A Group-based Authorization Model for Cooperative Systems

Klaas Sikkel
GMD-FIT, German National Research Center for Information Technology
klaas.sikkel@gmd.de

Requirements for access control in CSCW systems have often been stated, but groupware in use today does not meet most of these requirements. There are practical reasons for this, but one of the problems is the inherent complexity of sophisticated access control models. We propose a general authorization model that emphasizes conceptual simplicity and show that several issues—in particular negative access rights and delegation of rights—can be solved elegantly in this model.

1 Introduction

Traditional access control models from the operating systems and database world do not meet the needs of groupware systems. This was stated by Greif and Sarin (1986) more than a decade ago. Since then, some work on access control has been done in the CSCW community, but limited progress has been made. A handful of access control models specifically designed for collaborative environments has been published. Some have gained acceptance at least as a theoretical contribution to the field, notably the model of Shen and Dewan (1992); none enjoy large scale usage as part of a widely used CSCW system.

The requirements for access control are known, but these seem to have had little impact in the construction of actual systems. What are the obstacles that prevent system designers from supplying groupware with adequate access control?

Firstly, as has been observed by several authors (Ellis et al., 1991; Shen and Dewan, 1992), access control models for groupware tend to be rather complex. It

J Hughes et al. (eds.), Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 345-360.

is far from trivial to design a user interface that offers the user an adequate set of access control operations and is easy to understand.

A second cause is of a more mundane nature. In prototype systems—and the large majority of systems discussed in technical CSCW literature are prototypes—access control is a feature that can always be added "later." A prototype without proper access control can be employed in first tests, whereas access control without a running prototype doesn't show very well... Hence the natural tendency *not* to make access rights a priority.

A telling example is the development of our own system, "Basic Support for Cooperative Work" (*BSCW*), offering shared workspaces on the World Wide Web (Bentley et al., 1995; Bentley et al., 1997). Thousands of users accept its shortcomings because it provides some essential features not found in other systems: simple cross-platform data sharing in distributed groups, within one's regular working environment. User feedback showed a need for more powerful and easy to use access control, yet it took more than a year after the system's public release before we started designing the access control model that is currently being implemented.

Although our work has been motivated by the immediate needs of the BSCW system, the access control model presented in this paper is of a general nature. We address several issues in groupware authorization—in particular negative rights and delegation—at a fundamental level and propose general, simple solutions.

After reviewing some important issues in Section 2 we present the authorization model in Sections 3 and 4; Section 5 briefly discusses access control in BSCW. Conclusions follow in Section 6. A more formal and more elaborate presentation of the model can be found in a technical report (Sikkel, 1997).

2 Issues in authorization and groupware

A distinction is made between *authentication* (verifying that you are who you pretend to be) and *authorization* (what it is that you are allowed to do). There is also a distinction between *access rights* (here used as a synonym of authorization) and *access control*: ensuring that the rights are not violated. We focus on authorization.

The following issues are commonly mentioned in relation with access rights for CSCW systems:

- *Application-oriented access rights.* Traditional authorization models originate from the operating systems and database worlds. The emphasis is on protection of data against unauthorized access, cf. (Salzer, 1974). An operating system defines access rights on the level of OS operations, but access rights should relate to the operations available to the user (Greif and Sarin, 1986). Synchronous groupware, in addition, may offer various levels of object sharing (Patterson at al., 1990).
- *Flexibility and ease of use.* Access rights in groupware may depend on who is doing what and therefore are highly dynamic (Trevor et al., 1994). Edwards

(1996) states that access rights should dynamically adapt to changes in the real world. Access rights modifications to be explicitly performed by the user should be easy to carry out and easy to understand.

- *Roles.* Authorization should be given to roles (e.g. teacher, assistant, student; designer, programmer, project leader; superuser), rather than to individual users. Users should be able to take multiple roles and change roles dynamically. On the other hand, users should not have to change roles explicitly when the system can infer their roles. Shen and Dewan (1992), Dewan et al. (1994), and Kanawati and Riveill (1995) argue that a user's permission should be the sum of the permissions of his roles. In a more security-oriented approach, cf. (Lampson et al., 1992; Coulouris and Dollimore, 1994b), the level of trust should be based on the intersection, not the union of the authorizations of a person's actual roles.
- *Delegation.* The system should allow delegation of rights from one person to another (who, depending on the type of delegation, may or may not further delegate it), with the possiblity to revoke delegated rights (Coulouris and Dollimore, 1994ab; Kanawati and Riveill, 1995).
- *Negative rights* are often stated as a requirement and present in several systems and models (Saltzer, 1974; Satyanarayanan, 1989; Shen and Dewan, 1992; Härtig et al., 1993). When a system allows complex hierarchical group structures, negative rights are required in situations where it is essential that some users (groups) are excluded from some rights.

