

Arbeitspapier der GMD 1055, March 1997

A Group-based Authorization Model for Computer-Supported Cooperative Work

Klaas Sikkel

GMD – Forschungszentrum Informationstechnik GmbH
Institut für angewandte Informationstechnik (GMD-FIT)
D-53754 Sankt Augustin
sikkel@gmd.de

Abstract

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. Several extensions to the basic model address well-known issues in access control, notably negative rights and delegation.

Implementation of the authorization model in the *Basic Support for Cooperative Work* Shared Workspace system is envisaged.

Zusammenfassung

In der Fachliteratur werden oft Anforderungen an der Zugriffskontrolle bei computergestützte Zusammenarbeit aufgelistet. Die in der Praxis verwendeten Systeme kommen diesen Anforderungen jedoch nicht nach. Teilweise gibt es praktische Gründe dafür, es liegt aber auch an der inhärente Komplexität entsprechender Zugriffsrechtsmodelle.

Wir schlagen ein allgemeines Modell für Zugriffsberechtigung vor, das möglichst einfach gehalten ist. Einige Erweiterungen des Basismodells ergeben Lösungen für bekannte Probleme auf diesem Gebiet, insbesondere Negativrechte und Delegation von Rechten.

Eine Implementierung dieses Zugriffsrechtsmodells im Rahmen des Systems „Basic Support for Cooperative Work“ zur Unterstützung gemeinsamer Arbeitsbereiche ist vorgesehen.

Contents

1	Introduction	7
1.1	Access Control and Groupware	7
1.2	The BSCW Shared Workspace server	8
1.3	The structure of this paper	8
2	Issues in authorization and groupware	9
3	A minimal authorization model	10
3.1	Groups	10
3.2	User groups	11
3.3	Dynamics of the group model	13
3.4	Objects and user groups	15
3.5	Access rights	16
3.6	Views	17
3.7	Control	18
3.8	Defaults	19
4	Negative access rights	20
4.1	A motivating example	21
4.2	The static group model	21
4.3	Dynamics of the model	22
4.4	A note on the user interface	23
5	Delegation	23
5.1	Single-step delegation	24
5.2	Recursive delegation	24
5.3	Delegation within a trusted group	25
5.4	Implementation considerations	25
5.5	Transfer of responsibility	25
6	Further extensions to the model	26
6.1	Conditions	26
6.2	Roles versus Groups	27
6.3	Object structures	29
6.3.1	Rights on a combination of objects	29
6.3.2	Indirection of access rights	29
6.3.3	Variables	30
6.3.4	Inheritance of rights	30
6.4	System administration	31
7	Realization in BSCW	31
7.1	Removing objects	31
7.2	Defaults	32
7.3	Efficiency considerations	33
7.4	User interface considerations	34

7.4.1	Changing access rights	34
7.4.2	Visibility	34
8	Discussion and conclusions	34
	Acknowledgements	35
	References	36

1 Introduction

There is a general distinction between *authentication* (verifying that you are who you pretend to be) and *authorization* (what are you allowed to do). There is also a distinction between *access rights* (here used as a synonym of authorization) and *access control*, viz., ensuring that the rights are not violated. This paper proposes a simple but powerful model for authorization in groupware.

The work has been carried out in the context of the project *Basic Support for Cooperative Work* (BSCW) [BHST95, Sik95, B&a197]. Although the work has been motivated by the immediate needs of the BSCW Shared Workspace system, the authorization model presented here is of a general nature. We address several issues in groupware and authorization—in particular negative rights and delegation—at a fundamental level and propose general, simple solutions.

In order to make the model reusable, we distinguish between a basic model and several possible extensions to the basic model. Moreover, we discuss and motivate the design decisions taken in shaping the model. Issues related to realization in the context of the BSCW system are deferred to a separate section.

1.1 Access Control and Groupware

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 [GS86] 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 [SD92]; 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 [EGR91, SD92], access control models for groupware tend to be rather complex. The challenge is to hide much of the complexity in the user interface, providing the user with an adequate set of access control operations that allows easy specification of changes and is easy to understand. Designing such an interface is far from trivial.

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.” Typically, a prototype should prove the feasibility of a particular technique. If the technical infrastructure works, but there is only rudimentary access control, the system can be employed in first tests. In contrast, a full-fledged implementation of access control for a system that does not yet run does not allow practical testing. Hence the natural tendency *not* to make access rights a priority for your new prototype.

This problem is nicely illustrated with the development of our own system, BSCW, offering shared workspaces on the World Wide Web. Thousands of users accept its short-

comings 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 shows that there is a need for more powerful and easy to use access control. Hence—more than a year after the system's first public release—we designed a suitable access control model.

1.2 The BSCW Shared Workspace server

BSCW 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 [TK97].

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 most UNIX variants and on Windows NT. A public server is available free of charge at GMD.¹ The user interacts with a shared workspace using an ordinary WWW Browser. The only additional software that a user may want to (but does not have to) install is a “helper” for uploading documents, offering an interface that is more convenient than can be realized with the built-in upload functionality of popular browsers. 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. For more details, see [BHST95, B&a197].

Authentication in BSCW employs the Web's “basic authentication” scheme, the only Web-wide standard. Users authenticate with user name and password. Authorization is dealt with by the BSCW system.

In Version 1, a simple authorization scheme was hard-wired into the system. All members of a workspace have equal rights on every object, with one exception: removing an object from the system can only be done by the user who created the object. BSCW Version 2.0 offered access control along the same lines as default values, with the possibility for owners of an object to change the access rights. An overhaul of the access control model is envisaged for BSCW Version 2.3, due in the course of 1997.

1.3 The structure of this paper

Section 2 reviews some issues involved in access rights for cooperative systems. A minimal form of the general, abstract authorization model is introduced in Section 3. Negative rights and delegation of authorization are discussed in Sections 4 and 5, respectively. Some further extensions to the authorization model are more briefly discussed in Section 6. Some issues related to the realization of the authorization model in the context of the BSCW Shared Workspace system are highlighted in Section 7. Discussion and conclusions follow in Section 8.

¹<http://bscw.gmd.de>

2 Issues in authorization and groupware

The following issues are commonly mentioned in relation with access rights for collaborative 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 [Sal74]. An operating system defines access rights on the level of OS operations, but access rights should relate to the operations available to the user [GS87]. Synchronous groupware, in addition, may offer various levels of object sharing [PHRM90].
- *Flexibility and ease of use.* Access rights in groupware may depend on who is doing what and therefore are highly dynamic [TRM94]. Edwards [Edw96] 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. In [SD92, DCS94, KR95] it is argued that a user's permission should be the sum of the permissions of his roles. In a more security-oriented approach, e.g. [LABW92, CD94b], 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 possibility to revoke delegated rights [CD94a, CD94b, KR95].
- *Negative rights* are often stated as a requirement and present in several systems and models [Sal74, Sat89, RBKW91, SD92, HKK93]. 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 [SD92] state that authorization models satisfying these requirements are necessarily rather complex, but in Section 4 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 Razor² has been used as a main design principle.

In Section 3 we present the basic model. It extends the the canonical authorization model with a *single* concept: hierarchical group structures. Subsequently, in Sections 4–6,

²“*Entia non sunt multiplicanda praeter necessitatem.*” (Entities ought not to be multiplied except out of necessity.) – William of Ockham (1285-1349)

we define several orthogonal extensions to the basic model. 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 minimal 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 the *Lampson matrix* [Lam74], 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.

