Type-Level Access Controls for Distributed Structurally Object-Oriented Database Systems*

Udo Kelter
Praktische Informatik V, Fachbereich Informatik
FernUniversität Hagen, Postfach 940, D-5800 Hagen, Germany
kelter@fernuni-hagen.de

Abstract
Structurally object-oriented database systems are a new class of dedicated data storage systems which are intended to form the basis of CAD, CASE, and other design environments which are to support large, distributed development teams. Several concepts of discretionary access controls (DAC) for such systems have been proposed; these concepts support nested complex objects and nested working groups. However, they do not support roles in development teams such as designers, reviewers, managers etc., whose access rights are typically given in terms of object *types* rather than only in terms of objects.

This paper presents a concept of type-level DAC which is intended to complement instance-level DAC and to support roles in development projects. The concept consists of a formal model of the state of the object base with regard to access controls and a formula which derives from this state and the security context of a process the type-rights of this process. Our model has virtually no built-in, enforced policies; it allows users to realize a wide range of application-specific security policies. It supports multiple inheritance; in order to prevent inconsistent rights on types with common subtypes, certain consistency constraints are introduced. The model is group-oriented in that it supports nested working groups and inheritance of rights along group hierarchies. Access to individual types can be explicitly denied. It is implementable in a distributed system; the administration of rights can be fully decentralized.

Keywords: views; discretionary access controls; object-oriented database systems; distribution; multiple inheritance; group-orientation; access control lists

1 Introduction and Overview

Design environments for CAD, CASE or similar application domains impose new requirements on their underlying data management system. Conventional database systems or file systems do not fulfil these requirements. This has led to the development of a large number of object-oriented database management systems

*This work was supported by Bundesministerium für Forschung und Technologie, Bonn, Germany, under grant no. ITS 9104 C.

(**OODBMSs**; surveys appear in [UnS90, Vo91]). Since there is a wide variety of different data models, they are not easy to classify; a frequently used classification distinguishes between

- **structurally** object-oriented data models, which offer hierarchically structured complex objects; objects, attributes and relationships are accessed using generic operations such as `copy`, `read` or `write`;
- **behaviourally** object-oriented data models, which allow type-specific operations to be defined on objects, i.e. to encapsulate objects according to the principles of abstract data types; they do not generally support complex objects.

This paper deals with discretionary access controls for distributed, structurally object-oriented DBSs. More specifically, we will mainly refer to systems which have been designed to form the basis of *software development environments*, which are used in large development projects. We will use the term **object management system (OMS)** to refer to such database management systems and the term **object base** to refer to the database managed by the OMS. The main features of such OMSs will be presented in section 2.4.

This paper deals only with **discretionary access controls (DAC)**, not with mandatory access controls, since software developers traditionally prefer DAC. DAC are means of restricting access to data granules on the basis of the identity of **subjects** and/or groups to which the subjects belong. The controls are discretionary in the sense that certain subjects ("owners") of a data granule can determine whether and how other subjects can access this data granule.

DAC concepts for conventional database management or file systems, e.g. conventional access control lists or views, are not adequate for OMSs because they do not meet the novel conditions for, and requirements on, access controls in OMSs (see also [DDMR91, EURAC89, GrS87, Ke90, LuF90]):

- There is a hierarchy of nested, *overlapping complex objects*. A complex object, e.g. a document, a parse tree, a module hierarchy, or parts thereof, is the typical unit of access in software development environments, rather then a set of atomic objects which is specified by a query. Therefore, each complex object must be a granule of access control.
- Data managed by an OMS are typed. Typically, users can define *hierarchies of object types*, even using *multiple inheritance* in many OMSs. Type hierarchies must therefore be supported by access control concepts for OMSs.
- Design environments are mostly based on workstations and servers which may form large networks. *Distribution* of data in such architectures must be supported.
- The members of a development team are organized in *nested working groups*. Such hierarchies of groups must be supported; we call such access controls **group-oriented**.

The resultant requirements on access controls are discussed in more detail in section 2. In sum, access controls for OMSs present a new challenge and require new approaches.

Only a small number of concepts for DAC in object-oriented database systems OMSs have been proposed until now [Br91, CAIS88, DiHP88, FeGS89, Ke90, La&90, LuF90, Ra&90]. Most of these concepts support neither nested working groups nor distribution; in order to do so, they would have to be significantly modified or extended. Only some of them deal with types and type hierarchies; there are two completely different approaches for dealing with types:

1. One approach is to treat a type as the set of all its instances (i.e. as a *class*). Here, a right on an object type implies (!!) this right on all objects of this type. This means that the set of all instances of a type is simply another granule within the hierarchy of nested granules.
2. The other approach is to treat access controls for types and for instances as *orthogonal* to each other. This means that access can be restricted both
   - on the basis of individual instances, i.e. on the *instance level*, and
   - on the basis of types, i.e. on the *type level.*

   The overall effect of these levels (or dimensions) is roughly the same as the effect of a selection and a projection which form a view in relational database systems (s. fig. 1; a more detailed analysis appears in [Ke91]) if we regard an object as a tuple of a relation:
   - the rights on instances restrict access to certain objects / tuples. In views, this is achieved by means of a selection. Other systems use object-specific access control lists for this purpose.
   - the rights on types restrict (e.g.) access to certain attributes of objects or relationships. In views, this is achieved by means of a projection.