And, of course, access control should be implemented efficiently, i.e., without noticeable loss of performance for frequently used operations.

Shen and Dewan (1992) state that authorization models satisfying these requirements are necessarily rather complex, but in Section 4.1 we will argue that their model is more complicated than needed. One of the issues we wanted to investigate is how complex an authorization model really needs to be. Hence Occam's Razor[1] has been used as a main design principle.

In Section 3 we present the basic model. It extends the canonical authorization model with a *single* concept: hierarchical group structures. in Section 4, we add several orthogonal extensions. These can be included in particular implementations, or—when an application does not need one of these extensions—left out, so as to reduce the complexity of the underlying model.

3 A simple authorization model

Authorization commonly involves three parameters: a *subject* (also called principal) has a *right* to perform an operation on an *object.* A classical way to organize this is

[1] "*Entia non sunt multiplicanda praeter necessitatem* " (Entities ought not to be multiplied except out of necessity.) – William of Ockham (1285-1349)

the *Lampson matrix* (Lampson, 1974), enumerating subjects in one dimension and objects in the other. Each cell contains all the rights given to the particular subject on the particular object. The Lampson matrix can be split into rows (columns) along either dimension, yielding *capabilities* (all rights on all objects for a given subject) or *access control lists* (*ACLs*) (all rights for all subjects on a given object). Sophisticated authorization models can be designed by adding structure to the dimensions of the authorization space.

Our model is ACL-based. For every object there is a structure describing which subjects have which rights. We use set theory for building these structures but talk about *groups*, rather than sets.

Leaving out all semantics, a group is a set, an unordered collection of entities, comprised in a single structure so that you can refer to it with a single name. Groups are used for different purposes.

Firstly, users may be organized in some group structure according to their position in an organizational hierarchy A project is composed of teams, a team of subteams, etc. In the literature groups are often called *roles*. In Section 4.3 we will differentiate between roles and groups, for the moment we consider them equivalent: for each role there is a group of persons that may act in this role, or reversed, for each group there is some role that all members of the group may act in.

A second type of group could be called *access group*. Suppose you are collaboratively working on a paper. You may distinguish between "writers", "annotators" (proof-readers giving comments) and "readers" of this paper. Each of these groups may perform an appropriate set of operations on the paper. The allowed operations as well as the composition of the groups may change over time.

3.1 User groups

In order to define user groups, we assume the existence of a domain of **users**, e.g., *tom*, *dick*, *harry*, etc., or more abstractly, $u, v, \ldots$. **User groups** are recursively defined as follows:

- A (single) user is a user group.
- A set of user groups is a user group.

Empty user groups are allowed in principle. A user group may not, directly, or indirectly, be contained in itself.

For convenience we have defined single users to be user groups as well. In the remainder of the paper we can refer to "a user group" meaning "an individual user or a group of users." In the rare cases where we need to refer to a composite group, not an individual user, we call this a **proper user group**.

User groups form hierarchical structures. If user group g is contained in user group h (i.e. $g \in h$), h is called a *subgroup* of g (and g a *supergroup* of h). The subgroup relation defines a directed acyclic graph with users as leaves and proper user groups as non-leaf vertices. An example is shown in Figure 1. Subgraphs can be shared: a user group can be a subgroup of different supergroups.

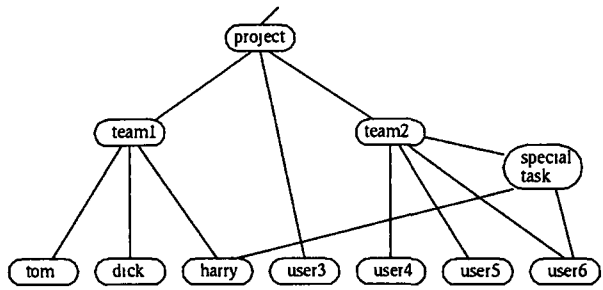

Figure 1: A hierarchical group structure

Members of a user group are those users who are, directly or indirectly, contained in the group (i.e., descendant leaves in the graph). Note the difference between subgroups and members. For example, in Figure 1, we have

$$team2.subgroups = \{user4, user5, user6, special\_task\},$$
$$team2.members = \{user4, user5, user6, harry\}.$$