The authorization model presented in this section does not focus on BSCW but addresses, in the abstract, the ontological model in the general groupware framework of [EW94]. The 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.

We provide some simple structuring of the subject and rights dimensions. Because, in the abstract, it is unclear how the objects in a groupware system are related to one another, we hardly discuss structuring of objects.

3.1 Groups

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 organization hierarchy. A person is employed in a department and currently working on some project as member of one of several project teams. The team, the project, the department, etc., are groups that he is a member of. Related to groups are *roles*. Within the project, there could be roles like designer, programmer, system administrator, quality assurance manager, project leader, etc. Several persons can act in the same role, and a single person may have different roles. The notions “group” and “role” are closely linked. A role can be seen as a specification of the group of persons that may act in this role. Which should be called “group” and which “role” is often a matter of taste and not relevant for the access rights model. In Section 6.2 we will differentiate between roles and groups. For the moment we consider them synonyms and use “group” to denote both.³

Another type of group could be called *access group*. Suppose you are collaboratively working on a paper. You may distinguish between “authors”, “annotators” (proof-readers giving comments) and “readers” of the 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. Access groups involve two different notions of grouping:

³In the literature often “role” is used to describe both roles and groups. We use “group” because it fits better to the set-theoretical approach of our model.

- A set of rights is grouped⁴ into what is usually called a *view*.
- This view is granted to a group of users.

3.2 User groups

If we only consider user groups, and don't think about access rights to objects, the model is truly simple.

User. We assume the existence of a domain of users, containing, for example, *tom*, *dick*, *harry*, etc.; or, more abstractly, u, v, \dots . At each moment a particular set of users U exists in the system (but obviously the set of users may change over time).

User group. We define user groups recursively 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. We write G for the set of groups existing at some particular moment.

For convenience, we have defined single users to be user groups as well (i.e., $U \subseteq G$). In the remainder of the paper we can refer to “a user group” meaning “an individual user or a group of users.” Only in rare cases we need to refer to a composite group, not an individual user. To that end we define

Proper user group. A user group g is called proper if $g \in G \setminus U$.

User groups form hierarchical structures.

Subgroup. A group g is called a subgroup of group h if $g \in h$.

Supergroup. A group g is called a supergroup of group h if $h \in g$.

The relation $<$ on $G \times G$ is the transitive closure⁵ of the subgroup relation. The relation $>$ on $G \times G$ is the transitive closure of the supergroup relation. We write \leq and \geq , respectively, for the transitive and reflexive closures.

Above we stated that a group may not be “contained in” itself. Using $<$ we can formulate this more precisely:

$g < g$ does *not* hold for any $g \in G$.

The reader may verify that $<$ ($>$) is a partial order.

The subgroup relation defines a directed acyclic graph (henceforth called “the group 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.

Sometimes it is convenient to use graph terminology rather than set terminology, e.g., a group g is a *descendant* of h if $g < h$.

⁴The technical details are slightly different, however; see Section 3.6.

⁵That is, $g < h$ if $g \in h$ or $g \in \dots \in h$.

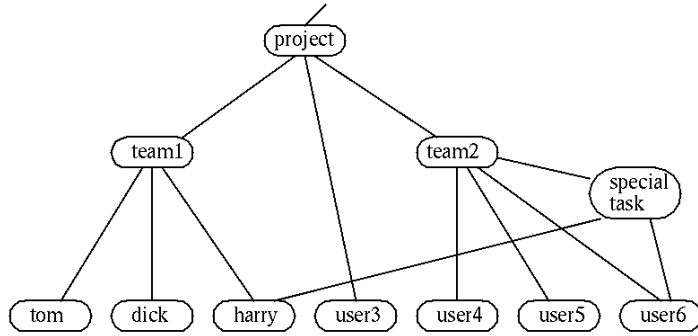


Figure 1: A hierarchical group structure

Members. For every group $g \in G$ we define a set of members

$$g.\text{members} = \{u \in U \mid u \leq g\}.$$

That is, members of a user group are the descendant leaves in the group graph. A user u has itself as a single member.

Throughout this paper we use the object-oriented notation $obj.f$ to denote a function applied to an object (equivalent to $f(obj)$ in functional notation). In mathematical terms we have a function $members$ mapping a user group onto a set of users.

For convenience, we also write $g.subgroups$ for the subgroups of a group g ⁶

Examples, drawn from Figure 1:

$$\begin{aligned} project.subgroups &= \{team1, team2, user3\}, \\ project.members &= \{tom, dick, harry, user3, user4, user5, user6\}, \\ team2.subgroups &= \{user4, user5, user6, special-task\}. \\ team2.members &= \{user4, user5, user6, harry\}. \end{aligned}$$

A notation for group structures

Group structures can be displayed as graphs, but for some purposes it is convenient to have a more compact, linear notation. Let $g.subgroups = \{h, k, j\}$ and $h.subgroups = \{u, v\}$. Then we may write each of the following to denote g :

$$\begin{aligned} g &= \{h, k, j\}, \\ g &= \{\{u, v\}, k, j\}, \\ g &= \{h = \{u, v\}, k, j\}. \end{aligned}$$

⁶Note that, according the formal model, the notation $g.subgroups$ is redundant. A group is defined as a set of subgroups. Hence g and $g.subgroups$ denote the same object, viz, the set that contains the subgroups of g as elements. However, in Section 4 the definition of group composition is extended and this equivalence holds no longer.

3.3 Dynamics of the group model

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 vertex*].

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 [*fails if a cycle would be created*].

DeleteSubgroups deletes subgroups from a user group [*removes edges*].

RemoveGroup removes a user group from the system [*removes a vertex 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 vertex; incident edges are replaced, linking each predecessor directly to each successor*].

More precisely: Let $g = \{h_1, \dots, h_j\}$ and $f = \{g, g', g'', \dots\}$ a supergroup of g . Dissolving g changes f to $f = \{h_1, \dots, h_j, g', g'', \dots\}$.

RenameGroup changes the name of a user group [*changes the label of a vertex*].

Note that RemoveGroup is not a primitive operation. It can be composed from DeleteSubgroups and DissolveGroup, hence it can be discarded. We have included it, however, in order to point out the difference between removing and dissolving. Whether a group should be removed or dissolved depends on the circumstances.

Consider the group *special-task* in Figure 1. All its members are member of some project team. Presumably, *harry* is a member of *team2* only for the duration of this special task. When the task is finished, there is no need to include him as a regular member in *team2*. Hence, *special-task* is removed (not dissolved); see Figure 2.

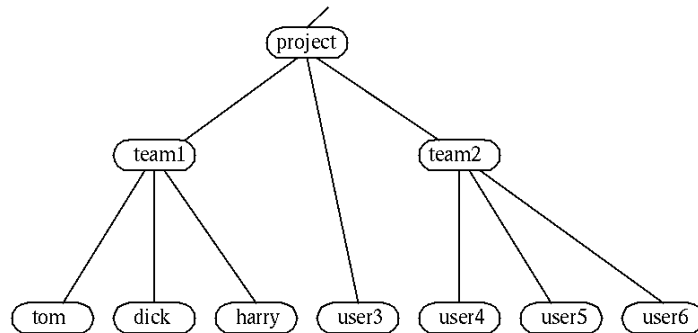


Figure 2: *special* group in Figure 1 has been removed.

On the other hand, consider the case that the work of *team2* is finished and the team can be deleted from the system. This does not mean that the employees that were involved in the team are sacked. *Team2* was a subgroup of the project group and the members of the team will remain to be members of the project group even after their team has ceased to exist. Hence *team2* is dissolved (not removed); see Figure 3.