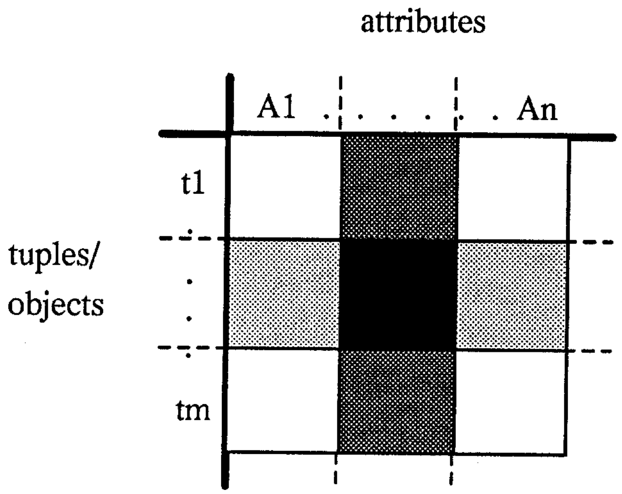

Figure 1: Effect of orthogonal type-level and instance-level access controls

This paper adopts the second approach, i.e. orthogonal access controls on the instance level and on the type level, and will be entirely concerned with type-level DAC. We will assume an instance-level DAC concept which supports complex objects as granules of protection, distribution and nested working groups (as e.g. [Ke90, PCTE90]).

This paper presents a concept for type-level access controls in OMSs which meets the requirements mentioned above. This concept includes a formal model of the state of an object base with regard to security, the security context of a process accessing the object base, and a formula which determines which accesses are allowed. The models and the formula should be seen as the *specification* of a security mechanism which allows one to implement a wide range of discretionary security policies. The model itself (or a mechanism which implements the model) does virtually not constrain the security policies. The constraints are necessary to prevent inconsistent access right determinations which can occur due to multiple inheritance.

This concept has been developed for, and implemented in, one particular OMS, namely H-PCTE. H-PCTE is a high-performance, main-memory-oriented version of PCTE[1] [PCTE90, PCTE91]. However, we present our concepts on a level which abstracts from most details of the data model of H-PCTE (or other OMSs) because they are not of immediate relevance here and because our concepts are actually applicable to a wide range of structurally object-oriented OMSs for CASE and other application domains. Of course, detailed features of other OMS data models may necessitate certain adaptations.

The rest of this paper is organized as follows: Section 2 introduces, for several problem areas, background information, definitions, basic features of our concept, and a summary of the main problems of this area. Section 3 introduces a central notion of our concept, type-level access right determinations. Section 4 presents a model of the type rights of a process, i.e. of the "semantics" of an external schema. Section 5 outlines the way in which the concept has been implemented in H-PCTE. Section 7 compares our approach with other proposals.

2 Problem Analysis and Basic Definitions

This section introduces, for several problem areas, background information, definitions, basic features of DAC concepts for OMSs, and a summary of the main problems. Readers familiar with OMSs or with DAC in OMSs may skip the relevant subsections.

2.1 Basic Notions

2.1.1 Instance-Level DAC

A **(data) granule** is a passive entity which contains or receives information. Access to a granule potentially implies access to the information it contains. Normally, the term 'object' is used instead of 'granule'; however, this would be confusing here since objects are not the only granules in DAC for OMSs.

A **subject** is an active entity, e.g. a person or a device, that causes information to flow among objects or changes the system state. We assume here that the OMS is accessed by executing programs on behalf of a (human) user.

[1]PCTE is the acronym of "A Basis for a Portable Common Tool Environment".

An **access mode**, or simply a **mode**, is a name for a set of elementary modifications or retrievals in the OMS.

We model the state of the object base with regard to instance-level DAC as a mapping

$$iard : S * G * M \rightarrow V$$

with S being the set of subjects, G being the set of data granules, M being the set of access modes, and V being the set of access values[2]. The mapping *iard* can also be regarded as a set of quadruples

$$(s,g,m,v)$$

s being a subject, g being a granule, m being a mode, and v being a value which indicates whether s is to be allowed to access g using operations in m. Such a quadruple is called a **instance-level access right determination (IARD** for short). The object base contains, for each triple (s,g,m), exactly one IARD (s,g,m,v) with $v = iard(s,g,m)$.

There are various ways in which the abstract state can be implemented, e.g. using object-specific access control lists.

The object base can only be accessed by a process. A process acts on behalf of one or several subjects[3]. If a process calls an OMS operation then the state as determined by the IARDs is **evaluated** (this evaluation can be non-trivial), that is, the **(instance-) rights** of this process are computed.

2.1.2 Schemata

We will assume that all the data managed by an OMS is typed. The **conceptual schema** of an object base contains the definition of all the types which are known in this object base.

Individual processes are given a selective view on the object base by means of an external schema. An **external schema** (or view[4]) of a process is, roughly speaking, a subset of the type definitions of the conceptual schema; these types are called **visible** in this external schema. In addition, the set of operations which are applicable to the instances of a given type may be restricted. An external schema thus represents a set of "**type-rights**" of a process, namely the right to "see" certain types and to perform certain operations with instances of these types.

2.2 Distribution

Design environments are typically based on workstations and servers which are connected by a local area network. We assume that the object base is distributed over

[2]Note that this is a straightforward extension of the well-known access-matrix model: the access matrix defines a mapping which maps each pair (s,g) onto a set of allowed access modes if access modes are distinguished, or onto a Boolean value otherwise. Since we assume more than 2 access values, the access matrix is no longer appropriate.

[3]How users identify and authenticate themselves and how their identity is passed on to processes falls out of the scope of this paper.