It is possible to perform operation on groups. We describe only those operations that change the state of the system (as opposed to functions, like, e.g., *members*, that merely retrieve values). [*In brackets the operations are defined in terms of graph operations.*]

- **NewGroup** creates a new (empty) user group [*creates a new node*].
- **AddSubgroups** adds a set of user groups as subgroups to a given group [*adds edges*]. AddSubgroups fails if a group would (indirectly) be included in itself [*a cycle would be created*].
- **DeleteSubgroups** deletes subgroups from a user group [*removes edges*].
- **RemoveGroup** removes a user group from the system [*removes a node with its incident edges*]. This may cause its supergroups to lose some members.
- **DissolveGroup** removes a user group from the system without affecting the membership of other groups: subgroups of the dissolved group are added as subgroups to its supergroups [*removes a node; incident edges are replaced, linking each predecessor directly to each successor*]. This is illustrated in Figure 2; (a) shows part of a group structure; user group g has been removed in (b) and dissolved in (c).
- **RenameGroup** changes the name of a user group [*changes the label of a node*].

Additional composite operations, like splitting, merging, or inserting (the reverse of dissolving) user groups can be added as required. Note that RemoveGroup is not a primitive operation. It can be composed from DeleteSubgroups and DissolveGroup,

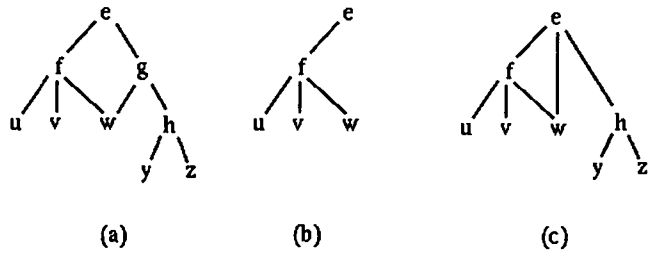

Figure 2: RemoveGroup (b) vs. DissolveGroup (c)

hence it could be discarded. It has been included, however, in order to point out the difference between removing and dissolving.

3.2 Objects and user groups

Access rights are to be defined for (classes of) objects, typically stored in a database of some kind. In general, such a database may hold different kinds of information. Disregarding *metadata*, used for the system's organization and not accessible to the user, we distinguish two categories of objects:

- Ordinary objects: documents, containers, programs, etc. Here we call them *regular objects*. These are the objects for which access rights have to be defined.
- In addition, the system may contain objects that cannot be accessed independently and appear to be attributes of other objects. We call these *attributed objects*.
  An example of an attributed object in the BSCW system is the "description" that an object (say, a document) may have: typically a line of text describing its nature, contents, or purpose. The description is regarded as an attribute of the document. Access to the description (see it, edit it) is governed by access rights defined *for the document*. Also, its existence depends on the existence of the document: if the document is deleted from the system, the description ceases to exist.

To which of both categories do user groups belong?

In the database world it is common to separate regular data and access rights, so that accessing data and changing access rights are regulated by different administrative procedures (Sandhu et al., 1996). In an object-oriented approach it seems rather more natural to see user groups as regular objects. There is an argument of elegance and simplicity: Groups themselves have to be accessed and manipulated, hence there is no reason why rights on user groups should be organized differently from rights on other objects.

We will use groups of either category. The issue of independent existence applies to groups in the same way it applies to other objects.

- A regular user group (*Rgroup*), is a regular object.
  Examples of Rgroups (cf. Figure 1) are *project* = {*team1*, *team2*} and *tom*.
- An attributed user group (*Agroup*) is an attribute of a regular object.
  As a consequence, accessing the Agroup is only possible through (and controlled by) the object to which it is attributed. Moreover, when that object is removed from the system, the Agroup ceases to exist.
  An example of the use of Agroups are the "writers," "annotators," and "readers" groups for a particular document.
  As a practical notation, we write, e.g., *obj.writers*, *obj.readers* for Agroups of an object *obj*.