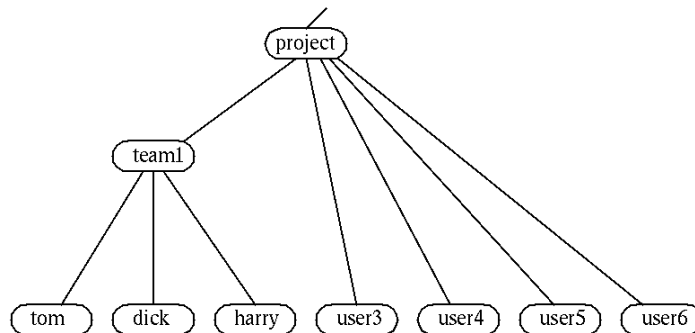


Figure 3: *team2* in Figure 2 has been dissolved.

Additional composite operations on user groups can be added as required. For example:

InsertGroup is, in a way, the inverse of DissolveGroup.

[*split a vertex in an ‘upper’ and a ‘lower’ vertex, as follows:*

- *incoming edges go to the upper vertex;*
- *outgoing edges leave from the lower vertex;*
- *connect the upper to the lower vertex with a single new edge add a single new edge from the upper to the lower vertex.]*

InsertGroup does not change membership of any group, other than the newly created one. The introduction of this operation follows from mathematical observations on the group graph. InsertGroup is more than a mathematical nicety, however. It could be quite useful for the maintenance of group structures. Consider the following example: A student comes as trainee the project. The best way to structure the project, as it turns out now, would have been the following:

$$project = \{project-staff = \{team1, \dots\}, project-students\}.$$

But, as often, the current situation has not been anticipated when the original group structure was drawn up. With InsertGroup this can easily be repaired. The group *project-staff* is inserted between *project* and its subgroups. This is illustrated in Figure 4. Subsequently a new group *project-students* can be created and made subgroup of *project*.

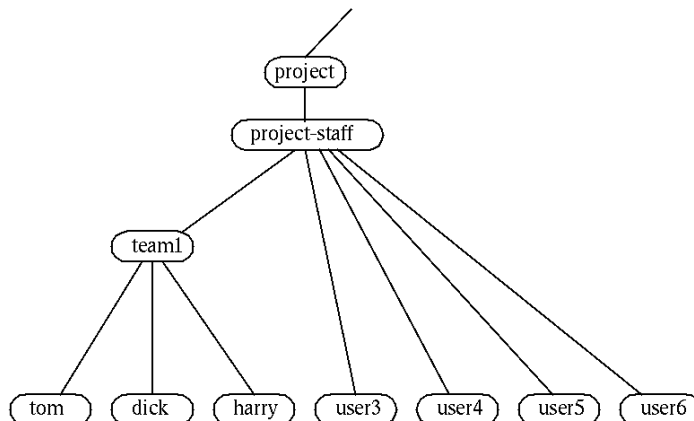


Figure 4: *Project-staff* has been inserted into Figure 3.

3.4 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 [SCFY96]. 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.

Rgroups (*regular user groups*) are regular objects.

Agroups (*attributed user groups*) exist only as attributes of regular objects. When a regular object is deleted from the database, its associated Agroups are removed.

Agrouns are accessed as attributes of the objects to which they are belong. We write, e.g., *obj.authors*, *obj.owners* for Agrouns of an object *obj*.

Usage

Individual users are Rgroups, proper user groups can be of either type. Rgroups can be subgroups of Agrouns and vice versa. The group properties defined above (having members, subgroups, etc.) are not related to the distinction between Rgroups and Agrouns. Some examples:

- *obj1.authors* = {*team1*};
an Agroun is composed of Rgroups (in this case a single one).
- *obj1.readers* = {*obj.authors*, *special-task*};
an Agroun contains another Agroun and an Rgroup.
- *obj1.readers* = {*obj1.authors*, *obj2.readers*};
an Agroun contains, among others, an Agroun of another object.
- *joint-group* = {*workspace1.members*, *workspace2.members*};
an Rgroup is composed of Agrouns.

Discussion

When an object is removed from the system, should its Agrouns be removed or dissolved? On the one hand, it is hard to give an answer that suits all circumstances. It depends, of course, with which intention the Agroun has been added to a supergroup. On the other hand, the system should clearly support a single mechanism—at the expense that users who need a different mechanism have to do some more work.

In the abstract, if one includes an Agroun *obj.a* into some user group *g*, it seems most reasonable to expect that the member of *obj.a* are no longer members of *g* when *obj* no longer exists. If it has to be different, one can insert an Rgroup *a'*, (making *obj.a* = {*a'*}) and add *a'* as a subgroup to *g*.

Hence, Agrouns are removed, not dissolved.

3.5 Access rights

For each regular object we assume a set of *access types* $a_1 \dots a_k$ where the number of types *k* depends on the (class of the) object. The semantics of the access types (i.e. which operations on the object they allow) are of no concern here.

For each type an Agroun is defined. The members of this a group have the right to perform actions on the object that correspond to the access type. We write *obj.a_i* to denote the Agroun that has the right to perform accesses of type *a_i* on object *obj*. The individual users that have this right are given by *obj.a_i.members* as defined in Section 3.1.

Authorization is usually described by means of a function *f* yielding a boolean value for each combination of subject, object, and access type. A subject has the *right* to perform an access of type *a* on object *o* if $f(s, o, a) = true$ [GD72, Lam74, FSW81].

In Section 3.1 we have postulated a set of users U . In addition we assume a set O of objects and a set A of access types. For any object $obj \in O$ and for any access type a_i defined on obj it should hold that $a_i \in A$. The authorization function

$$f : U \times O \times A \longrightarrow \{true, false\}$$

is defined by

$$f(u, obj, a) = \begin{cases} true & \text{if } u \in obj.a.members \\ false & \text{if } u \notin obj.a.members \end{cases}$$

In a sloppy but convenient terminology, we use “access right” and “access type” to denote the same thing. Typically we say “user u has right a_i on object obj ”, rather than the formally correct “user u has the right to perform an access of type a_i on object obj .”

Access rights are realized by Agroups. Access types must exist for *changing* access rights, that is, modifying the composition of such an Agroup. Note, however, that Agroups have no access types for themselves. Authorization to access Agroups is regulated by rights on the object to which the Agroup is attributed. This avoids a recursion of access rights on access rights, access rights on access rights on access rights, and so on.

Modification of access rights is discussed in more detail in section 3.7.

3.6 Views

A *view* is a subset of the set of access types defined for a particular (class of) object [Hag94, CD94b]. There is a similarity with views on databases. Views can be used to organize large sets of access rights into manageable proportions. An example: In the BSCW system, an object of type *folder* has 12 different rights: *add_article*, *add_document*, *add_folder*, *add_URL*, *add_versions*, *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),
- *modify* (add/delete any kind of object),
- *edit* (edit_description, edit_banner, rename), and
- *relocate* (cut).

Views may overlap. We could, for example, add another view

- *annotate* (get, info, and *add article*, i.e., create a note to which others can reply).

Suppose that *team2* and *harry* (in Figure 1) should get *annotate* right on for a folder *f1*. We could, in principle define $f1.get = \{team2, harry\}$, $f1.info = \{team2, harry\}$, and $f1.add_article = \{team2, harry\}$. A more general approach, however, is the following. For

each view on an object we create a special “view group” as an Agroup attributed to that object. In this case:

$$f1.annotate = \{team2, harry\}.$$

Then we make $f1.annotate$ a subgroup of the appropriate groups.