[4][PCTE90] actually uses the term 'working schema'; we will use more usual terminology in this paper.

the network. This requires the object base to be partitioned into several **segments** which can be independently stored, e.g. in a file or a "raw volume". Moreover we assume that certain types of volumes, e.g. optical discs or floppy discs, can be temporarily dismounted.

Finally we assume that the OMS allows users to *move individual design objects* between segments (typically in order to have them locally available).

2.2.1 Main Problems

Owing to the advent of large networks and portable workstations, we have to assume that it is fairly probable that single workstations which are part of an OMS installation are unreachable, e.g. due to a network failure or because they are disconnected from the network or non-operational (e.g. switched off). Note that this kind of situation may last for a *long period* (up to several weeks). Moreover, a segment may be unavailable because the volume where it is stored has been dismounted. A very important design goal is therefore for the OMS to remain *resilient against the unavailability of segments.* It should be possible to perform sensible work on a site with the segments reachable from this site. A special case of this is the autonomous operation of an isolated workstation. Dependencies on data stored in other segments, or on central resources, must be strictly avoided. (Communication delays are another reason for adopting this design policy.)

It must also be possible to install new software development tools when only a subset of all segments is accessible. Since tools typically use private data types, their installation will require an extension, or modification, of the conceptual schema and the external schemata. As a result,

- the conceptual schema and the external schemata must be managed in such a way that they *can evolve independently at different sites*;
- a single site may only know *part* of the conceptual schema;
- we *cannot* assume a *centrally* administrated conceptual schema.

Traditional approaches for the definition of views, e.g. view definition languages in relational systems, are no longer applicable, because they are biased towards central administration.

2.3 Subjects

We make certain assumptions on how projects which use an OMS-based environment are organized into subgroups. These assumptions and the resultant features of group-oriented DAC will be outlined in this section. A more detailed discussion of this topic can be found in [Ke90, Ke90a].

Working groups in a project are formed according to a repeated division of the overall task of the project into smaller tasks. In general, there can be a partial order of groups. Work may be divided

- *quantitatively*, e.g. a system is divided into subsystems which are developed independently, or

- *qualitatively*, e.g. according to usual roles in a project (analyst, designer, programmer, manager, technical writer etc.).

We will assume the following *basic features of group-oriented DAC* in the rest of this paper: Groups and their subgroup structure are managed by the OMS. The "subgroup of" structure is a connected, acyclic graph with one root, namely the predefined group WORLD. Each user is a member in at least one group and therefore directly or indirectly a member of the group WORLD. Groups can be subjects in IARDs. An object can have IARDs for arbitrarily many groups.

The rights of a group can only be exploited if this group has been "**activated**" for the process which performs the access. In general, *several* groups can be activated at the same time. The set of active subjects of a process is called its **security context**.

2.4 Data Granules

This subsection intends to give the reader some intuitive understanding of the data granules occurring in OMSs. We abstract from all the details of concrete OMSs which are irrelevant for our DAC concept.

2.4.1 Structurally Object-Oriented Database Systems

Typically, the data model of an OMS is derived from the entity relationship model. An object base contains **objects** and **relationships**. Objects and/or relationships are said to have **attribute instances**. We will assume that relationships are binary and that each single relationship is actually a pair of directed **links**, which are mutually reverse of each other and which are used to navigate between objects.

The most prominent feature of the data model of OMSs are complex objects. A **(complex) object** consists of its **root node**, which contains the attribute instances and the outgoing links of the object, and its **component objects**, which in turn are complex. We assume that each component object is connected with the root node by a special kind of link, namely a **composition link** (see below). Complex objects enable us to directly model all kinds of documents occurring in software development environments. A nested complex object could represent, e.g., a (program) module and contain, for each inner module of this module, another object representing the inner module. Other examples are a book consisting of chapters and sections, or a data flow diagram with its stepwise refinements. Complex objects can share components (**shared objects**). Two books, for example, might share a glossary.

2.4.2 Basic Features of Instance-Level DAC

We assume an instance-level DAC concept with the following features (regarding data granules; support of nested working groups has already been discussed above): Complex objects and root nodes are granules of protection. Administration of rights is decentralized on a per-object basis.

Some features of instance-level DAC are not directly relevant here, e.g. consistency rules which restrict the ways in which access rights can be granted on complex objects with shared components, or ways of overriding IARDs which are inherited from outer granules.

The concept presented in this paper does not make any assumptions about how these features are implemented.

2.5 Data Types

The following sections give a simplified description of how types can be defined in OMSs. We omit a number of detailed features, e.g. integrity constraints, because they vary considerably between different OMSs. We assume that each definition has a system-wide unique **definition identifier**.

2.5.1 Type Definitions

Object Types. We assume that each object has exactly one object type. An **object type** is essentially defined by:

- a set of direct supertypes
- a set of direct subtypes
- a set of explicit attributes
- a set of admissible types of outgoing links

We assume that the subtype structure is a lattice with one root named `Object` and that a new object type is always created as a subtype of one or several existing object types (multiple inheritance).

The set of attributes of an object type is the union of the set of its own explicit attributes and, for each direct supertype, of the set of attributes of this supertype. These sets are not necessarily disjoint; two attributes are "the same" if they have the same definition identifier.

Attributes. An attribute is defined by:

- an attribute type, i.e. a set of values, e.g. string, integer or real
- an initial value.

