A Typed Access Control Model for CORBA

Gerald Brose*

Institut für Informatik
Freie Universität Berlin, D–14195 Berlin, Germany
brose@inf.fu-berlin.de

Abstract. Specifying and managing access rights in large distributed systems is a non–trivial task. This paper presents a language–based approach to supporting policy–based management of access rights. We develop an object–oriented access model and a concrete syntax that is designed to allow both flexible and manageable access control policies for CORBA objects. We introduce a typed construct for access rights called *view* that allows static type checking of specifications and show how a realistic example policy is expressed using our notation.
Keywords: Access control, roles, types, CORBA.

1 Introduction

Due to the heterogeneity inherent in open distributed systems such as CORBA [OMG99], security requirements cannot be enforced by operating systems with their established set of mechanisms alone. Rather, it is the middleware that has to provide platform–independent security services. The correct design and implementation of security mechanisms according to the Object Management Group's (OMG) *Security Service Specification* [OMG98] are not, however, the only technical challenges in ensuring proper overall protection in a distributed object system.

The design, specification, and implementation of security policies and the management of the corresponding access rights at runtime are both error–prone and security–critical. There are few methods or tools that provide adequate support for application designers and security administrators in distributed object systems. The main problems are ensuring scalability while at the same time allowing the description of fine–grained accesses, which requires appropriate grouping constructs. A related problem is manageability. To make large numbers of fine–grained access rights manageable, it is necessary not just to group these rights but also to provide abstractions that represent the underlying policies. Otherwise, the inherent logic is lost and administrators are left with just "raw data". Unfortunately, the existing access control model for CORBA as specified in [OMG98] is inadequate in all these respects [Kar98,Bro99].

In this paper, we are concerned with support for specifying access control policies. In general terms, an access control policy is a description of which

* This work is funded by the German Research Council (DFG), grant No. LO 447/5–1.

F. Cuppens et al. (Eds.): ESORICS 2000, LNCS 1895, pp. 88–105, 2000.

accesses are allowed and which are denied. In a more technical, but still abstract sense, an access control policy is a set of rules that, when parameterized with access control information, is evaluated by an access decision function to yield a boolean result, i.e. an access is either allowed or denied. We take a language–based approach to the problem of specifying and managing access control policies and devise a formal notation that allows designers and administrators to deal with abstractions that are adequate for their tasks. Our aim is to reap all the benefits of language support like documentation, structuring, type–safety, reuse, and enhanced communication between developers and administrators.

An interesting aspect of this work is that providing a usable and manageable environment has implications for the underlying access model. Existing access control models [HRU76,San92,BN89], which have been designed to make certain safety properties tractable or to allow certain classes of security policies that were not expressible in other models, either do not apply well to CORBA or do not support high–level language constructs that help writing specifications. The contributions of this paper are the introduction of a typed grouping concept for access rights called *view* and the definition of a view–based access control language for CORBA that allows static type–checking to ensure the consistency of specifications. The resulting model is intended to serve as a basis for the development of a comprehensive set of tools.

The rest of this paper is structured as follows. Our access model is presented and discussed in section 2. Section 3 contains a realistic example for a policy with dynamic rights changes. Related work is discussed in section 4. The paper concludes with a brief summary and an outlook on future work.

2 A View–Based Access Control Model

To address the scalability and manageability problems outlined in section 1 we need to define the appropriate grouping concepts and abstractions. While the main contribution of this paper is a typed grouping construct for access rights, our policy language also relies on roles as a concept for grouping users. Rather than introducing a new role model, we describe a few basic assumptions for suitable role–based authentication systems in the context of our model. The remainder of this section then introduces our access control model both formally and in a concrete syntax, the *view policy language* VPL [Bro99].

2.1 Roles

To support flexible development, deployment, and management of policies in potentially diverse environments, we cannot rely on the actual identity of users because they are not known in advance. With regard to object invocations, the most suitable abstraction for users is that of a *role* as it allows us to concentrate on the specific, logical function in which a principal is operating on application or system objects. Our notion of role is different from the widely–used role concept

in role–based access control (RBAC) [SCFY96] where roles represent a combination of user groups with sets of authorizations, or just a set of authorizations [FK92].

We use the term role as a synonym for actors as in use case diagrams. In other words, roles are sets of users and group principals based on their common aspects in different interaction contexts. We assume a public–key based service that issues privilege attribute certificates that represent role membership [HBM98] to principals upon request and whose signature is trusted by the access decision function. A security service like [OMG98] can then manage access sessions between callers and objects so that objects always see "users in roles" rather than individual users.