An ACL for the folder object $f1$ is defined by

$$\begin{aligned} f1.get &= \{f1.read, f1.annotate\}, \\ f1.info &= \{f1.read, f1.annotate\}, \\ f1.add_article &= \{f1.add, f1.annotate\}, \\ f1.add_document &= \{f1.add\}, \\ f1.add_folder &= \{f1.add\}, \\ \text{etc.} & \end{aligned}$$

with the view groups composed as appropriate.

Finally, it should be noted that the possibility to organize rights into views is an *application* of the model defined so far, not an extension of its definition. All concepts we have employed for the creation of views (other than the notion “view” itself) had already been introduced in previous sections.

A note on the user interface

In a practical system one can show (and have the user interact with) a matrix that lists user groups against views, as in Figure 5. The table entries are (check boxes representing) boolean values. From the model it is evident that a user has a view if and only if he is member of at least one of the user groups to which this view has been granted.

	view 1	view 2	...	view n
group 1	✓	✓		
group 2		✓		
...				
group m				✓

Figure 5: Suggested presentation of access rights

The matrix is only a convention for presentation. Formally, the access control list of an object obj is the ordered list of Agroups $obj.a_1, \dots, obj.a_k$.

3.7 Control

For purposes of system maintenance 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 to 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 a_c , that allows any access on any of the access right groups $obj.a_1, \dots, obj.a_k, obj.a_c$.
- *Control group*. The control right of an object *obj* is granted to members of the control group $obj.a_c$. In addition, the control right is also granted to *obj.responsible*, irrespective of his membership of $obj.a_c$.

Typically, the creator of an object is the responsible. We write $obj.responsible = u$ (and not $obj.responsible = \{u\}$); the responsible is an attribute, not an Agroup of the object. This means that transfer of responsibility is *not* governed by the rules for accessing Agroups. A protocol for transfer of responsibility has to be specified in a more concrete refinement of the general model. We discuss this in more detail in Section 5.5.

Discussion

Note that control right allows unrestricted access on the access right groups $obj.a_i$. It does *not*, by definition, allow unrestricted access on the object. Of course, a person in control can change the access rights so that he can do anything with the object. But there are situations where you want to prevent yourself from doing unwanted actions by mistake. For example, you don't want to be able to overwrite an object that should not be changed. (Compare access mode `r-xr-xr-x` in UNIX; the owner can always change this to `rwxr-xr-x`.)

It is debatable whether the responsible should be a single user, or an arbitrary group. The current (Version 2.0) BSCW model is somewhat confused about this; the owner group has a “prime owner,” but the existence of this concept is hidden in the user interface. Either shared ownership is truly shared, and there is no difference between the owners, or else the differences in rights and responsibilities between different users have to be made visible at the user interface.

3.8 Defaults

An issue that is hard to address in general—but important in the realization of practical systems—is how the access rights for newly created objects are determined. Every created object gets some default rights, that can be adapted by the creator, if necessary. There are various ways to determine such defaults:

- *Class*. An object class should supply a newly created object with suitable defaults.
- *User profile* (“`umask`” in UNIX). The rights to new objects depend on the user who creates them. This can be generalized to a group profile.
- *Location*. The environment, e.g., the container in which the new object is being placed, determines the access rights given to the object.

In a minimal model, the defaults should depend only on the object's class. In a more sophisticated model, the user profile and/or location can be used to override the defaults of the object class.

4 Negative access rights

It is often argued that it should be possible to specifically deny a right to some person or group. There are situations in which it is more important to know that some persons have *no* access than to know precisely who are the persons having access the object. Also, if somebody has access to a large collection of objects but should be excluded from a single object in this collection, it has pragmatic advantages if this exclusion can be noted somewhere in the system.

How to define negative rights? A simple, straightforward solution is to add a special access type *excluded* to each object, which overrides rights granted to other access types. In our terminology: right a_i is granted to those members of $obj.a_i$ who are *not* members of $obj.a_{excluded}$. This implies, however, that a user or group can be excluded only from *all* types of access, not from a particular subset of access types.

A straightforward model of negative rights is used in the Andrew system [Sat89]. For each access type, a positive and a negative right exists. 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.

Rabitti et al. [RBKW91] discuss an advanced authorization model for databases. They distinguish positive/negative and weak/strong rights for each access type. This is used to define implicit rights inherited via a hierarchical structure on any of the authorization dimensions. Weak rights can be overruled by weak or strong rights at a lower level in the hierarchy, strong rights cannot be overruled. Conflicts between negative and positive authorization are banned by imposing consistency constraints.

Shen and Dewan [SD92] have defined one of the more sophisticated access rights models for cooperative work. Following [RBKW91], they define a positive and a negative right for each access type, but without the (in view of flexibility rather unsuitable) consistency constraints. This has dramatic consequences for the complexity of their model, however. After having introduced inheritance of rights in various ways, a handful of conflict resolution rules are needed to solve ambiguities caused by interference of positive and negative rights. It is hard to give a proper formal semantics of their model.

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 on access types, we introduce *negative group membership*.

Excluding somebody from a group, as opposed to deleting him as a member, has a different modality, stating that "this person *cannot* be member of the group." If somebody inadvertently adds him to the group or to a subgroup, his membership is overruled by the exclusion. This is similar to the exclusion principle of [Sat89], but much more powerful because it applies to groups rather than access types.

Without introducing modal logic into the model (which would indeed complicate matters a lot), we can do this quite satisfactorily within graph theory. The directed acyclic graph that comprises the group structure of the current state of the system has *labelled* edges. Edges labelled *subgroup* represent "positive" inclusion, edges labelled *excluded* represent "negative" inclusion in a group. "negative" membership overrules positive membership. The precise semantics follow from the formal definition presented Section 4.2.

4.1 A motivating example

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 6 a new group has been created from which Harry is specifically excluded (dotted line marked “X”).

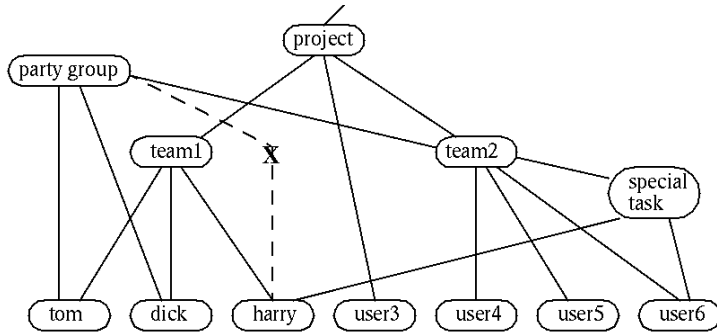


Figure 6: exclusion from group membership

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. Hence specific exclusion, rather than non-inclusion, is needed to cope with such situations.

Negative rights are probably only used in exceptional cases. But in order to be general, the model should allow arbitrary combinations of negative rights. In order to be precise about its meaning, we give a formal definition.

4.2 The static group model

In Section 3.4 we differentiated between Rgroups and Agroups. The difference between these kinds of groups is related only to (in)direct access and persistence. It has no bearing on group composition. Hence, when we introduce negation in group composition, we do not have to make a distinction between Rgroups and Agroups. As in Section 3.1 we postulate a set of users U and a set of user groups G with $U \subseteq G$, and $G \setminus U$ the set of proper user groups.

