

ACCESS CONTROL MECHANISM IN BEHAVIORALLY OBJECT-ORIENTED DATABASES

Cristian Radu Mark Vandenwauver René Govaerts Joos Vandewalle

Katholieke Universiteit Leuven, Laboratorium ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium

Abstract

Providing security for a general-purpose database system is an expensive operation. An access control mechanism has to offer the level of security the people are willing to pay for. The behaviorally object oriented database management systems have features which can be used to achieve this goal. We design a mechanism able to attach reliable, albeit expensive, security means to individual objects, when this is appropriate. The mechanism uses both object protection and subject restriction to enforce various security policies.

1 Introduction

The conditional access of users to a database containing sensitive information is decided in the framework of an authorization model. This model represents a system of rights, specifying who is allowed to do which action on what part of the database. The rights are established according to a set of security policies. They are dictated by user needs, the installation environment and the existing legal constraints. The *access control mechanism*, also referred to as the *security mechanism*, is a set of functions that are used to enforce the various security policies. These functions can be implemented in software, hardware or by administrative procedures [2].

In this paper we present an access control mechanism for behaviorally object oriented database management systems (*bOMS*). A software implementation is outlined. For the sake of concreteness we often refer to the ONTOS database system, but the solution can be adapted to other similar database systems as well. We use an authorization model, presented in [1], consistent with the data model of this type of databases. On the one hand, we restrict our presentation to the database granularity hierarchy dimension of this authorization model. On the other hand, we use only explicit strong positive and negative authorization rules. The solution we suggest is based on the features of encapsulation, communication among objects through

method invocation and (single or multiple) inheritance. These features characterize a behaviorally object oriented environment. The paper is organized as follows: Section 2 reviews the main object oriented concepts. It explains three different hierarchies in the object-oriented database model. In this context a taxonomy of the object oriented databases is stressed out. Section 3 evaluates the useful features of the *bOMS* in relation with security targets. Section 4 details the object view and subject view security mechanisms. The implementation of these mechanisms in the framework of the chosen authorization model and tailored to the particularities of ONTOS object management system is presented in Section 5. Finally, we conclude our paper and present possible future work.

2 Review of Object-Oriented Concepts

In both programming languages and database management systems based on the object oriented paradigm, an entity is uniformly modeled as an object. Therefore, an object database can be seen as a collection of objects. Any object is associated with a unique identifier.

An object has the *encapsulation* feature. This implies that the object consists of two parts: the interface to the object and the implementation of the object [3]. The interface part is realized by a set of operations defined on the object and fully specifies its behavior. It is a description of the unique way in which users view the object. The implementation part of an object is only visible to its implementor and thus is hidden. It consists of:

- the internal representation of the object, also known as the data part or the state of an object. It contains the values for the *attributes* of an object.
- the implementation of each operation, detailing the behavior of an object. It is a set of *methods* that operate on the state of an object.

Using the C++ terminology, all of the objects with the same set of attributes and methods are grouped into a higher level object called a *class*. We can see an object as an *instance* of a class. If we refer to ONTOS, the classes describing the object database schema are denotable. They can be treated as instances of a metaclass, named *Type*. Then, a class can be also referred to as an *object class*. Considering the above remarks, we use the term *abstract data type (adt)* anytime we consider the template from which an instance was generated. We denote an instance with respect to an abstract data type by *iadt*. Any *iadt* belongs to one and only one *adt*. For example, in C++, a class can be seen as an *adt* and an object as an *iadt*. In the same way, in ONTOS, a class of the database schema is an *iadt* with regard to the *adt Type*.

We can define a first fundamental hierarchy in the *bOMS*, defining how objects are organized in terms of other objects. This hierarchy is named the database granularity hierarchy. Given an object *o*, this hierarchy defines how *o* is composed

of objects of lower levels. Each *adt* object class has its set of *iadt* instances. Any *iadt* instance has a set of values for its attributes.

An attribute of an object may take on a single value or a set of values. The value of an attribute of an object is also an object in his own right and thus belongs to a class. This class is called the *domain* of the attribute of an object. The fact that the domain of an attribute may be an arbitrary class, causes the nested structure of the definition of a class. In this way we obtain the class composition hierarchy. It shows how classes are defined in terms of other classes. This is a horizontal hierarchy describing the structure of an object. It is the main view of the structurally object oriented databases, according to Dittrich's taxonomy [4].