Role names for sets of users are declared in `roles` clauses as in Fig. 1. If we wanted to write a policy that describes accesses to objects representing resources in a university setting, we might want to refer to principals in roles such as lecturer, student, or examiner.

```
roles
     head, lecturer, student, examiner
role assertion
     card( examiner and student) == 0;
     card( head ) == 1
```

Fig. 1: Roles and role assertions

In addition to referring to logical actors, some policies may make assumptions on the way roles are structured, and on the way membership is granted. Role assertions express requirements on the authentication service used to authenticate users in roles and can be written as in Fig. 1. Here, we require that the intersection of `examiner` and `student` is empty, i.e. the authentication service must ensure that no principal who is a member of role `student` is ever granted membership in role `examiner`. Another possible assertion is that membership in certain roles, e.g. the role `head` in the example, is only ever granted to a single principal.

A deployer of a policy must check that role membership is certified in accordance with the assertions expressed in the policy. The assignment of individual users or whole user groups to roles is not expressed in a static policy description but is performed at deployment time. In case the authentication service does not already offer certificates for the roles listed in a policy specification, the deployer can perform a simple renaming operation for role identifiers to map them onto existing roles, e.g., to map an existing role `professor` onto role `lecturer`. If existing roles do not map well, new roles need to be set up in the authentication service.

2.2 Views

To address the requirements outlined in section 1 we propose an object–oriented access model based on *views*. A view is a named set of access rights. These access

rights are both permissions or denials for operations on objects. While access decisions are made based on individual rights, views are the units of description and of granting or revoking.

The need for such a concept is motivated by the observation that it is not adequate to describe object systems using a limited set of generic rights such as "read", "write", and "execute". Thus, unlike the classical access matrix model [Lam74] or the standard CORBA access model [OMG99], individual rights in our model directly correspond to operations in the target object's interface. While this allows access policies that are more expressive and suitable for object–oriented systems, it also introduces additional complexity because the set of access rights is open and potentially very large. To make these access rights manageable, we need to exploit their inherent structure.

An important property of views is that they are typed by the object interface they control. Views are defined as part of the policy specification. Figure 2 shows an example of a view definition in VPL. Views are defined as access controls for a particular IDL interface, which is referenced in the `controls`–clause of the view definition.

```
view NameResolver controls CosNaming::NamingContext {
   allow
      resolve;
      list;
};
```

Fig. 2: A view definition

In the example, the view `NameResolver` controls the IDL type `CosNaming::NamingContext`, which is the IDL interface for the CORBA name service [OMG97]. Permissions are listed after the keyword `allow`, denials would be introduced by `deny`. In the example, only operations to list the name bindings in the context and to resolve a name are allowed.

More formally, let $\mathcal{U}$ be the set of user identifiers (i.e., public keys) and $\mathcal{ROLE}$ the set of roles. At any one time, a user interacts with an object in a single role, so we define the set of subjects $\mathcal{S} : \mathcal{U} \times \mathcal{ROLE}$ as users in roles. Let $\mathcal{O}$ be the set of objects and $\mathcal{V}$ the set of views. We model a system's protection state as a tuple (S, O, M), where $S \subseteq \mathcal{S}$, $O \subseteq \mathcal{O}$ and $M : S \times O \to \mathbb{P}(\mathcal{V})$ is an access matrix, $\mathbb{P}$ denoting the power set. The matrix entry $M_{(s,o)}$ contains subject s's views on object o.

Let $Mode = \{allow, deny\}$ be the set of rights modes, $Prio = \{strong, weak\}$ the set of priorities and Op the set of operation names. Priorities are used for conflict resolution as described below. We define the set of rights as $\mathcal{R} = Op \times Mode \times Prio$. A right $r \in \mathcal{R}$ is thus a tuple (op, m, p), so any right has a corresponding operation, a priority and is either a permission or a denial.

The set of views $\mathcal{V}$ is defined as $\mathbb{P}(\mathcal{R}) \times \mathcal{T}$, where $\mathcal{T}$ denotes the set of object types. A view $V \in \mathcal{V}$ is thus a tuple (R, T), i.e. a set of rights and a controlled object type. We define a number of restrictions for views that ensure their well–

formedness. We demand that the operations for all rights in a view are operations of the object type controlled by the view (1). Also, views may only contain one right definition for any given operation, so a view has no conflicting rights (2).