Let $\wp(X)$ denotes the powerset (i.e. the set of subsets) of X . We define functions

$$\begin{aligned} \text{subgroups} & : G \setminus U \rightarrow \wp(G) \\ \text{excluded} & : G \setminus U \rightarrow \wp(G) \end{aligned}$$

yielding a set of subgroups and a set of excluded groups for each proper group, respectively. A function

$$\text{members} : G \rightarrow \wp(U)$$

is defined for $u \in U$ by⁷

$$u.members \stackrel{\text{def}}{=} u$$

and for $g \in G \setminus U$ by

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

In order to make sure that this definition is sound, it must be guaranteed that the recursion in the definition “bottoms out,” i.e., the group graph is acyclic.

To this end we define a relation $<$ on G as follows: $<$ (with the usual infix notation) is the smallest subset of $G \times G$ satisfying

- (i) if $h \in g.subgroups$ then $h < g$,
- (ii) if $h \in g.excluded$ then $h < g$,
- (iii) if $h < g$ and $g < f$ then $h < f$.

We impose a constraint that the relation $<$ must be antireflexive, that is, $g < g$ does not hold. Formally:

$$(\forall g, h \in G) h < g \Rightarrow h \neq g.$$

Hence $<$ is a partial order on G and the group graph is a acyclic.

The reader may verify that discarding exclusion from the above definitions results in a model that is equivalent to the one presented in Section 3.1.

Notation

In writing out group structures as sets, we write $\neg x$ for an excluded group x . For example, let $g.subgroups = \{h, k, j\}$ and $g.excluded = \{x, y\}$, moreover $x.subgroups = \{x_1, x_2\}$ and $x.excluded = \{z\}$. Then we may write each of the following to denote g :

$$\begin{aligned} g &= \{h, k, j, \neg x, \neg y\}, \\ g &= \{h, k, j, \neg x = \neg\{x_1, x_2, \neg z\}, \neg y\}, \\ g &= \{h, k, j, \neg\{x_1, x_2, \neg z\}, \neg y\}. \end{aligned}$$

4.3 Dynamics of the model

In addition to the operations defined in Section 3.3 we need the following operations:

AddExcluded excludes groups from membership,

DeleteExcluded undoes such exclusions.

AddExcluded fails when it is attempted to exclude a supgroup. The operation **DissolveGroup**, as defined in 3.3, fails when it is applied to a group that contains exclusions.⁸

⁷as in Section 3, we use the object-oriented notation $o.f$, rather than the functional notation $f(o)$ for the definition of functions.

⁸The reason for this is the following. The idea behind dissolving a group is that the group is removed from the system, but other groups are not affected. What does this mean, precisely? We call two states

4.4 A note on the user interface

The matrix presentation of groups vs. views in Section 3.6 can be retained with a small extension. Groups to be excluded from a view are listed in the bottom section of the matrix; see Figure 7. Note that some of the groups listed in the matrix may have exclusions themselves, but this is not relevant for the definition of an ACL to an object (excluded persons are simply not be members of the group).

	view 1	view 2	...	view n
group 1	✓	✓		
group 2		✓		
...				
group m				✓
excluded				
group g	✓	✓	...	✓

Figure 7: Suggested presentation of negative access rights

5 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.

Notation

As a convenient ad-hoc notation we write g_u to express that user u is the owner of group g . In Section 3.7 we made a distinction between “responsible person” and “control right group”, but here, for the sake of simplicity, we assume that the responsible is the only

of the system *weakly equivalent* if all groups existing in both states have an identical set of members in either state. Weak equivalence, however, is not enough. Somebody might do something to the system without being aware that a group has been dissolved. We call two states *equivalent* if they are weakly equivalent and any sequence of operations that can be applied to one of these states will result in weakly equivalent states when applied to both.

Suppose, now, that we have a group

$f = \{g = \{j, \neg k\}, h\}$ with $k \not\prec j \not\prec k \not\prec h \not\prec k$. We want to dissolve g . What should be the new composition of f ? There are, in principle, two options

$$f' = \{j, h\},$$

$$f'' = \{j, \neg k, h\}.$$

f' and f are not equivalent because adding k as a subgroup to j yields $f'.members \setminus f.members = \{k\}$;
 f'' and f are not equivalent, because adding k as a subgroup to h yields $f.members \setminus f''.members = \{k\}$.

person with control right and call him “the owner”. Multiple ownership does not affect any of the following.

5.1 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

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

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

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

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.

5.2 Recursive delegation

In other scenarios recursive delegation is needed. A typical example is the work flow in a ministry [MR95] an object moves down several layers of hierarchy, is processed by somebody, and then moves up the hierarchy in reverse direction. Let obj be owned by the (head of the) minister’s office, denoted o . Let u , the head of a department, be granted a view on the object, with the right to further delegate this view:

$$obj.view = \{o, \{u\}_u\}_o.$$

The head of the department delegates this to the head of the relevant sub-department v_1 . To this end he changes the view group (or, rather, the system changes the view group for him, following a request through a sensible user interface) as follows:

$$obj.view = \{o, \{u, \{v_1\}_{v_1}\}_u\}_o.$$

The group of delegates is owned by v_1 so he can extend this group and delegate the task to v_2 , the head of a unit:

$$obj.view = \{o, \{u, \{v_1, \{v_2\}_{v_2}\}_{v_1}\}_u\}_o.$$

The head of the unit delegates the task to a member of the unit, v_3 , yielding a group structure

$$obj.view = \{o, \{u, \{v_1, \{v_2, \{v_3\}_{v_3}\}_{v_2}\}_{v_1}\}_u\}_o.$$

Now v_3 actually does the work, and asks w to type it. So he nonrecursively delegates the view to w :

$$obj.view = \{o, \{u, \{v_1, \{v_2, \{v_3, w\}_{v_3}\}_{v_2}\}_{v_1}\}_u\}_o.$$

Note that any person in such a chain of delegations may revoke the delegations he made, including further delegations made by his delegates.

5.3 Delegation within a trusted group

Coulouris and Dollimore [CD94a] 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 Section 4). Furthermore, we denote “everybody in the system” by a special constant E .⁹

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 “*non-trusted 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 as in one of the ways discussed in 5.1 and 5.2. But delegation to a person who is not a member of t is overruled by the exclusion of nontrusted persons.

5.4 Implementation considerations

The right to delegate is not an explicit right. One may delegate a view if and only if one has control right on (a subgroup of) the view. The user interface could offer an operation *Delegate*, that allows to specify a delegated group and a delegation mode (recursive or nonrecursive). The system adapts the group structure as appropriate.

Similarly, the right to *revoke* exists if and only if one has control right on (a subgroup of) a view that contains further subgroups other than the group which licenses you the revoke right. The user interface offers an operation *Revoke* listing the names of groups from whom the view can be revoked.

5.5 Transfer of responsibility

In the delegation model discussed so far, the responsibility remains with the creator of the object. It should be possible, however, to transfer the responsibility to another person. A document need not be created by the head of the minister’s office in person, but, more likely, by the administrative staff in the office. At some point, the head of the office takes the responsibility for the document.

In Section 3.7 we briefly discussed the notion of responsibility. There are various ways to model transfer of responsibility (of course to be supported by appropriate functionality in the user interface). A general solution allows the responsible to create an Agroup *obj.candidate-responsible*. Members of this group have the right to take responsibility.

Alternatively, when a person explicitly wants somebody else to take over responsibility, the object could be enhanced with an attribute *transfer_responsibility* that is set to the target user. The user interface should make him aware of the fact that an action (viz., accepting responsibility) is required.

⁹This is elaborated further in Section 6.3.3.