Individual users are Rgroups, proper user groups can be of either type. Rgroups can be subgroups of Agroups and vice versa. For example in *obj1.readers* = {*obj2.writers*, *team1*} an Agroup is composed of an Agroup and an Rgroup. The group properties defined in Section 3.1 (having members, subgroups, etc.) are not related to the distinction between Rgroups and Agroups.

3.3 Rights

For each regular object we assume a set of rights $r_1 \ldots r_k$ where the number of rights k depends on the (class of the) object (k need not be fixed; it is conceivable that new rights for new operation are added to an object during its lifetime). The semantics of these rights, i.e., which operations they allow, are of no concern here.

For each right, an Agroup is defined. We write $obj.r_i$ to denote the Agroup that has right r_i on object *obj*. The individual users that have these rights are given by $obj.r_i.members$ as defined in Section 3.1.

Agroups have no access rights for themselves. Accessing an Agroup amounts to accessing an attribute of some object, which is controlled by one of the rights on that object. This avoids a recursion of rights on rights, rights on rights on rights, etc. *Modification* of access rights is discussed in Section 3.5.

In Figure 3(a), part of the ACL of some imaginary object is displayed. It is a directed acyclic graph with the rights to the object (the graph's sources) at the top and the users (the graph's sinks) displayed left. Auxiliary nodes *a*, *b*, and *c* have been introduced to improve the structure of the graph. The Agroups corresponding to rights $r_1, \ldots, r_4$ have a common subgroup *a* which has a single subgroup *special task*. The same distribution of rights to users could have been obtained by adding *special task* directly as a subgroup to each of these four rights.

Figure 3(b) shows a conventional matrix-shaped ACL with a rights distribution that has the same effect. The essence of our model is that we have replaced this matrix by a graph.

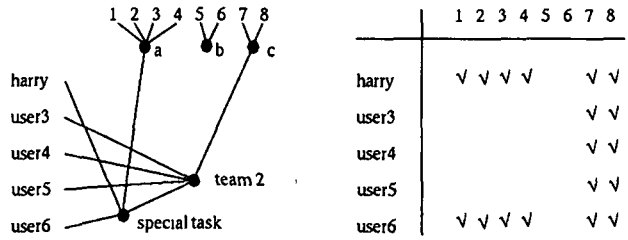

Figure 3: ACL in our model (a) and as a matrix (b)

3.4 Views

The groups *a*, *b*, and *c* in Figure 3 exemplify the notion of a *view* on an object: a subset of the rights defined for that object. Views can be used to organize large sets of access rights into managable proportions. An example: In the BSCW system, an object of type *folder* has 12 different rights: 5× *add* (for different types of objects), *delete* (from the folder), *cut* (i.e. relocate) the folder, *edit description*, *edit banner*, *get* (folder contents), get *info* about the folder, and *rename*. In order to simplify things in the user interface, we could partition these rights into 4 views:

- *read* (get, info),
- *add* (add any kind of object),
- *edit* (edit description, edit banner, rename), and
- *dispose* (delete, cut).

Views may overlap. We could, for example, add a view *annotate* (get, info, and *add article*, i.e., create a note to which others can reply).

The possibility to organize rights into views is a *consequence* of the model, not part of its definition. In the structure of Figure 3 we have simply created an Agroup *obj.a* and defined $obj.r_i = \{obj.a\}$ for $i = 1, 2, 3, 4$.

In a practical system we can employ views and their subgroups to show (and have the user interact with) access rights. Listing these against each other, we obtain a conventional —but much smaller—access rights matrix, see Figure 4. Of course the user should have additional means of finding out who are the members (subgroups) of a group and which are the rights gathered in a single view.

	a	*b*	*c*
team 2	√		√
special task			√

Figure 4: Simplified presentation of the access rights in Figure 3

3.5 Control

For purposes of system maintainance there must be some notion of *responsibility* for objects. E.g. if obsolete objects have to be deleted to free disk space, somebody should be told to do so. Also, it should not be possible that a user creates a "zombie" object that does not allow further access to any user, including its creator.

These problems have be dealt with by introducing some conventions into the model. There are various ways of doing this; the following is simple and general.

- *Responsible.* Every object *obj* is associated with a particular user who is responsible for it. We write *obj.responsible* to denote this user.
- *Control right.* Every object *obj* has an access right r_c, that allows any access on any of the access right groups $obj.r_1, \ldots, obj.r_k, obj.r_c$.
- *Control group.* The control right of an object *obj* is granted to members of the control group $obj.r_c$. In addition, the control right is also granted to *obj.responsible*, irrespective of his membership of $obj.r_c$.