Let the function $rights : \mathcal{V} \rightarrow \mathbb{P}(\mathcal{R})$ associate a view with its rights. The function $controlled : \mathcal{V} \rightarrow \mathcal{T}$ maps a view to its controlled object type, and function $ops : \mathcal{T} \rightarrow \mathbb{P}(Op)$ maps an object type to the set of its operation names. Note that in CORBA IDL, $ops(t)$ is indeed a set, i.e. all operation names in a type must be unique, so overloading of operation names is not possible.

$$\forall\, v \in \mathcal{V} \;: \forall\, (op, m, p) \in rights(v) : op \in ops(controlled(v)) \tag{1}$$

$$\forall\, v \in \mathcal{V} \;: \forall\, (op_i, m_i, p_i), (op_j, m_j, p_j) \in rights(v) : \\ op_i = op_j \Rightarrow m_i = m_j \wedge p_i = p_j \tag{2}$$

Matrix entries must be well–typed, i.e. they must satisfy condition (3), which ensures that views entered into the access matrix are always applicable to the object in the matrix's column. i.e. that the object has the same type or a subtype of the view's controlled type. Subtyping of object types is denoted by $\sqsubseteq$ and has the usual substitution semantics.

$$\forall v \in \mathcal{V}, o \in \mathcal{O}, s \in \mathcal{S} : v \in M_{(s,o)} \Rightarrow type(o) \sqsubseteq controlled(v) \tag{3}$$

For each object access the views held by the calling subject are checked to determine whether they contain a permission for the requested operation. If they do and, as explained in 2.3, no denials override this permission, the access is granted.

View extension. Like object types, views may be directly or indirectly related through extension so that definitions can be reused. A derived view inherits all its base view's rights — both permissions and denials — and may also add new rights. These added rights may only increase the permissions in the view, however. It is not possible to declare any denials in a derived view.

The semantics of view extension is thus one of monotonically adding permissions, just like interface inheritance adds operations to interfaces and never removes them. As a result, a derived view is substitutable for any of its base views. For the access decision function, this means that if a subject holds multiple views, only the most derived views need to be checked for permissions.

In VPL, extension is expressed by listing base view names after a colon. In the example in Fig. 3, all view definitions directly or indirectly extend `NameResolver`, so they inherit the permission for the operation `resolve` and additionally permit the operations listed in their own definitions.

In the example, the view `NameBinder` extends `NameResolver` by additionally allowing the `bind` operation. The view `NamingContextManager` allows two more operations: new contexts can be created with `new_context` or created and at the same time bound to a given name with `bind_new_context`. Note that

```
view NameBinder: NameResolver {
   allow
      bind;
};

view NamingContextManager: NameBinder {
   allow
      new_context;
      bind_new_context;
};
```

Fig. 3: View extension

`NamingContextManager` does not have an explicit `controls` clause, the controlled type is inherited from its base view. In the case of multiple base views, the controlled object type must be listed explicitly.

Formally, extension on views is denoted by $\leqslant$ ("extends") and has the following properties:

$\forall\, v, w \in \mathcal{V} : v \leqslant w \Rightarrow$

$$controlled(v) \sqsubseteq controlled(w) \wedge rights(w) \subseteq rights(v) \wedge \tag{4}$$

$$\forall (op, m, p) \in rights(v) \setminus rights(w) : m = allow \tag{5}$$

Property (4) requires a derived view to control the same or a more derived object type than its base views and that it has at least as many rights. Property (5) requires all rights introduced in the derived view to be permissions. Note that we do not exclude extension of multiple base views here. View hierarchies can thus be designed along object type inheritance hierarchies.

2.3 Implicit Authorizations, Denials, and Conflict Resolution

An implicit authorization [RBKW91] is one that is implied by other authorizations whenever a grouping construct is used for granting. If, e.g., a view v on an object is granted to a role, this implies granting v to every individual user who may take on this role.

While it is more convenient to specify general access rules this way than to grant each of the implied authorizations individually, it must also be possible to express exceptions to rules, e.g., that users in one particular role are denied one particular operation on an object. Because the absence of a permission cannot be used to override an existing permission, it is necessary to define a means by which negative authorizations [Sti79] or denials can be explicitly specified, as shown in Fig. 4.