One could also think of a reverse protocol, seeking responsibility, in which case the responsible has to consent that a person requesting responsibility takes over.

We have introduced a single level of responsibility. In a more refined model, one could distinguish between different levels of responsibility. In a working environment where tasks are delegated we can distinguish between the person who is responsible for, say, a document (the delegator) and the person who is currently *in charge* (the delegate). The delegator has the overall responsibility, the delegate has a responsibility towards the delegator to do the things he ought to do within an appropriate time limit.

6 Further extensions to the model

Some other extensions to the basic model are discussed here, in much less detail than Sections 4 and 5. We will briefly discuss conditional authorization (6.1), explicit role change versus implicit group membership (6.2), object structures (6.3), and system administration (6.4).

6.1 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.

Before we discuss further details of the model, let us first look at some examples:

- *Only between 9 AM and 6 PM.*
- *Only from a workstation within the building*, not over a telephone line or remote login.
- *Only in case of emergency.* For example: protected patient data in a hospital can be retrieved by any staff if there is an emergency.¹⁰
- *Only if another member of this group performs the same operation* within a certain time limit. Examples: two persons needed for launching a nuclear missile; two keys needed jointly to open a safe.
- *Only until March 31st* (for a group or a user ID that has been set up temporarily).
- *Only for members of a particular group.* This models the situation where membership of a (sub)group can be extended by some members, but not beyond the pre-set limits of a group of trusted persons (as in [CD94a]: typing the university exams can be delegated only to those secretaries who are on a list of trusted persons.)

¹⁰Note 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.

The same result can be achieved with negative rights, rather than conditions, see Section 5.3.

- “*Dynamic roles*”: Edwards [Edw96] proposes a similar scheme to dynamically include persons in a group (rather than deny access to a group) by run-time evaluation of a function. In our model, one could grant access to “everybody” (cf. Section 6.3.3) and then use a condition to prevent everybody from getting in all the time.
- *Only if you have some rights on another object* you may perform this operation.

Design choices

Two design choices warrant a more detailed motivation.

- Edwards uses dynamic roles to grant access to groups not explicitly mentioned but resulting from the run-time evaluation of a python script. In contrast, we use conditions as filters to *deny* access to groups who, in different circumstances, would have been granted access. 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 arbitrary access, conditions should only allow access within a scope that has been defined inside the access control component.
- On a more technical level, there are two choices for adding conditions into the model
 - conditions are applied to *groups*,
 - conditions are applied to *subgroup relations*.

The latter may seem more natural (*john* is a member of this group only if ...), but the former is more practical. Typically one wants to apply a condition to a view on an object *obj.view*, rather than to all the subgroups contained in that view. Similarly, a condition “John’s authorization expires March 31st” can be added to the user (group) *john*.

Combination of conditions

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.

6.2 Roles versus Groups

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.

There is a need for roles 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 *sysadm*, 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).

Having made the distinction between implicit and explicit role switching, we can now settle the terminology.

Role. A role is a proper user group that requires authentication in order to obtain the authorization given to that group.

In a way, a user ID is also a role: when the user enters the system, he has to authenticate himself. An important difference, however, is that in order to take a role, one must be authenticated as a user of the system. It should not be possible to become “anonymous superuser” by authenticating directly as *sysadm*.

Multiple roles

Roles can be stacked, the same way as groups can be nested. For example, there could be various levels of system administrator, where *sysadm* is the owner of system administration objects, but an additional *superuser* authentication is needed for manipulating objects of arbitrary users.

Acting in multiple roles is possible in principle. However, when awareness information of the type illustrated above has to be created, it could be ambiguous in which role a person does something.

The user should be able to track which roles he currently has and there should be some protocol for laying down a role.

A note on the user interface

In many systems, operations not available to the user in the current state are “greyed out” in the user interface. With the concept of roles that is introduced here, one could distinguish between

- *lightly greyed out*: in principle the operation is possible, but when you try to perform it the system will prompt you for additional authentication.
- *strongly greyed out*: the operation is not logically possible or you cannot take a role that would authorize you to perform it.

6.3 Object structures

It has been shown at length how the group-based authorization model supports a hierarchical structure of the subject dimension and a flat structure of the access type dimension. Structuring the object dimension has not yet been addressed. In this section we will briefly discuss various issues that arise when we consider relations between objects.

We will not consider access rights structures *within* composite objects, as in [SD92, CD94a, CD94b]. In the BSCW Shared Workspace server most objects are “atomic” in the sense that what is inside the object is outside the scope of the BSCW system. Containers (workspaces and folders) provide a hierarchical structure in which such atomic objects can be stored.

6.3.1 Rights on a combination of objects

It is possible that operations involve a combination of rights on different objects. Trivial examples are *move*, which requires both the right to remove an object from its current location and the right to add the object to its new location, and *copy*, requiring the right to read the object and the right to create a new one.

More subtle, in fact, is the right to remove an object. Is it a right on the object, a right on the container in which the object is located, or a combination of both? (In Section 7.1 we discuss in detail the BSCW approach to this issue).

In every case where an operation needs rights on different objects, the user should have all rights involved, otherwise the operation fails.

6.3.2 Indirection of access rights

Rather than defining the access rights for each object individually, it should be possible to refer to the access rights of *another* object. Note that Agroups as introduced in the general model support this. An example: we have an object *obj* in a folder *f1* and we want the object to be readable to all persons who have the *add* view on the folder (and no others). To that end, we define

$$obj.read = \{f1.add\}.$$

As a result, if members are added to or deleted from *add* view group on *f1*, these changes automatically apply to the *read* view group of *obj*.

6.3.3 Variables

Related, but slightly different from indirection is the use of variables—a small extension of the model.

Suppose that the object *obj* of Section 6.3.2 is moved from folder *f1* to another folder *f2*. The *read* view on *obj* is still granted to persons having the *add* view on *f1*. This has not changed when the object was transferred. In order to automatically adapt to the new situation, the *read* view should have been related to the object’s container—irrespective of which container it currently is.

To this end, we can introduce *variables* that adapt their values to the specific circumstances. In the current BSCW system, a variable *members of the current workspace* is used in determining access rights. When an object is moved across workspaces, these access rights to the workspace the object is currently located in.

Along similar lines we may use a variable “Everybody in the system,”¹¹ for convenience denoted by *E*. This is *not* the root of the group graph—because if it were, it could not be used in composing other groups—but a special system variable. Examples of its use:

- Edwards’ “dynamic roles” [Edw96], see also Section 6.1. Create a group $g = \{E\}$ with *g.condition* some condition under which anybody is allowed to access the system (e.g. “today is the demonstration day and the user is located in our lab”).
- Restricting access to a predefined group of trusted persons. (taken from [CD94a], see also Section 6.1). Let *t* be the group of persons who can be trusted to handle some security sensitive task, and *g* the current user group currently involved with the task (Assume $g \subseteq t$). Members of *g* may enlarge the group, but it should be prevented that anybody outside *t* becomes a member. To that end, we define a group $\{E, \neg t\}$ which contains everybody who cannot be trusted, and a group *g'* that excluded untrusted persons:

$$g' = \{g, \neg\{E, \neg t\}\}.$$

Hence only trusted persons can be member of *g'*, irrespective of the composition of *g*. This is treated in more detail in Section 5.