Links and Relationships. Links (or relationships) connect objects which are associated with each other. Links form specifically the basis for navigation in the object base. The details of this vary in different OMSs. Some OMSs realize links as surrogate-valued attributes. We will assume a fairly complex notion of a keyed, attributed link (as a result, our concept can be easily adapted to less complex cases).

Each link has a **link name** which identifies the link among those links which lead off from the same object, i.e. link names have a *local* key property. A link name consists of the values of the key attributes of the link type and the name of the link type.

A **link type** is essentially defined by:

- a sequence of key attributes
- a set of non-key attributes

- a set of admissible destination object types
- a category

The category of a link type determines certain semantic properties of links of this type. One such category is 'composition'. Another category is 'reference'. A reference link expresses an association between the two objects, but does not possess any additional system-defined semantics.

Subtypes of an admissible destination object type are implicitly also admissible destination object types of this link type.

Modification of Schemata. Many OMSs provide operations by which ***applications*** can create new type definitions or modify or delete existing type definitions. In other words, applications can read and modify the conceptual schema (provided that appropriate rights are granted). Typical examples of modifications of existing type definitions are: an attribute is added to, or removed from, the set of attributes of an object type or link type; a link type is added to, or removed from, the set of admissible types of outgoing links of an object type.

2.5.2 An Example

The example that will be used throughout this paper is an object type representing modules (including their inner modules), with attributes and component objects as shown below. We denote type definitions using the notation

```
type X = subtype of Y
with attribute A1; ... An;
with link L1; ... Lm;
end;
```

which expresses that object type `X` is defined as subtype of `Y` with the additional attributes `A1` ... `An` and with the additional admissible outgoing link types `L1` ... `Lm`. Each link type definition specifies the link type name, key attributes (if any), category and admissible destination object types.

```
type Module = subtype of Object
with attribute
   ReviewResult : string;
   CompletionDeadline : date;
   HoursSpent : real;
   HourlyRate : real;
   CustomerAccount : AccountNumber;
with link
   hasSpecification composition link
      to Specification;
   hasSourceProgram composition link
      to SourceProgram;
   hasInnerModule [ModuleName]
      composition link to Module;
   ...
end;
```

```
type SourceProgram = subtype of Object
with attribute
   Author : string;
   ProgramText : string;
with link
   isSpecifiedBy reference link
      to Specification;
   ...
end;

type AdaProgram = subtype of
   SourceProgram
with attribute
   PackageNames : string;
   ...
end;
```

The main aim of the type-level DAC is to support roles such as designer, programmer, reviewer, manager etc. If we take another look at the above example, it should be obvious that the attributes or component objects relate to different roles. In other words, *complex objects may contain data related to several roles*, and users working in these roles access *the same complex objects*. However, it is only role-specific parts that are accessed, not the entire object. Typical examples of type-level access restrictions are:

- Only designers are allowed to create or modify the `Specification` component of a `Module`.
- Only reviewers are allowed to write the **`ReviewResult`** attribute of a **`Module`**.

These rules apply recursively to a module and to all its inner modules, i.e. to an object and to all its component objects.

Some attributes may even be invisible to some subjects, e.g. the attributes **`Hours-Spent`**, **`HourlyRate`**, and **`CustomerAccount`** of **`Module`** may be invisible to the designers and reviewers. Thus, the type **`Module`** should look to reviewers as follows (the allowed access modes are shown in brackets)[5]:

```
EXTERNAL SCHEMA for Reviewers:

type Module = subtype of Object
with attribute
   ReviewResult : (read,write) string;
   CompletionDeadline : (read) date;
with link
   hasSpecification (navigate)
      composition link to
      Specification;
```

[5]Note that the concept which is presented in this paper does *not* imply a specific view definition language; it only contains a model on which the *semantics* of a view definition language can be based. A compiler can translate the above definition of an external schema into modifications of the state of the object base (s. section 3).

```
   hasSourceProgram (navigate)
      composition link to
      SourceProgram;
   hasInnerModule [ModuleName]
      (navigate) composition link
      to Module;
end;
```

2.5.3 Main Problems

Combination of External Schemata. We have seen in section 2.3 that a process will generally have several active subjects. Typically, each subject will have an associated external schema which this subject can exploit. The type-rights contained in the external schemata of the active subjects must basically be "added" (in order for rights to be inherited from other groups). This leads to the question of how such an "addition" can be defined.

Conventional view mechanisms do not support nested working groups: they assume only one active subject at a time; if several subjects are active and if their external schemata contain different definitions of the same type, it is unclear how these different views upon the same type should be combined.

Additive Type-Rights. Frequently, a subgroup must be granted a set of *additive type rights*, assuming that the instance rights are inherited from a supergroup. In our above example, all members of a project may have the instance right to write a hierarchy of `Modules`, but they have only the type right to read the attribute instances appearing in that hierarchy. In sum, they can only read. The additive type right to write the attribute `ReviewResult` is only given to the subgroup of reviewers of modules. In concepts (such as relational views) where instance rights and type rights are always tightly coupled, one would have to specify the instance rights again for the subgroup. This would be very inconvenient, or not practicable at all, and would in fact make the inheritance of rights from supergroups useless. We can conclude that it must be possible to *grant type rights independently of instance rights.*

Conflicting roles. Different groups can correspond to "conflicting" roles which exclude one another, for example the producers and reviewers of a document. Such roles will use different external schemata. If a user plays different roles in different projects, this user may, in principle, have the right to exploit these external schemata (but in fact only in connection with different data instances). Thus, there must be ways in which the parallel exploitation of "conflicting" external schemata or their use with the "wrong" objects can be prevented.