If it is possible to define both permissions and denials, conflicts can arise, some of which may represent exceptions. We now describe a strategy that determines whether, in a given conflict situation, the denial or the permission takes precedence. Our conflict resolution strategy relies on the extension relation between views and on explicit priorities in view definitions.

```
view BaseView controls T {
   allow
      op_1;
      op_2;
   deny
      strong op_3;
      op_4;
};
```

```
view DerivedView : BaseView {
   allow
      op_4;
      strong op_3; // incorrect
};
```

Fig. 4: Denials and explicit priorities

Priorities in our model can only take one of two values: strong or weak. As in [RBKW91], the intention of marking a right as "strong" is that it should not be possible for another right to override the strong right in case of conflicts. As an example for explicit priorities, consider again Fig. 4. In the definition of the view `BaseView`, the keyword `strong` marks the denial for operation `op_3` and the permission for the same operation in `DerivedView` as "strong"; all remaining rights in both views are weak.

To control how derived views may redefine inherited rights, we add a restriction to the definition of derived views: A derived view may only redefine weak rights. Strong rights may not be redefined:

$$\forall\, v, w \in \mathcal{V} :\ v \leqslant w \Rightarrow \forall\, (op, m', p') \in rights(v), (op, m, p) \in rights(w) : \\ m' \neq m \Rightarrow p = weak \tag{6}$$

Property (6) allows redefinitions that change an inherited right's mode, but because of property (5) only denials can be redefined as permissions and not vice versa. A redefinition may also be used to make a right `strong` without changing the right mode, so that it cannot be further redefined in derived views. When we check the definitions in Fig. 4 using (6), `DerivedView` is found to be incorrect because the strong denial of `op_3` in `BaseView` cannot be redefined.

Conflict resolution. Basically, conflicts between a permission p and a denial d can only arise in two cases. The first case is that the views that contain the definition of p and d are related by extension. In this case, the more derived view takes precedence.

Two views are only related by extension if one view is directly on the path along the extension hierarchy from a view to the root of hierarchy. Otherwise, two views are unrelated even if they have a common ancestor. In the case that conflicting views are not related, the stronger right takes precedence. If the two conflicting rights' priorities are both weak, the denial takes precedence. To guarantee that a strong permission cannot be overridden in a conflict, however, requires static analysis of view definitions to detect potential conflicts. In case such a potential conflict is discovered, those view definitions that could potentially violate the semantics of "strong" must be rejected.

For the data model defined by OMG IDL, we can statically detect view definitions with potentially inconsistent rights definitions. A type checker can reject specifications or at least print warnings if potentially conflicting definitions are both strong. For two unrelated views, conflicts between rights definitions are only possible if their controlled object types are either equal or related by inheritance. This is the only way that two identical operation names can refer to the same operation in an IDL interface. Note that this restriction on the data model, which prevents interfaces inheriting operations with the same name from different supertypes, is not present in other languages, e.g., in Java. If we were to use Java as the distributed object model, a type checker would not be able to guarantee that a strong right cannot be overridden.

If the following condition holds, we must reject the specification:

$$\begin{gathered}\exists\ v, w \in \mathcal{V} : v \not\leq w \wedge v \not\geq w \wedge\ T_v = \mathit{controlled}(v) \wedge T_w = \mathit{controlled}(w) \wedge \\ (T_v \sqsubseteq T_w \vee T_v \sqsupseteq T_w)\ \wedge \exists\ (\mathit{op}1, m1, p1) \in \mathit{rights}(v), (\mathit{op}2, m2, p2) \in \mathit{rights}(w) : \\ \mathit{op}1 = \mathit{op}2 \wedge m1 \neq m2 \wedge p1 = \mathit{strong} \wedge p2 = \mathit{strong} \end{gathered} \tag{7}$$

Note that checking this condition requires system–level analysis, i.e., all existing views must be checked. To make this analysis and the administration of views in general practical, we assume that the scope of object type extensions and the visibility of view definitions is restricted by the boundaries of policy management domains [Slo94]. We do not further address domains as a grouping concept for objects in this paper, but it is obvious that manageability of large–scale systems depends on appropriate domain management concepts.

The resolution strategy presented above is simple, flexible and sound. It is possible to express both denials and permissions as the exceptional case, and the use of explicit priorities is straightforward. The downside of this approach is that some view definitions must be rejected simply because of their potential for conflict, even if the conflicts might never actually arise.