6.3.4 Inheritance of rights

An alternative to indirection and variables as introduced in Sections 6.3.2 and 6.3.3 is inheritance of rights according according to object hierarchy. Note the difference with our object-oriented approach: in 6.3.2 and 6.3.3 it is specified *with the object* where it derives its access rights from, in an inheritance-based approach it is specified higher up in the hierarchy what the access rights of dependent objects are. In database systems, which have an orderly and more or less rigid structure, this increases space efficiency (no access rights to be stored for individual objects) at a rather small cost in time efficiency, cf. [RBKW91].

¹¹Note that BSCW has in fact two different notions: “everybody registered in the system”, and “everybody, whether registered or not”.

For fine-grained authorization in cooperative systems, yet another solution is chosen in [Dew91, SD92]: the introduction of *value groups* in which access rights for a set of objects can be specified.

6.4 System administration

A system may need some system administration functionality. These could be “regular” system administration operations, e.g., get usage statistics of the system, available disk space, etc. These are operations on objects owned by the system and hence within the regular access control model.

Furthermore, there are situations in which a system administrator may have to override normal access control specified by an object’s ACL and perform operations for which he has not obtained explicit authorization from the object’s responsible. Typical examples: a user might ask the system administrator to install a new password, because he has forgotten his current password; the system administrator may remove objects of a user who left without tidying up.

It would be possible to extend the authorization model with notions of system administrator capabilities. One could have role *superuser* which has control right on any object. But it would be simpler, perhaps, to leave system administration *outside* the formal authorization model and define some clear protocol for cases and circumstances under which ACL can be bypassed by users with specific system administration privileges.

In any case, the “superuser” model is too coarse: for most system administration functions one needs *some* additional authorization, not the right to do everything on any object. There is no reason why, in order to change somebody’s password, one should at the same time obtain read and write access to all the objects owned by that user.

7 Realization in BSCW

So far we have discussed authorization in general, without addressing the needs of a particular system. In this section we will have a closer look on some of the issues that arise in the context of the BSCW Shared Workspace system [BHST95, B&al97]. A complete design for the user interface of the access rights components is beyond the scope of this paper, but we will raise some issues that have to be addressed.

A realization is envisaged in Version 2.3, to appear in the course of 1997.

7.1 Removing objects

It is not self-evident how rights for removing objects have to be organized, hence it is worthwhile to have a closer look at the solution that has been adopted for the BSCW system (from version 2.0 onwards).

Following Section 3.7 an object has an owner group containing a (single) responsible. The requirements for object removal have been formulated as follows.

- An owner of a container (typically a folder) should have the right to remove objects from the container of which he thinks they should no longer be there—irrespective of his rights to these objects.

- The owners of an object should have the guarantee that the object is not removed from the system by somebody who does not have the right to do so.
- An owner of an object should, in general, have the right to retract or replace an object, irrespective of the rights on the container in which the object is located.

In order to meet these requirements, we distinguish between the following rights:

- *Relocate*¹² is a right on the object. It allows to remove an object from an arbitrary folder and to put it into another folder (on which the actor should have the appropriate *add* right).
- *Delete* is a right on a container. It allows to remove arbitrary objects from the container. A deleted object migrates to a special container, some person’s *wastebasket*:
 - the actor’s wastebasket if he is an owner of the object,
 - the wastebasket of the object’s responsible if the actor is not an owner.
- *Destroy* is a right on a wastebasket. It removes an object in the wastebasket from the system.¹³ The destroy operation is restricted to the owner of the wastebasket, hence only owners (and perhaps, in extreme situations, system administrators) can destroy an object.

The division into *relocate/delete/destroy* allows the owner of a container to tidy up, while avoiding the risk that objects are destroyed against the wish of their owners. Also it prevents that third persons “capture” an object so that it cannot be retracted by the owners.

7.2 Defaults

It is possible to change the access rights after an object or a set of objects has been created, but it is rather more convenient when newly created objects get the appropriate rights automatically. This raises the issue of defaults, cf. Section 3.8.

In the context of BSCW, Workspaces are the prime structuring aid for giving groups of users access to groups of objects. Workspaces should be organized differently in different *settings*.

In a cooperative environment, for example, you do not want access rights to be overly restrictive. People are supposed to behave in a responsible manner. In order to create a workable environment, it is extremely helpful when *conventions* about what users ought not do can be overridden as contingencies arise. Everybody is, in principle, able to do almost everything. If, however, certain conventions (with which the system need not be burdened) are overridden, the actors can be held accountable for their deeds.

In another setting, e.g., a university course in which students have access to information that may be updated frequently, it is not appropriate to assume the cooperative

¹²This used to be called “cut,” but with the envisaged disappearance of the clipboard (“bag”) “relocate” is a better name.

¹³When a container object (a folder) is destroyed, all the the objects contained in it are destroyed as well—with exception of those objects not owned by the actor. These migrate to the wastebaskets of there respective responsables.

model. You want to physically prevent users from doing certain things. If they have to be done, then go and see a properly authorized person.

Yet another setting is a workspace in which submissions for a conference and referee reports are collected. Following the usual conventions for anonymity and privacy, the emphasis is on making sure that everybody can only see those objects he is entitled to, and no others.

It is assumed that the workspace organization reflects the setting in which users interact with the system, hence it seems most logical to associate access rights defaults with workspaces. Such a set of defaults is described in a *workspace profile*.

In the simplest case, a workspace profile describes the default access rights for newly created objects of every class. In a more sophisticated approach it is possible to differentiate according to location and types of users.

Systems have to be configurable, but within reasonable limits. A system in which everything can be configured by the end user is very flexible, but might be hard to learn and to handle. Additional flexibility has to be weighed against additional complexity. We conjecture that the following policy allows enough flexibility for practical applications, while keeping the system complexity and the user interface lean.

- A set of appropriate workspace profiles can be determined by the system administrator. In addition to standard profiles, in the standard distribution, he should be able to create or adapt workspace profiles optimally suited to the system's intended usage.
- When a user creates a new workspace, he selects one of the available profiles (if there is a choice).

If, for whatever reason, it becomes clear that a workspace in use should have had a different profile, one can create a new workspace and move the objects. Also, custom-tailoring profiles for individual folders by end-users is an extension that can be dispensed with. It is essential, however, that a user is able to change the access rights of a collection of objects (e.g., all documents in a folder) with a single operation.

7.3 Efficiency considerations

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:

- Has user u right r_i on object obj ?
- Which rights does user u have on object obj ?
- 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.

7.4 User interface considerations

So far we have sketched the functionality of the access rights component, not yet its realization in a user interface. To give a complete design of the user interface is beyond the scope of this paper. We only highlight a few important issues.

7.4.1 Changing access rights

Access rights should be grouped into views. Whether these views are fixed or custom-tailorable by the user is a debatable issue; there is a trade-off between simplicity and flexibility and there is something in favour of either option.

Presentation of and interaction with the access rights on a single object should use a groups/views matrix as suggested in Figure 5 on page 18.

The major shortcoming in the access rights as implemented in BSCW Version 2.0 is that changing rights of multiple objects is not possible in a single operation.

This should not be too difficult to implement. If you are owner of the selected objects (otherwise the attempt ends with a warning) you get an access rights form that allows you to add/delete/overwrite access rights in the selected objects.

Note: changing access rights on multiple objects may involve objects with different views. These views can always be decomposed into a set of smaller views that partition (i.e. cover completely without overlap) the applicable set of access rights. A simpler solution is to disallows changing multiple access rights when the view on the selected objects are incompatible.

7.4.2 Visibility

Given the possibility to adapt access rights of objects, it is important that the user is able to see the access rights. A user should not only be able to see his own rights on the objects displayed in a folder listing, but also what other users can do with it. This has two levels.