If a group were to be responsible, the question remains which group member is accountable when something is wrong. To avoid that, we have laid responsibility with an individual. Some policy has to be laid down to transfer responsibility for objects when users are removed from the system.

4 Extensions to the basic model

We define negative rights (4.1), conditional authorization (4.2), and explicit role switching (4.3) as extensions to the basic model. Delegation (4.4) is supported by the model without further extensions.

4.1 Negative rights

A straightforward model of negative rights is used in the Andrew system (Satyanarayanan, 1989). Rights exist both in positive and negative form. When both apply, the negative right overrides the positive right. In this way, negative rights can be used to immediately revoke a permission in a distributed system where propagation of changes may take a while.

Shen and Dewan (1992) do not have an a priori bias for negative rights, but let other factors decide which, in case of conflict, is the more important. In principle this is a good idea, but it has dramatic consequences for the complexity of their model: several conflict resolution rules have to be introduced *solely* to solve ambiguities caused by positive and negative rights applying simultaneously.

We can keep the flexibility of Shen and Dewan's model but avoid its complexity, by introducing negation at another level. Rather than negative rights, we introduce negative group membership. *Excluding* somebody from a user group, as opposed to deleting him as a member, has a different modality, stating that "this person cannot

be member of the group." If another person tries to make him a member (e.g. by adding him to a subgroup), this is overruled by the exclusion.

As an example, consider again the group structure in Figure 1. Tom and Dick are organizing a surprise party for Harry. Team 2 will be involved at some stage, and perhaps some other people, but it is essential that Harry is not a member of the group of persons who know about the party. In Figure 5 a new user group has been created from which Harry is excluded (dotted line marked "X"). Whether Tom and Dick have realized that Harry is indirectly a member of team 2 is not relevant—even if he were not, currently, somebody could add him to (a subgroup of) team 2 any time. Hence specific exclusion, rather than non-inclusion, is needed.

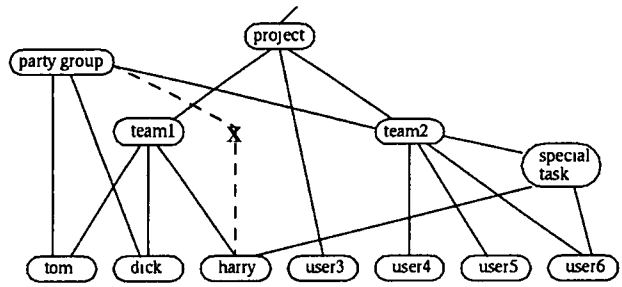

Figure 5: Exclusion from group membership

The model is therefore extended as follows. Each proper user group g has attributes *g.subgroups* and *g.excluded*; both are (possibly empty) sets of user groups. Members of a group are defined as the members of its subgroups, with the exception of those users that are also members of an excluded user group. In order to be precise, we define the following function *members* on user groups (for compatibility with the rest of the paper we do not use the conventional functional notation *members*(g) but the object-oriented notation *g.members*):

$u.members \stackrel{\text{def}}{=} u$ for users u, and

$g.members \stackrel{\text{def}}{=} \bigcup_{h \in g\ subgroups} h.members \setminus \bigcup_{h \in g\ excluded} h.members$

for proper user groups g.

Exclusions can be nested, an example will be given in Section 4.4.3.

As in Section 3 it is required that the group graph is acyclic. We have a labelled graph now, each edge carries a label "subgroup" or "excluded". The type of label is not relevant for the acyclicity constraint. Acyclicity of the graph guarantees that the recursion in the definition of *members* is finite.

We need two more operations to change the state of the system (in addition to those of 3.1): **AddExcluded** and **DeleteExcluded**, with the obvious semantics.

4.2 Conditions

So far we have assumed that the rights to access an object are stored with the object and do not depend in any way on the state of the system—or the state of the real world outside the system. "Context" can be handled by introducing *conditional access rights.*

Again we will adopt a more general solution and attach conditions to groups, rather than access rights. Every user group (Rgroup or Agroup) can be supplied with an attribute *condition*, a boolean function to be evaluated at the moment of access. If the condition evaluates to *false* the system behaves as if the group does not exist.