2.4 Dynamic Rights Changes

A system's protection state is usually not constant. Objects and subjects are added to or deleted from a system, and rights may be granted and revoked for the purposes of delegation of responsibility or as part of an application–specific security policy. We distinguish the following three cases:

1. *Discretionary granting* or revocation occurs when a grantor explicitly calls a grant operation provided by the security service, thereby inserting views into or removing views from access matrix entries.
2. *Automatic granting* or revocation is performed implicitly by the security service. This occurs when operations are invoked that were defined as triggers in an application–specific policy such as *Chinese Wall* [BN89].
3. *Delegation* occurs implicitly during the course of an operation invocation when the target object delegates the call to another object. This might

require to pass on security attributes of the caller, such as role membership certificates. A number of different delegation policies are possible here [OMG98], which also depend on whether the target object has security attributes of its own. We do not describe delegation in more detail here. Suffice it to say that, in general, these attributes do not correspond to rights, so no new rights are added to the system. Rather, receiving subjects may now qualify for membership in additional roles and thus gain additional access rights.

Discretionary Granting. A granting subject can pass on only views that it possesses and that are also marked as grantable. Rights granted in this way are restricted to permissions, so a subject cannot restrict another subject's allowed operations by granting its own denials. With this restriction, there is no need for recipients to explicitly accept granted views. A simple example is given in Fig. 5.

```
view GrantableView controls T {
   allow
      grant {tech_staff, janitor};
      enter;
};
```

Fig. 5: A grantable view

To allow for explicit, discretionary granting the view `GrantableView` is marked grantable by allowing the operation `grant`. While grant is in fact a meta–right that corresponds to the `enter` command in matrix models such as [HRU76, San92], it is modeled just as any other right and may be both a permission or a denial. It can also be marked as strong.

Additionally, we assume a relation *grantable_to:* $\mathcal{V} \times ROLE$ which lists legal recipient roles for a view. In VPL, this relation is expressed by parameterizing `grant` with a set of roles that may receive this view. If no recipient roles are given, a view can be granted without restrictions, otherwise the view may only be entered into $M_{(s,o)}$ if $s = (u, r)$ and r is one of the target roles.

As mentioned above, an additional restriction for well–formed views is that a grantable view must not contain denials:

$$\forall\, v \in \mathcal{V} :\ \exists\, (grant, allow, p) \in rights(V) \Rightarrow\ \neg\, \exists\, (op, deny, q) \in rights(V) \quad (8)$$

By inserting a view v into $M_{(s,o)}$, a grantor effectively acquires a right on the matrix entry $M_{(s,o)}$ that allows him to again remove that view at his discretion, which potentially leads to further revocations if the grantee has passed on the view after receiving it.

One potential problem with our approach is that a matrix entry is a *set* of views, so if a grantor passes on a view that the grantee already possesses, the grantor would acquire the right to delete that right at any time — such as in the

very instance the view was granted. In effect, it would be possible for a grantor to revoke all those views from another subject that both the grantee and the grantor possess. This can be avoided if we change the model so that either matrix entries are multisets, or if a grantor only acquires revoke rights on views that the grantee does not already possess. For practical reasons, we chose the latter solution because it requires no additional bookkeeping.

Automatic granting. It is appropriate to describe discretionary granting in a view definition because the ability to grant a view depends only on the possession and grantability of a view. This is not so for automatic granting or delegation, however. Both delegation and automatic granting depend only on the invocation of a particular operation and could thus be described in an extended interface notation or as IDL annotations. For better integration with other parts of the policy specification we introduce a new language construct `schema`.

As an example, we describe how an owner status is assigned to the subject calling a factory object's create operation. For this example, we rely on the two IDL interfaces `Document` and `DocumentFactory` in Fig. 6.

```
interface Document {
   void read(out string text);
   void write(in string text);
};
```

```
interface DocumentFactory {
   Document create();
};
```

Fig. 6: Interfaces `Document` and `DocumentFactory`

Figure 7 lists the view and schema definitions. To give owner status for a newly created object to the caller of the `create` operation on a `DocumentFactory` object, the schema for `DocumentFactory` has a `grants` clause for the `create` operation. This clause specifies that a view `Owner` on the `result` object is to be granted to `caller`. `result` and `caller` are reserved identifiers with the obvious meanings. Schema rules must also be able to refer to out parameters of the operation in case it is necessary to modify views on objects passed to the caller this way. In these cases, the schema can use the name of the formal parameter, which is unique in the operation context. This is not shown in the figure.