Visibility of Instances of Subtypes. In all object-oriented languages and systems, an instance of a type can always be used as an instance of any of its supertypes. Thus, if *ot1* is a subtype of *ot*, an instance of *ot1* can be used whereever an instance of *ot* is required; *ot1* is type-compatible with *ot* in this kind of situation.

A question that arises is whether the analogous approach should be taken with regard to visibility. Let *ot1* be an *invisible* subtype of the *visible* type *ot*. Should instances of *ot1* be visible as instances of *ot*? The answer to the above question will depend on whether objects of type *ot1* require higher security than objects of type *ot*.

Assume again the object type `SourceProgram` with the attribute `ProgramText` and a subtype `AdaProgram`. Then only the group 'AdaProgrammers' might have the right to write the `ProgramText` of an `AdaProgram`. In this case, the answer is "no".

On the other hand, the project secretary should be able to print any `ProgramText`, regardless of the subtype in which it appears. In this case, then, the answer is "yes" (this has been called **inheritance policy P1** in [La&90]).

Thus, *both cases must be supported.* Inheritance policy P1 alone is not sufficient.

2.6 Metadata and Metabase

Type definitions are data about data, i.e. **metadata**. Metadata occur in external schemata and in the conceptual schema. Each OMS provides means of reading and modifying the conceptual schema; some OMSs provide means of reading the external schema. Obviously, a process must not be able to change the access rights contained in its external schema.

Most OMSs are *self-referential* in that they use a special part of the object base, called the **metabase**, in which the conceptual schema is represented by objects and links. In some OMSs, a type can be created (or modified) implicitly by creating (or modifying) an object in the metabase which represents this type. Other OMSs provide specific operations for the creation and modification of types.

2.6.1 Main Problems

The problems relating to the distribution of metadata have already been discussed in section 2.2.1. Another problem is that metadata must, in general, be protected. Type definitions as such can be secret. Unauthorized modification or use of metadata can cause considerable damage.

Protection of the Conceptual Schema. "Normal" applications should not have access to the conceptual schema (or to the metabase) at all. This requirement mainly concerns the ways in which the programming interface of the OMS is designed. It is not met by systems in which, e.g., the operation that creates an object of type *ot* requires as one of its input parameters the surrogate of the metabase object which represents *ot*. In such systems, users need to be able to scan the metabase when performing elementary operations, and one cannot keep metadata completely secret. For reasons of space, this problem will not be discussed further in this paper. A solution is presented in [Ke91, Ke92].

Objects in the metabase can be protected by instance-level DAC.

Information about the conceptual schema might be obtained via the external schema; this problem will be discussed below.

Protection of the External Schema. One possible question is whether a process should be able to query its external schema and obtain the complete definitions and type rights of all visible types. This should, in fact, be allowed. This flow of information is no security problem because it does not convey new information to the process, but is indispensable for practical reasons (program testing, writing generic browsers etc.).

A more difficult problem is the following: Assume that a type *ot* is visible in an external schema. Should then *implicitly* (a) all supertypes (b) all subtypes of *ot in the conceptual schema* also be visible? If so, a process could get information about all supertypes and subtypes in the conceptual schema. The answer to both questions is therefore clearly "no". It must be possible to hide arbitrary supertypes or subtypes of *ot* in an external schema. *ot* must, of course, not inherit attributes from an invisible supertype.

3 Type-Level Access Right Determinations

This section will present an abstract model of the state of the object base with regard to type-level DAC. This state is modelled as a mapping

$$tard : S * T * M \rightarrow V$$

with *S* being the set of subjects, *T* being the set of type definition units, *M* being the set of access modes, and *V* being the set of access values. These sets will be defined below. The mapping *tard* can also be regarded as a set of quadruples

$$(s,t,m,v)$$

s being a subject, *t* being a type definition unit, *m* a mode, and *v* being a value which indicates whether *s* is to be allowed to access instances of *t* using operations in *m*. Such a quadruple is called a **type-level access right determination (TARD** for short). The object base contains, for each triple (s,t,m), exactly one TARD (s,t,m,v) with $v = tard(s,t,m)$.

There are various ways in which this abstract state can be implemented (implementation issues will be discussed only very briefly in section 5).

IARDs and TARDs are conceptually very similar (s. section 2.1.1). They refer to the same set of subjects and access values, therefore the same definition of the rights of a process can be used. IARDs and TARDs differ inevitably in the units of protection to which they refer, and in the set of access modes.

3.1 Type Definition Units

Type definition units (in short: **units**) are sensible "fractions" of type definitions which way be known, or unknown to a user and, more generally, for which rights can be controlled independently. The way in which a type definition is split into type definition units depends, obviously, on the data model of the OMS. The following are H-PCTE's type definition units and their denotations; they should be easily adaptable to any data model which is based upon the entity-relationship approach:

- each single object type **ot** and each single link type **lt** is a type definition unit. (Note that this does not include any information about attributes of the object or link type.)