Some examples of conditions:

- *Only from a workstation within the building*, not over a telephone line or remote login.
- *Only in case of emergency.* E.g.: protected patient data in a hospital can be retrieved by any staff if there is an emergency. (Note that some protocol is required to decide what is an emergency. For example, the user may *declare* an emergency by answering the question "Is this an emergency?" with "yes"; emergency accesses are logged and the user is held accountable for such accesses.)
- *Only until March 31st* (for a temporary user group).
- *"Dynamic roles"*: Edwards (1996) proposes a similar scheme to dynamically include persons in a group by run-time evaluation of a function. In our model, one could grant access to "everybody" and then use a condition to prevent everybody from getting in all the time.

A design choice in our model, motivated by security considerations, is that conditions can only be used to *deny* access that, in different circumstances, would have been granted. The evaluation of a condition might call arbitrary code outside the realm of access control (as in Edwards' model). In order not to open the system to abitrary access, conditions should only allow access within clear limits defined in the access control component.

Indirect membership via a chain of subgroups may lead to a conjunction of conditions; membership via multiple chains of subgroups to a disjunction of conjunctions. When conditions have no side effects this is unproblematic. But if, for example, evaluation of a condition may require interaction with the user, a policy has to be specified that defines the order of evaluation.

4.3 Roles

Roles are modelled as user groups. A user can act in a number of different roles. Most of these are trivial (e.g., *harry* may act as a member of *team1* and as a member of *team2*) and the system should keep track of this. This is what we have modelled

in Section 3. There are, however, less trivial roles that should require a conscious act on behalf of the user in order to take on that role. A typical example is the role of system administrator. The person administering the system has, in principle, superuser capabilities. But in his regular work these should not be effective, so as to prevent accidents by unintended use. If the user wants to take on such a role, some action on behalf of the user is required.

In our model, a role is a user group that requires additional authentication. This could be anything from checking a box "I want to have role R" to authenticating yourself with a personal chip card. The important thing is that a role does not become effective without the user knowing that he is taking on the role. Roles (as any other groups) can be subject to conditions. For example, in order to take on the role *superuser*, it might be required that the user is working from (and not remotely logged in on) a workstation within the building.

Which roles should be tracked automatically and which roles should require explicit adoption of a role? An indication is the awareness information that one would like the system to generate for other users. If, for example *tom* has edited a document that belongs to the user group *project*, at some place you want to see a line

```
<doc> modified by tom
```

and you don't want to be bothered with the trivial information that *tom* is a member of *project* via *team1*. If, on the other hand, an obsolete object is removed by the system administrator, it is more appropriate to get a line such as

```
<obj> removed by tom as sysadm
```

stating which role licenced the operation (and who was acting in that role).

4.4 Delegation

Several delegation models have been proposed in the literature. We show how these are supported by the authorization model presented above.

The user should not have to perform the complicated group operations "manually," and it is obvious that additional delegation functionality in the user interface is required when delegation of rights is to be included in an application. The purpose of this section, however, is to make clear that the authorization model needs no further extensions to support delegation.

For the sake of simplicity we assume that the responsible is the only person with control right and call him, more conventionally, the *owner*. (Joint ownership does not affect any of the following.) As a convenient shorthand we write g_u to express that user u is the owner of a user group g.

4.4.1 Single-step delegation

A user u has been told to do something with an object. To that end, the owner of *obj* has constructed a view

$$obj.view = \{\{u\}_u, \ldots\}.$$

The view has a subgroup that contains u and is owned by u. However, u wants to delegate the job to v.

In the simplest case, the delegate can perform the task, but may not further delegate it to somebody else. To this end u extends $\{u\}_u$ to $\{u, v\}_u$, yielding

$$obj.view = \{\{u, v\}_u, \ldots\}.$$

This gives user v access to the view, but he cannot change the user group to which the view is granted. Moreover, u can revoke the delegation by deleting v again.

4.4.2 Recursive delegation

In other scenarios a delegate may further delegate a right. For example in the work flow in a German ministry (Mambrey and Robinson, 1995): an object moves down several layers of hierarchy, is processed by somebody, and then moves up the hierarchy in reverse direction. Starting from the same situation as above, user u now adds a user group that contains v and is owned by v:

$$obj.view = \{\{u, \{v\}_v\}_u, \ldots\}.$$

Hence v can delegate the view to w in similar fashion. Any person in such a chain of delegations may revoke the delegations he made, including further delegations made by his delegates.