An `Owner` view, which gives access to all operations in the interface, allows the unrestricted granting of this view to other subjects. To also illustrate automatic revocation the schema also contains a `revokes` clause which specifies that a `User` view is to be revoked from the caller, thereby removing his right to call the create operation again.

A number of points are worth noting about schemas. First, while schemas do introduce dynamic state changes whose consequences for a particular policy might be hard to predict, they do not introduce new conflicts. This is only possible through the definition of new views. Thus, resolvability of all potential conflicts between rights is still preserved. Second, the `grants` and `revokes` clauses can be regarded as operation postconditions with respect to the protection state of the policy domain as they describe an operation's effect on this protec-

```
view Reader controls Document {
   allow
      read;
}
view Owner: Reader {
   allow
      grant;
      write;
}
view User controls DocumentFactory {
   allow
      create;
}

schema DocumentFactory {
   create
      grants
         Owner on result to caller;
      revokes
         User on this from caller;
};
```

Fig. 7: Views and schema for document creation

tion state. Third, an implementation of this concept must be able to undo the effects of these clauses if they occur in an invocation context that is later aborted, e.g., because of lack of permissions for a subsequent operation invocation or because of an exception.

3 An Example Policy

As a case study in VPL we present an application–specific policy for a system that supports programme committees in reviewing papers for a conference.[1] This system is a simple workflow application and supports the following procedure:

1. Authors may submit papers until the deadline is reached. The programme committee (PC) chair assigns a number of reviewers to each paper. This assignment process is not explicitly supported.
2. Reviewers write and submit reviews. After a reviewer has submitted a review for a paper, he may read other reviews for the same paper — but not before he submits. He may now also modify his own review, but not others. (The idea is to shield each reviewer from other reviewers' influence until he commits, but to allow the resolution of conflicts between reviews before the final PC meeting.)

[1] This example is designed after a similar system called CyberChair which is being used by the ECOOP conferences. For details see `http://wwwtrese.cs.utwente.nk/CyberChair/`

3. The final decision for each paper — approval or rejection — must be unanimous. (The resolution of all remaining conflicts is left to the involved reviewers and is not explicitly supported.)

3.1 Application Design

We can derive use cases or scenarios directly from the above description. Even if, in simple cases like this one, a use case–model is not strictly necessary, it is useful for the design of the application's security policy. In this example, the following scenarios occur (the respective actors and interfaces are listed in brackets):

1. Change of processing phases (PC chair, `Conference`)
2. Submission of papers (authors, `Conference`)
3. Reviewing (Reviewers, `Conference, Paper, Review`)

```
interface Conference {
   void callForPapers();
   void deadlineReached();
   void makeDecision();
   void submitPaper(in string paper);
   void listPapers(out string list);
   Paper getPaper(in long paper);
};
```

Fig. 8: Interface `Conference`

The identified actors are not represented in the system. Notification of authors is via e–mail and not through remote invocation. Figures 8 and 9 list the necessary object interfaces in CORBA IDL.

The main process starts when the PC chair issues a call for papers by invoking `callForPapers()`. Papers are submitted as arguments of the operation `submitPaper()` and represented as strings. The conference object creates objects of type `Paper` from these strings. When the chair has called `deadlineReached()` to finish the submission phase, reviewers can retrieve submissions by calling `getPaper()` and giving a reference number as an argument. The operation `listPapers()` is called to list available papers with their reference numbers.

```
interface Paper {
   void read (out string text);
   Review submitReview(
         in string review,
         in long reviewer);
   void listReviews(out string list);
   Review getReview(in long reviewer);
};
```

```
interface Review {
   void read(out string text);
   void update(in string text);
};
```

Fig. 9: Interfaces `Paper` and `Review`

The interface `Paper` allows reading and listing reviews that have already been submitted for this paper. Reviewers who submit by calling `submitReview()` get a `Review` object in return which they can then modify if necessary. After calling `submitReview()` they may also retrieve the reviews of other reviewers using `getReview()`.