- A general awareness that certain object are “more visible than others” is needed. In the current BSCW version this is indicated by a special icon (inverted “i”) attached to the object.
- When a user changes access right so as to make an object (in)accessible to others, he should have the possibility to verify whether the result matches his intentions. Showing a large table with boolean values is not an optimal way to do this. More convenient would be the possibility to see a page *as it would be displayed* to users in the target group.

8 Discussion and conclusions

The motivation for the work that has been reported here is the need for more flexible access control in the BSCW Shared workspace system. But the authorization model proposed here is of a more general nature.

In Section 3 we presented a basic model. The canonical authorization model (the Lampson matrix) is extended with a *single* concept, viz., hierarchically structured user groups.

Several possible extensions and applications of the basic model have been treated in Sections 4, 5, and 6. We discussed negative rights and delegation in detail and sketched some further major extensions (conditional rights, explicit role switching) and practical applications (location-based authorization, system administration). Section 7 highlighted some issues in the realization of this model in the framework of the BSCW Shared workspace system.

Section 2 listed several of issues raised in the literature on groupware and access rights. We have addressed most of these issues (roles, delegation, and negative rights). In particular for negative rights we have proposed an extended model that is as general as the model of Shen and Dewan [SD92] but rather more simple because no conflict resolution rules are needed.

We have hardly touched upon the variety of different rights needed in the context of groupware systems (see [DCS94] 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. These issues have to be addressed in the context of a particular system or a particular field of application, rather than in an abstract, general model as has been presented here.

A realization of this general model in the context of BSCW is to be expected in the course of 1997.

It was our intention to keep the authorization model as lean as possible, doing justice to Occam's Razor. But increased functionality and flexibility comes with increased complexity. One cannot state in general which features are needed and which features can be dispensed with; this obviously depends on the application. Therefore we have distinguished between a basic model and several extensions. These extensions are independent: each one can be included or left out as desired, and the different extensions do not interfere with each other.

The presented model is *minimal*: each concept that has been introduced is evidently needed to satisfy legitimate practical requirements. Moreover, several features (views, delegation) could be covered without extending the formal model.

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, and Richard Bentley for discussions and feedback on previous drafts.

References

- [B&a197] R. BENTLEY, W. APPELT, U. BUSBACH, E. HINRICHS, D. KERR, K. SIKKEL, J. TREVOR and G. WOETZEL. 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).
- [BHST95] R. BENTLEY, T. HORSTMANN, K. SIKKEL and J. TREVOR. Supporting collaborative information sharing with the World Wide Web: The BSCW Shared Workspace system. *4th International WWW Conference*, Boston, December 1995, pp. 63–74.
- [CD94a] G. COULOURIS and J. DOLLIMORE. Requirements for security in cooperative work: two case studies. Technical Report 671, Dept. of Computer Science, Queen Mary and Westfield College, University of London, 1994.
- [CD94b] G. COULOURIS and J. DOLLIMORE. A security model for cooperative work. Technical Report 674, Dept. of Computer Science, Queen Mary and Westfield College, University of London, 1994.
- [DCS94] P. DEWAN, R. CHOUDHARY and H. SHEN. An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing* 4 (1994) pp. 219–239.
- [Dew91] P. DEWAN. An inheritance model for supporting flexible displays of data structures. *Software—Practice and Experience* 21 (1991), pp. 719–738.
- [Edw96] W.K. EDWARDS. Policies and Roles in Collaborative Applications. *ACM Conference on Computer Supported Cooperative Work (CSCW'96)*, Cambridge, Mass. (1996), pp. 11–20.
- [EGR91] C.A. ELLIS, S.J. GIBBS and G.L. REIN. Groupware: Some Issues and Experiences. *Communications of the ACM* 34, 1 (1991), pp. 38–58.
- [EW94] C.S. ELLIS and J. WAINER. A Conceptual Model of Groupware. *ACM Conference on Computer-Supported Cooperative Work (CSCW'94)*, Chapel Hill, N.C. (1994), pp. 79–88.
- [FSW81] E.B. FERNANDEZ, R.C. SUMMERS and C. WOOD. *Database security and integrity*. Addison-Wesley, Reading, Mass, (1981).
- [GD72] G.S. GRAHAM and P.J. DENNING. Protection: Principles and practice. In Proc. Spring Joint Computer Conference, *AFIPS Conference proceedings* 40, AFIPS Press, Montvale, N.J. (1972), pp. 417–429.
- [GS86] I. GREIF and S. SARIN. Data Sharing in Group Work. *ACM Conference on Computer-Supported Cooperative Work*, Austin, Texas (1986).
- [GS87] I. GREIF and S. SARIN. Data Sharing in Group Work. *ACM Transactions on Office Information Systems* 5 (1987), pp. 187–211.
- [Hag94] D. HAGIMONT. Protection in the Guide Object-Oriented Distributed System. *ECOOP'94*, Bologna (1994), pp. 280–298.
- [HKK93] H. HÄRTIG, O. KOWALSKI and W. KÜHNHAUSER. The BirliX Security Architecture. *Journal of Computer Security* 2 (1993), pp. 5–21.

- [KR95] R. KANAWATI and M. RIVEILL. Access Control Model for Groupware Applications. In G. ALLEN, J. WILKINSON, and P. WRIGHT (Eds), *HCI'95: People and Computers*. School of Computing and Mathematics, University of Huddersfield, UK (1995), pp. 66–71.
- [LABW92] B. LAMPSON, M. ABADI, M. BURROWS, and E. WOBBER. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions in Computer Systems* **10**, 4 (1992).
- [Lam74] B.W. LAMPSON. Protection. *ACM Operating Systems Review* **8** (1974), pp. 18–24.
- [MR95] P. MAMBREY and M. ROBINSON. 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, 1995.
- [PHRM90] J.F. PATERSON, R.D. HILL, S.L. ROHALL and W.S. MEEKS. Rendezvous: An Architecture for Synchronous Multi-User Applications. *ACM Conference on Computer-Supported Cooperative Work (CSCW'90)*, pp. 317–328.
- [RBKW91] F. RABITTI, E. BERTINO, W. KIM, and D. WOELK. A model of authorization for next-generation database systems. *ACM Transactions an Database Systems* **16** (1991), pp. 79–86.
- [SCFY96] R.S. SANDHU, E.J. COYNE, H.L. FEINSTEIN, and C.E. YOUMAN. Role-Based Access Control Models. *IEEE Computer*, February 1996, pp. 38–47.
- [Sal74] J.H. SALZER. Protection and Control of Information Sharing in Multics. *Communications of the ACM* **17** (1974), pp. 388–402.
- [Sat89] M. SATYANARAYANAN. Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems* **7** (1989) pp. 247–280.
- [SD92] H. SHEN and P. DEWAN. Access Control for Collaborative Environments. *ACM Conference on Computer-Supported Coperative Work (CSCW'92)*, Toronto, Canada (1992), pp. 51–58.
- [Sik95] K. SIKKEL. BSCW: Zusammenarbeit im World-Wide Web. *GMD-Spiegel* 3/95, pp. 20–22.
- [TK97] J. TREVOR and T. KOCH. MetaWeb: Bridging the Gap between Synchronous Groupware and the WWW. Draft paper, GMD-FIT, Sankt Augustin, 1997.
- [TRM94] J. TREVOR, T. RODDEN and J. MARIANI. The Use of Adapters to Support Cooperative Sharing. *ACM Conference on Computer-Supported Cooperative Work (CSCW'94)*, Chapel Hill, North Carolina (1994), pp. 219–230.