4.4.3 Delegation within a trusted group

Coulouris and Dollimore (1994a) present a case study of examination preparation at a university college. A task (viz., typing of the exams) can only be delegated within a predefined group of trusted persons. In order to model this, we assume negative rights (see 4.1). We write

$$g = \{\ldots, \neg x\}$$

to denote that x is excluded from user group g. Further, we denote "everybody in the system" by a special constant E.

Let t be the group of trusted persons. Then we define "nontrusted persons" as "everybody in E who is not in t", denoted

$$n = \{E, \neg t\}.$$

Having an expression for nontrusted persons, we can now define groups from which "*nontrusted persons are excluded.*" Thus the owner of object *obj* creates a view

$$obj.view = \{\{u\}_u, \neg n\} = \{\{u\}_u, \neg\{E, \neg t\}\}$$

(assuming u is member of t). User u may delegate his right in one of the ways explained above. Delegation (whether single-step or recursive) to a person who is not a member of t is overruled by the exclusion of nontrusted persons; see (Sikkel, 1997) for a more detailed treatment.

5 Realization in BSCW

An implementation of the presented authorization model in the BSCW system ("Basic Support for Cooperative Work") is currently under development. A public release of a BSCW version offering full access control is due Autumn 1997. BSCW (Bentley et al, 1995; Bentley et al, 1997) is based on the notion of a "shared workspace." A workspace is a repository for shared information, accessible (only) to group members. Workspaces may contain different types of objects (documents, folders, threaded discussions, etc.). It offers some awareness facilities: one can see what other members in the workspace have been doing; soft locking can be used to prevent simultaneous editing by different users. Support for synchronous cooperation is being integrated (Trevor and Koch, 1997).

The BSCW server is realized as an auxiliary component to a WWW server. All interaction takes place via the Common Gateway Interface (CGI) hence the BSCW server is not dependent on any particular WWW server. BSCW runs on must UNIX variants and on Windows NT. A public server is available free of charge at GMD.[2] The user interacts with a shared workspace using an ordinary WWW Browser. One of the advantages of the system—and probably one of the reasons for its success—is that a BSCW user can join a working group without any prior installation of software.

Authentication in BSCW employs the Web's "basic authentication" scheme, the only Web-wide standard. Users authenticate with user name and password.

A prime consideration in the implementation of access control is efficiency. In the context of BSCW, the following queries to the access control component have to be efficient:

(1) Has user u right r_i on object *obj*?

(2) Which rights does user u have on object *obj*?

(3) Which users have right r_i on object *obj*?

In order to address all of these, the computation of $obj.r_i.members$ has to take minimal effort. To that end, a list of group members is stored with the user group. It probably suffices to maintain membership lists for Rgroups, because that is where most of the nesting is to be expected. When the composition of a user group is changed, the membership lists of its supergroups are to be adapted accordingly.

Our main concern for the near future is in designing and testing a user interface that our users find easy to work with.

6 Discussion and conclusions

In this paper we have investigated the foundations of access control in groupware. We have drawn up an authorization model that addresses many of the issues in

[2] `http://bscw.gmd.de`

access control.

The treatment of negative rights is as general as in (Shen and Dewan, 1992). It is simpler, however, because no conflict resolution rules are needed. Further, we have shown how various models of delegation, drawn from case studies in cooperative work, are supported by the general authorization model.

Not addressed in this paper is the variety of different rights needed in the context of groupware systems (see (Dewan et al., 1994) for an elaboration in the context of joint editing) and, more prominently, a user interface that supports the appropriate functionality and conveys a suitable mental model. An implementation of access control in the BSCW system based on this authorization model is, at the time of writing, still under construction.

Given the inherent complexity of the matter, we have stated simplicity as an important goal. The presented model is *minimal*: each concept that has been introduced is evidently needed to satisfy legitimate practical requirements. In addition, the model is *modular*: extensions not needed in a particular application can be discarded, yielding a simpler model.

Acknowledgements

I am grateful to Uwe Busbach, Wolfgang Appelt, Jonathan Trevor, David Kerr, Gerd Woetzel, Richard Bentley and two anonymous referees for discussions and feedback on previous drafts.
This work was partially supported by the Ministry of Science and Research of North Rhine-Westphalia.