3.2 Policy Design

Using the actors identified above we can directly derive the role declarations in Fig. 10. The role `author` is meant to be available to all users excluding those that have been assigned membership in role `chair` — which may only be done for a single principal. Membership in role `chair` implies membership in role `reviewer`.

```
roles
   chair, author, reviewer
role assertion
   author implies not chair;
   chair implies reviewer;
   card( chair ) == 1
```

Fig. 10: Roles and assertions

Static policy aspects. Designing views for the scenarios sketched above is straightforward. We define the views in Fig. 11 to capture the static aspects of this policy.

```
view Member controls Conference {
   allow
      listPapers;
      getPaper;
}
view Chair: Member {
   allow
      callForPapers;
      deadlineReached;
      makeDecision;
}
```

```
view ReviewPaper controls Paper {
   allow
      read;
      listReviews;
}
```

Fig. 11: Views

There are two views for reviewers: `Member` controls objects of type `Conference`, and the view `ReviewPaper` controls `Paper` objects. A `Member` view on conference objects allows listing as well as retrieving papers, a `ReviewPaper` view on paper objects allows reading and listing information about reviews that have been submitted so far. Another view `Chair`, that extends the member view

that controls `Conference`, defines the rights to switch between the processing stages.

To assign initial views on all objects of a type to roles, VPL has a keyword `holds` as shown in Fig. 12. If an object's type can be inferred from the view, it need not be listed explicitly. It is possible, however, to assign a view on subtypes of the view's controlled object type. The scope of an object type extension is assumed to be restricted by the boundaries of the management domain.

```
chair holds Chair;
reviewer, chair holds ReviewPaper on Paper, read on Review;
```

Fig. 12: Initial views

Here, the chair holds an initial `Chair` view for all `Conference` objects in the domain, but the extension of `Conference` is supposed to contain just a singleton. At the same time, reviewers and the chair hold two more views, viz. `ReviewPaper` on all `Paper` objects and another, anonymous view that allows reading all reviews: `read` is simply a shorthand notation for:

view _ **controls** `Review` { **allow** `read;` }

Since, at this stage, reviewers have no views that would allow to retrieve papers using `getPaper` or to retrieve reviews using `getReview`, these operations will in effect only be usable after a `Member` view has been granted.

Dynamic aspects. The most interesting feature of this policy are the changes in the protection state when reviews are submitted: before this point, reviewers may not read other reviews; from then on, they may. Which accesses are permitted thus depends on earlier accesses, similar to the *Chinese Wall* policy [BN89]. To describe transitions like these that are directly connected to changes in the application state, we use schemas. In this example there are two schemas, one for the conference interface and one for submissions.

The `Conference` schema describes how the protection state changes in reaction to operations on the conference object. If `callForPapers` is called, views are assigned to authors that allow to submit papers. These views are again anonymous and only contain the permission for the operation `submitPaper`. In addition, the `Member` view on the conference object is assigned to reviewers. After the deadline for submissions is reached, the permission to submit papers is revoked. At the same time, reviewers receive views that allow them to submit reviews. Finally, when the reviewing process is ended by the chair calling `makeDecision`, reviewers may no longer submit reviews.

The `Paper` schema defines that the right to submit a review for this `Paper` object is revoked for the caller so that only one review may be submitted per reviewer. When submitting, reviewers receive new views that allow to retrieve other reviews for this paper.

```
schema Conference {
   callForPapers
      grants
         submitPaper on this to author;
         Member on this to reviewer;
   deadlineReached
      grants
         submitReview on Paper to reviewer;
      revokes
         submitPaper on this from author;
   makeDecision
      revokes
         submitReview on Paper from reviewer;
}
schema Paper {
   submitReview
      grants
         update on result to caller;
         getReview on this to caller;
      revokes
         submitReview on this from caller;
}
```

Fig. 13: Schemas

4 Related Work

Language approaches to protection have been know since the 1970s. An early approach is [JL78] which uses ADTs for enforcing protection. Here, however, protection is part of an implementation and not described separately. Another language–based and also object–oriented model for non–distributed environments is [RSC92]. This model does distinguish between different classes of principals but again does not separate policy specifications from application implementation.

The concept of *views* as an access control concept was first used for a distributed object system in [Hag94]. Views are also used for protection in relational and object–oriented databases[SLT91]. Their use for access control purposes resembles the use of type abstraction as a protection concept. Unlike database views that can span multiple types, a view in our model is restricted to objects of a single IDL type. Joining views on different IDL types $T_1, ..., T_n$ can, however, be modeled by specifying an additional IDL interface T that extends $T_1, ..., T_n$ and defining a view on T. Another difference is that database views may define content–specific access controls, e.g., by stating that an attribute may only be read if its value is above a certain threshold. While this is a possible extension to our model, it is not possible in its current form.

PolicyMaker [BFL96] is a generic skeleton language for policy statements in which filter programs can be embedded. Applications perform access checks themselves by querying a policy database that evaluates unstructured request

strings according to the filters. While this approach addresses distributed systems and can model complex trust relations between keys, it does not impose structure on policy specifications nor offer any kind of type checking for policy statements.