- each single attribute **a** is a type definition unit. (Note that "the same" attribute appearing at several object types and/or link types is only one type definition unit.)
- for each object type *ot*, this object type together with all its subtypes form the unit **ot***, i.e. $ot^* = \{ot\} \cup \{ ot' \mid ot'$ is a direct or indirect subtype of $ot \}$. (Note that $ot \neq ot^*$ even if *ot* does not have subtypes; in this case, $ot^* = \{ot\} \neq ot$.)
- for an object type *ot* and an attribute *a*, the fact that *a* is "applied" to *ot* (and implicitly to every subtype of *ot*) constitutes the unit **appl(ot,a)**[6].
- for a link type *lt* and an attribute *a*, the fact that *a* is "applied" to *lt* constitutes the unit **appl(lt,a)**[7].
- for an object type *ot* and a link type *lt*, the fact that *ot* (and every subtype of *ot*) is a valid origin object type for *lt* constitutes the unit **orig(ot,lt)**.
- for a link type *lt* and an object type *ot*, the fact that *ot* (and every subtype of *ot*) is a valid destination object type for *lt* constitutes the unit **dest(lt,ot)**.

3.2 Access Modes

The different sorts of type definition units have different relevant access modes, as specified in the following table.

Unit	Relevant Access Modes
ot	owner, existence, create, delete
lt	owner, existence, create, delete, navigate
a	owner, read, write, append, execute
ot^*	owner, existence, create, delete
appl(ot,a)	existence
appl(lt,a)	existence
orig(ot,lt)	existence
dest(lt,ot)	existence

3.3 Access Values

There are three access values (see [Ke90, Sa89] for a detailed justification of a three-valued logic):

+	"granted"
?	"undefined" (neither granted nor explicitly denied)
-	"denied"

[6]The definitions of *appl(ot,a)*, *orig(ot,lt)* and *dest(lt,ot)* assume that a subtype of a type inherits all properties of this type, in particular its attributes and the link types of which it is an admissible origin or destination type. Any other definition would be entirely inconsistent with the general philosophy of object-oriented systems whereby each object can also be regarded as an instance of any supertype of its type.

[7]We assume that link types do not have subtypes. Otherwise, a unit lt^* needs to be introduced and *appl(lt,a)* must implicitly include any subtype of *lt*.

3.4 Consistency Rules

There are certain consistency rules between TARDs. They have two main motivations: the nesting of type definition units and integrity constraints.

Nesting of Type Definition Units. Some type definition units are nested due to subtyping. Assume that *ot2* is a subtype of *ot1*. Then the following are **subunits** of each other:

- *ot1* is a subunit of *ot1**.
- *ot2** is a subunit of *ot1**.
- *appl(ot2,a)* is a subunit of *appl(ot1,a)*.
- *orig(ot2,lt)* is a subunit of *orig(ot1,lt)*.
- *dest(lt,ot2)* is a subunit of *dest(lt,ot1)*.

Due to multiple inheritance, units can *overlap*: *ot1** and *ot3** have a common subunit if *ot1* and *ot3* have a common subtype *ot2*. Two TARDs (*s,ot1*,m*,+) and (*s,ot3*,m*,-) would then be semantically inconsistent since they would imply contradictory TARDs for *ot2*. Similar problems arise for the other subunits. Therefore the following *consistency rule for subunits* is necessary:

If t2 is a subunit of t1 then

- *tard(s,t1,m)*='+' $\Rightarrow$ *tard(s,t2,m)*='+'
- *tard(s,t1,m)*='-' $\Rightarrow$ *tard(s,t2,m)*='-'

This consistency rule is the only *built-in policy* in our model. Similar consistency rules appear in some instance-level DAC concepts [Ke90, Ra&90]. Note that our TARDs are *explicit*. Our concept could be extended by implicit TARDs along the lines of [Br91, La&90, Ra&90].

A TARD for some $ot^* \in OT^*$ *is valid for all its subunits*. This allows one to implement a security policy where all instances of a subtype of a type *ot* are visible as instances of *ot* (policy P1 in [La&90]).

Integrity Constraints. The data model of H-PCTE has (like most other OMSs) a number of inherent integrity constraints. For example, a link cannot exist without its reverse link. Thus, when a link is created, its reverse link must also be created. It does therefore not make sense to grant the create-right for a link type *lt*, but not for the (unique) reverse link type *ltr*. This is prevented by the consistence rule:

- *tard(s,lt,m)*='+' $\Rightarrow$ *tard(s,ltr,m)*='+'
- *tard(s,lt,m)*='-' $\Rightarrow$ *tard(s,ltr,m)*='-'

3.5 Operations on TARDs

There are operations which set or read TARDs. The **set** operation propagates changes of TARDs to subunits or superunits whenever this is necessary due to the consistency rule. Attempts to modify TARDs or the subtype structure, which, by the above consistence rule, would lead to an inconsistency, are rejected by the system.

4 Type-Rights of a Process

This section presents a formal model of the type-rights contained in the external schema of a process, defines how they are computed from the state of the object base, and explains how they are interpreted during accesses to the object base.

The type-rights of a process are, conceptually, a mapping

$$has\_type\_right : T * M \rightarrow Boolean$$

with T being the set of all the type definition units and M being the set of access modes, as defined above. If *has_type_right(t,m)* $=$ *true* for a process, then we say that the process *has the m-right on t.*

4.1 Evaluation of TARDs

The type rights of a process are derived from the TARDs in the object base as follows:

A process acts generally on behalf of a set of *active* subjects (see also section 2.3). A TARD *(s,t,m,v)* is active for a process iff subject s is active for this process. The TARDs of *all active subjects* are "added" according to the following formula:

has_type_right(t,m) := *true* if

- there is an active TARD *(s,t,m,+)*
- there is *no* active TARD *(s,t,m,-)*

The above "formula" which defines *has_type_right(t,m)* can be implemented in various ways. In the H-PCTE prototype, for example, all type rights of a process are computed and stored in a cache when the external schema of a process is set. Other systems may use other schemes.