References

Bentley, R., Appelt W., Busbach U., Hinrichts E., Kerr, D., Sikkel, K., Trevor, J. and Woetzel, G. (1997): "Basic Support for Cooperative Work on the World Wide Web." *International Journal of Human-Computer Interaction*, special issue on novel applications of the World Wide Web, 1997 (in press).

Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. (1995): "Supporting collaborative information sharing with the World Wide Web: The BSCW Shared Workspace system." *4th International WWW Conference*, Boston, December 1995, pp. 63–74.

Coulouris, G. and Dollimore, J. (1994a): "Requirements for security in cooperative work: two case studies." Technical Report 671, Dept. of Computer Science, Queen Mary and Westfield College, University of London.

Coulouris, G. and Dollimore, J. (1994b): "A security model for cooperative work." Technical Report 674, Dept. of Computer Science, Queen Mary and Westfield College, University of London.

Dewan, P., Choudhary, R. and Shen, H. (1994): "An Editing-Based Characterization of the Design Space of Collaborative Applications." *Journal of Organizational Computing*,

Vol. 4, pp. 219–239.

Edwards, W.K. (1996): "Policies and Roles in Collaborative Applications." *ACM Conference on Computer-Supported Cooperative Work (CSCW'96)*, Cambridge, Mass., pp. 11–20.

Ellis, C.A., Gibbs, S.J. and Rein, G.L. (1991): Groupware: Some Issues and Experiences. *Communications of the ACM*, Vol. 34, No. 1, January 1991, pp. 38–58.

Greif, I. and Sarin, S. (1986): Data Sharing in Group Work. *ACM Conference on Computer-Supported Cooperative Work*, Austin, Texas, 1986.
An extended version appeared in *ACM Transactions on Office Information Systems*, Vol. 5 (1987), pp. 187–211.

Härtig, H., Kowalski, O., Kühnhauser, W. (1993): "The BirliX Security Architecture." *Journal of Computer Security*, Vol. 2, pp. 5–21.

Kanawati, R. and Riveill M. (1995): "Access Control Model for Groupware Applications." In Allen, G., Wilkinson, J., and Wright, P. (Eds), HCI'95: People and Computers. School of Computing and Mathematics, University of Huddersfield, UK, pp. 66–71.

Lampson, B.W. (1974): Protection. *ACM Operating Systems Review*, Vol. 8, pp. 18–24.

Lampson, B., Abadi, M., Burrows, M. and Wobber E. (1992): "Authentication in Distributed Systems: Theory and Practice." *ACM Transactions on Computer Systems*, Vol. 10, No. 4.

Mambrey, P. and Robinson, M (1995): "Preparing a speech for the minister: Notes towards understanding the role of artefacts in a flow of work." Unpublished manuscript, GMD-FIT, Sankt Augustin, Germany.

Paterson, J.F., Hill, R.D., Rohall, S.L. and Meeks, W.S. (1990): "Rendezvous: An Architecture for Synchronous Multi-User Applications." *ACM Conference on Computer-Supported Cooperative Work (CSCW'90)*, pp. 317–328.

Sandhu, R.S., Coyne, E.J., Feinstein, H.L. and Yourman, C.E. (1996): "Role-Based Access Control Models." *IEEE Computer*, February 1996, pp. 38–47.

Salzer, J.H. (1974): "Protection and Control of Information Sharing in Multics." *Communications of the ACM*, Vol. 17, pp. 388–402.

Satyanarayanan, M. (1989): "Intergrating Security in a Large Distributed System." *ACM Transactions on Computer Systems*, Vol. 7, pp. 247–280.

Sikkel, K. (1997): "A Group-based Authorization Model for Computer-Supported Cooperative Work." Arbeitspapiere der GMD 1055, GMD, Sankt Augustin, Germany.

Shen, H. and Dewan P. (1992): "Access Control for Collaborative Environments." *ACM Conference on Computer-Supported Cooperative Work (CSCW'92)*, Toronto, Canada, pp. 51–58.

Trevor, J., Koch, T. and Woetzel, G. (1997): "MetaWeb: Bridging the Gap between Synchronous Groupware and the WWW." *European Conference on Computer-Supported Cooperative Work (ECSCW'97)*, Lancaster, UK (these proceedings).

Trevor, J., Rodden, T., and Mariani, J. (1994): "The Use of Adapters to Support Cooperative Sharing." *ACM Conference on Computer-Supported Cooperative Work (CSCW'94)*, Chapel Hill, North Carolina, pp. 219–230.