Grouping privileges into named protection domains to enhance support for the security management of relational databases has been proposed in [Bal90]. Our approach is similar, but more fine–grained and more modular: named protection domains inherently group not only privileges but also objects, and may even group users. Views describe authorizations on individual objects and combine with more appropriate management concepts for users and objects, viz. roles and domains. We believe that protection domains are not applicable to the richer data models of distributed object systems.

A general framework for defining arbitrary access control policies is proposed in [JSSB97] where policies are formulated as a set of rules in a logic–based language. This model leaves open all design decisions about how implicit authorizations are derived, how rights propagate in groups, which conflict resolution strategies are used and how priorities are employed. Rules for these questions have to be defined first as part of a policy library. The data model for protected objects is also left open and has to be described separately. The protection state is extended with a history component that logs all accesses as facts in a database in order to enable state–based policies like *Chinese Wall*. This model exhibits a more complex concrete syntax than ours, policy specifications are less structured.

Adiron [Adi], a vendor of CORBA Security products, provides an access control language with their product, but this language is not object–oriented and also limited to the restricted standard model of access control in CORBA.

5 Summary and Future Work

In this paper we presented a new access control model based on a typed grouping construct for rights called *view*. We introduced a concrete syntax, VPL, that allows application developers and security administrators to specify access control policies for CORBA domains at an abstract language level and gave an example of how a specific access policy can be expressed using our notation. The model might be extended with a more general priority system in the future.

We are currently working on a partial implementation of the CORBA Security Service for our own CORBA implementation JacORB [Bro97] and on integrating our model with this implementation to prove its feasibility and practical value. Other current and future work includes refining our notion of domains and defining language constructs that allow the composition of domains and their policies, e.g., into domain hierarchies. Building on these concepts, we intend to develop management tools for policy domains.

Acknowledgments. I would like to thank Peter Löhr for many valuable discussions. I would also like to thank the anonymous referees for helpful suggestions.

References

[Adi] Adiron. http://www.adiron.com/.

[Bal90] Robert W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 116–132, 1990.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Distributed trust management. In *Proc. IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[BN89] David Brewer and Michael Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[Bro97] Gerald Brose. JacORB — design and implementation of a Java ORB. In *Proc. International Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, pages 143–154, Cottbus, Germany, September 1997. Chapman & Hall.

[Bro99] Gerald Brose. A view–based access model for CORBA. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, pages 237–252. Springer, 1999.

[FK92] David Ferraiolo and Richard Kuhn. Role–based access control. In *Proc. 15th National Computer Security Conference*, 1992.

[Hag94] Daniel Hagimont. Protection in the Guide object–oriented distributed system. In *Proc. ECOOP 1994*, LNCS, pages 280–298. Springer, 1994.

[HBM98] R. J. Hayton, J. M. Bacon, and K. Moody. Access control in an open distributed environment. In *Proc. IEEE Symposium on Security and Privacy*, pages 3–14, 1998.

[HRU76] M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[JL78] Anita Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Communications of the ACM*, 21(5):358–367, May 1978.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proc. International Conference on Management of Data*, pages 474–485, 1997.

[Kar98] Günter Karjoth. Authorization in CORBA security. In *Proc. ESORICS'98*, pages 143–158, 1998.

[Lam74] Butler W. Lampson. Protection. *ACM Operating Systems Rev.*, 8(1):18–24, January 1974.

[OMG97] OMG. *CORBAservices: Common Object Services Specification*, November 1997.

[OMG98] OMG. *Security Service Revision 1.5*, November 1998.

[OMG99] OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2.3*, June 1999.

[RBKW91] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrel Woelk. A model of authorization for next–generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.

[RSC92] Joel Richardson, Peter Schwarz, and Luis-Filipe Cabrera. CACL: Efficient fine–grained protection for objects. In *Proc. OOPSLA 1992*, pages 263–275, 1992.

[San92] Ravi S. Sandhu. The typed access matrix model. In *Proc. IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role–based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Slo94] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable views in object–oriented databases. In *Proc. 2. Int. Conf. on Deductive and Object–Oriented Databases*, number 566 in LNCS, pages 189–207, Berlin, Germany, 1991. Springer.

[Sti79] Helmut G. Stiegler. A structure for access control lists. *Software — Practice and Experience*, 9:813–819, 1979.