4.2 Interpretation of the Modes

The definition of a type, as seen by a process, is composed from the type definition units which the process can "see", i.e. on which it has the existence-right. The existence-right on the different type definition units is necessary

- to be able to "see" instances of object or link types; the existence-right on an object type is also necessary for navigating over objects of this type
- to know that an attribute is applied to a (visible) object or link type; then the attribute is visible at the object or link type; no additional existence-right for the attribute itself is necessary
- to know that an object type is a valid origin or destination type of a link type

The create- or delete-right is necessary for creating or deleting instances of object or link types.

Modes 'read' and 'write' are obvious. Modes 'append' and 'execute' are only relevant for string (i.e. long field) attributes. The execute-right is necessary for loading and executing an executable program stored in the string.

The owner-right is necessary for changing the set of TARDs on a unit, except for units of the type *appl*, *orig* and *dest*. For the latter units, the owner-right on the involved object type, link type and/or attribute is necessary.

Note that the administration of type-rights is decentralized in the same way as the administration of instance-rights.

5 Implementation in H-PCTE

Our model can be implemented in a variety of ways. For reasons of space, we will only very briefly discuss how it has been implemented in H-PCTE. A more detailed presentation can be found in [Ke91].

H-PCTE uses a metabase in which all object types, link types and attributes are represented by objects. A subtype relationship between two object types is represented by a relationship between the objects representing these object types. If an attribute is applied to an object type (or link type) then there is a relationship between the objects representing the attribute and object type (or link type). Admissible origin and destination object types of link types are represented analogously. In sum, *each type definition unit is represented by an object or relationship.*

All TARDs for a type definition unit are encoded in an ACL, with access value ? being the default value. This ACL is stored at the object or link representing this unit. Metabase objects representing an object type *ot* have two type-level ACLs: one for *ot* and one for *ot**.

Objects which represent type definition units can be stored in different sites of the network. As a result, *the management of type rights is fully decentralized.*

6 Conclusion

This paper has presented a concept of type-level discretionary access controls for distributed OMSs, which is intended to complement instance-level DAC and to support roles in development projects. The concept consists of a formal model of the state of the object base with regard to access controls and a formula which computes from this state and the security context of a process the type-rights of this process. The most important features of our model are:

- It has virtually no built-in, enforced policies, thus it allows us to realize a large range of application-specific security policies. It directly supports inheritance policy P1 [La&90].
- It is group-oriented in that it supports nested working groups and inheritance of rights along group hierarchies. Access can be explicitly denied.
- It allows us to tightly couple instance-rights and type-rights or to specify type-rights independently of instance-rights.
- It is implementable in a distributed system. The administration of rights can be fully decentralized.

It is left as an exercise to the reader to check that the more detailed requirements listed in sections 2.2.1, 2.5.3 and 2.6.1 are actually fulfilled by our concept.

7 Discussion

7.1 Other Approaches

Views or type-level access controls are an aspect of OODBMSs which has been widely neglected so far: most OODBMSs do not have them (many even have no access controls at all) and are therefore unacceptable in many cases [De&91].

Concepts for views or type-level DAC must, of course, be tightly integrated within the data model, they depend in particular on the notion of an (object) type and the notion of inheritance, and also on the data manipulation operations (e.g. relational vs. navigational access). Since most OODBMSs differ quite substantially in this respect, most of the related DAC concepts published so far are not directly comparable with each other or with our concept.

The only type-level DAC concept in a directly comparable data model appears in PCTE+ [PCTE+88, PCTE90]. PCTE+ uses an indirect, highly complicated mechanism for managing type rights (which cannot be presented here for reasons of space); most details of the type-level DAC are entirely different from the instance-level DAC. The concept in PCTE+ is also group-oriented and the protection achievable is more or less the same as in our concept, with the following notable exceptions (some of which tend to weaken the degree of security achievable in practice):

- It is not possible to explicitly deny access.
- It is not possible to tightly couple a set of instance rights and a set of type rights as in relational views, so that they can only be exploited together. (In H-PCTE, one can readily achieve this by granting both sets of rights to one subject.)
- The administration of rights is based upon the principle of delegation (instead of ownership) without any possibility of transitively revoking delegated rights.
- PCTE+ has only a fixed inheritance policy (P1).
- Only one group paradigm (the rights package paradigm, see [Ke90a]) is supported. However the task paradigm, needs to be supported as well. The absence of this support has led to certain exceptions to the type-level access controls, i.e. to "holes" in the security system. The consequences of this are exemplified by the fact that a process can often create objects of a type *ot*, although it does not have the create-right on *ot*.

Several other view concepts have been designed for a significantly different type of OODBMS, namely "relational" OODBMSs (e.g. [Br91, La&90, Ra&90]): these OODBMSs are based upon the relational model, assume set-valued, ad-hoc queries, have data-dependent views, while not having general complex objects with recursive types. These concepts either do not consider distribution at all or assume a conventional, "relational" distribution model. They aim at performing access controls at query translation time (resulting in query modifications), rather than within each access to an object.

These conditions appear to be the reason why all the view concepts mentioned above have only *one* level of access controls. A type is treated as the set of all instances (i.e. as a class), that is access to a type implies (!!) access to all instances;

this is not acceptable at all for our purposes. Thus, type-level and instance-level access control are not orthogonal to each other as they are in H-PCTE. All concepts mentioned above have only one fixed inheritance policy (P1).

