 and f2 and the `student` role has negative rights to them. Further assume that if conflicts arise we want the positive `serc` role to win in the case of f1 and the negative `student` role to win in the case of f2. According to this requirement, the capability list of the `serc` role must be used when the protection object is f1, while the capability list of the `student` role must be used when the object is f2. Thus, which capability list should be used depends on the object being accessed. This suggests that the information for the choice should be attached to objects, which can be more straightforwardly supported by an access list approach.

We have used an extended form of access list in our implementation of the model. In our approach, an object is associated with a list of pairs (r, u-list), where r is a right and u-list is a string of the form $(\omega_1 U_1 \omega_2 U_2 ... \omega_i U_i)$. Here, ω_j can be either + or - indicating a positive or negative right, and U_j is a subject, where $1 \leq j \leq i$. When a subject s is checked against the access rights r on object o, we first search the access list associated with o to get the (r, u-list) pair. The u-list, which is sorted according to the topology of the *take* relationship, is then used for inheritance in the subject space. If no definite right can be calculated from the subject space, inheritance in the rights dimension is used. If no definite right can be inferred based on the access list associated with the object, we use inheritance in the object dimension to infer the right. Finally, if we still do not find a definition for the right, we infer a negative right.

As shown above, the notion of negative rights makes the inheritance scheme more complicated. Nevertheless, we support negative rights for two related reasons. First, introducing negative rights can make access specification much easier, as discussed above. Second, 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.

5.7 Automation

The various aspects of the access control model described above allow end-users to easily and flexibly specify access rights but make it difficult for programmers of collaborative applications to implement access control. It requires implementation of the user interface for specifying access definitions, the extended access lists, and the various rules given above. We have automated this tedious process by implementing access control completely in Suite dialogue managers, which are provided by the system and which control all interactions between users and applications. As a result, the access awareness in an application such as Ctool is kept low since it is responsible only for specifying access control and not implementing it.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have identified access control requirements for a collaborative environment and presented a new access control model for meeting these requirements.

The main components of our model are:

1. A generic set of collaboration rights.
2. Fine-grained specification of access rights.
3. Multiple, dynamic user roles.
4. A set of inheritance rules in the subject, object, access right dimensions.
5. A set of conflict resolution rules in the multiple dimensions.

Some of the ideas in the model are based on a variety of previous works. The idea of *imply* relationship among access rights has been suggested in [6] which is extended by us to support collaboration rights. The concept of the *include* relationship among rights has not been discussed in other works to the best of our knowledge. Inheritance among objects takes its origin from the work in this area to support flexible displays and flexible coupling [3, 2], which has not been used before for access control. The idea of inheritance among subjects has been proposed by database researchers[6, 10]. We have extended their work by providing two kinds of inheritances: *take* according to user roles and *have* according to user privileges, and comparing these two relationships for

inheriting access rights. By allowing inheritance of *have* to be based on a selected subset of rights, the "least privilege" principle is reflected more faithfully in our model. We have also developed rules for resolving role conflicts that employ the "position first" rule to resolve conflicts.

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

It has been observed in [5] 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". Our access control model is a first-cut effort at meeting this goal. It is important to mention that while our model is designed for a collaborative environment, parts of it are also suitable for non-collaborative domains in which the set of objects, subjects, and rights are large.

A current limitation of our model is that its set of protected objects includes only active variables. We are investigating mechanisms for protecting other shared objects such as windows, sessions and subjects. We are also investigating policies for using our model. We plan to devise rules for checking consistency of access requirements based on these policies.

Acknowledgments: Discussions with Rajiv Choudhary, John Riedl and Tim Korb have been helpful in developing the model.

References

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

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

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

[4] Prasun Dewan and Rajiv Choudhary. Primitives for programming multi-user interfaces. *Proc. of the 4th ACM SIGGRAPH Symp. on User Interface Software and Technology*, pages 69–78, November 1991.

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

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

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

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

[9] 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.

[10] 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.

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