7.2 Application to other Data Models

Our concept can easily be transferred to behaviourally OODBMS (s. section 1). The basic constituents of our concept (two-level access controls, group-orientation, type rights in an external schema, TARDs etc.) can be transferred in a rather straightforward way to behaviourally OODBMSs[8]: Broadly speaking, one can regard an access to an attribute instance of an object in H-PCTE and the execution of a type-specific operation on an object as corresponding to each other. Thus, the visibility of attributes corresponds to the visibility of operations of an object type. However, the modes read, write, and append are not applicable to operations because they can only be executed. If relationships between objects are modelled as surrogate-valued attributes, all features related to links in our concept can be dropped.

References

[Br91] Brüggemann, H.H.: Rights in an object-oriented environment; internal report, Universität Hildesheim; 1990/10 (also to appear in: Database Security V (Proc. 5th IFIP WG 11.3 Workshop, Shepherdstone, West Virginia, Nov. 1991); 1991/11)

[CAIS88] Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Revision A; DoD-STD-1838A; 1988/05

[DDMR91] DDM Requirements - Draft Proposal; CAD Framework Initiative, Inc., Austin TX; 1991/05

[De&91] Dewal, S.; et al.: Evaluation of object management systems for software development environments (in German); p.404-411 in: Proc. BTW 91; Informatik-Fachberichte 270, Springer Verlag; 1991/03

[DiHP88] Dittrich, K.R.; Härtig, M.; Pfefferle, H.: Discretionary access control in structurally object-oriented database systems; p. 105-121 in: Landwehr, C.E. (ed.): Database security II: status and prospects (Proc. 2. Workshop IFIP WG 11.3, Kingston, Ontario, 5.-7. Oct. 1988); Elsevier Science Publ.; 1989

[EURAC89] Requirements and design criteria for tool support interface (Version 4); IEPG TA 13 (PCTE+/EURAC); 1989/01/13

[FeGS89] Fernandez, E.B.; Gudes, E.; Song, H.: A security model for object-oriented databases; p.110-115 in: Proc. IEEE Symp. on Security and Privacy, Oakland, California; 1988/04

[GrS87] Greif, I.; Sarin, S.: Data sharing in group work; ACM TOIS 5:2, p.187-211; 1987/04

[8]Note, however, that the implementation of type-specific operations in behaviourally OODBMSs leads to additional problems, which are discussed in [Sp89].

[ITS90] Information Technology Security Evaluation Criteria (ITSEC) Harmonized Criteria of France, Germany, the Nederlands, the United Kingdom (Version 1); Der Bundesminister des Inneren, Bonn; 1990/05/02

[Ke90] Kelter, U.: Group-oriented discretionary access controls for distributed structurally object-oriented database systems; p.23-33 in: Proc. European Symposium on Research in Computer Security, Toulouse, October 24-26; AFCET; 1990/10

[Ke90a] Kelter, U.: Group paradigms in discretionary access controls for object management systems; p.219-233 in: Long, F. (ed.): Software Engineering Environments. Proc. Ada Europe International Workshop on Environments, Chinon, September 1989; LNiCS 467, Springer Verlag; 1990

[Ke91] Kelter, U.: Views in H-PCTE; University of Hagen, Dep. Computer Science, Informatik Berichte 113; 1991/06

[Ke92] Kelter, U.: Distribution of Schemata in H-PCTE; International Workshop on Distributed Object Management, August 19-21, 1992, Edmonton, Canada; 1992

[La&90] Larrondo-Petrie, M.M.; Gudes, E.; Song, H.; Fernandez, E.B.: Security policies in object-oriented databases; p.257-268 in: Spooner, D.L.; Landwehr, C.E. (ed.s): Database security III: status and prospects (Proc. 3. IFIP WG 11.3 Workshop, Monterey, California, 5.-7. Sept. 1989); Elsevier Science Publ. B.V.; 1990

[LuF90] Lunt, T.F.; Fernandez, E.B.: Database security; IEEE Data Engineering Bulletin 13:4, p.53-50; 1990/12 (appears also in: ACM SIGMOD RECORD 19:4, p.90-97; 1990/12)

[PCTE+88] PCTE+ Functional Specification, Issue 3; IEPG TA-13; 1988/10/28

[PCTE90] Portable Common Tool Environment - Abstract Specification (Standard ECMA-149); European Computer Manufacturers Association, Geneva; 1990

[PCTE91] Portable Common Tool Environment - C Bindings (Standard ECMA-158); European Computer Manufacturers Association, Geneva; 1991

[Ra&90] Rabitti, F.; Bertino, E.; Kim, W.; Woelk, D.: A model of authorization for next-generation database systems; ACM ToDS 16:1, p.88-131; 1991/03

[Sa89] Satyanayaranan, M.: Integrating security in a large distributed system; ACM Trans. Comp. Systems 7:3, p.247-280; 1989

[Sp89] Spooner, D.L.: The impact of inheritance on security in object-oriented database systems; p.141-150 in: Landwehr, C.E. (ed.): Database security II: status and prospects (Proc. 2. Workshop IFIP WG 11.3, Kingston, Ontario, 5.-7. Oct. 1988); Elsevier Science Publ.; 1989

[UnS90] Unland, R., Schlageter, G.: Object-oriented database systems: concepts and perspectives; p.154-197 in: Blaser, A. (ed.): Database systems of the 90s; LNiCS 466, Springer; 1990

[Vo91] Vossen, G.: Bibliography on object-oriented database management; SIGMOD Record 20:1, p.24-46; 1991/03