The inheritance feature allows one to declare a class as a specialization of another class. The former is named the *subclass* of the latter, also known as the *superclass*. The user may specify additional attributes and methods for the subclass or may even override old attributes and methods. A class may have any number of subclasses. Some systems allow a class to have only one superclass from which it inherits its attributes and methods. This is the *single inheritance* feature, leading to a graph named the class hierarchy. In a *multiple inheritance system* a class inherits attributes and methods from more than one superclass. The classes form a rooted, directed, acyclic graph, called the class lattice. The class hierarchy is related to the behaviorally object oriented databases. It represents an orthogonal view to the class composition and database granularity hierarchies. In *fully* object oriented systems, both class composition hierarchy and class hierarchy views can be considered, together with the database granularity hierarchy.

3 Security considerations

By the point of view of security requirements, the behaviorally object oriented database management systems have suitable features. We mention here the security implications of the following features.

3.1 The encapsulation and the communication paradigm

An initial layer of protection is provided by the fact that all the data is stored as values for the attribute variables. These are encapsulated inside objects and available only through the methods defined for the corresponding object class. Therefore, objects interact by only one well-defined communication paradigm, the invocation of methods. Methods are invoked by sending *messages* to an object (see Figure 1).

A message consists of a *command*, which selects the appropriate method, and some *arguments*, if necessary. The methods perform several kinds of activities, and they can be classified correspondingly as:

- constructors and destructors. They operate on the state of the class, in the sense that objects, as instances of the class, can be created or deleted.

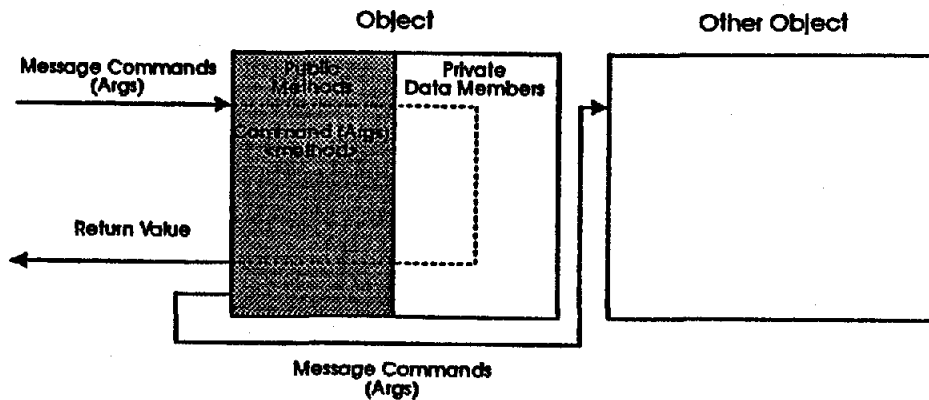


Figure 1: The Communication paradigm

- data status accessors. These methods may read and write (modify) the data members of the object where they reside.
- data storage/retrieve methods. The objects can persist beyond the application boundaries. During this time the object is stored in the object database, in a suitable form. Thus, the object can be retrieved in another application session that the one which created it. The data storage/retrieve methods are also named persistence-related methods.

3.2 The enriched semantic modeling capabilities

The "real world" and its security needs can be modeled more naturally using the enriched semantic modeling capabilities of the object paradigm. Object interfaces are much richer than the simple read/write/execute operations used in relation with files. There are as many operations as methods in the interface of an object. This allows a more problem-oriented expression of semantics. It also implies that the access rights have to be controlled in a correspondingly fine grained and problem-oriented way. This can be achieved by the direct control of the invocation of the methods.

3.3 The single and/or multiple inheritance

The ability to attach different security mechanisms and policies to individual objects is well supported by inheritance. This can be realized in two different ways [6]:

- statically - the security mechanisms are defined globally for all instances of the same abstract data type;
- dynamically - the security mechanisms are derived each time a single instance is created.

Multiple inheritance can be used to combine several security mechanisms. For example, access control lists and subject restriction lists mechanisms, can be simultaneously applied to the same object. The multilevel inheritance simplifies the specification of the security requirements and makes possible to build hierarchies of cooperating policies. Therefore, the security requirements of an application can be defined for object classes near the top of the inheritance hierarchy. These are then inherited by all the classes lower in the hierarchy.

4 Object/Subject View Access Control

The access control mechanism, shortly named the security mechanism, of a *bOMS* enforces a set of security policies. A security policy deals with attacks on the privacy and integrity of the information in the object database. Let S be the set of subjects, O the set of granule of the object database whose privacy and integrity have to be protected and A the set of possible actions or operations. The main function of the security mechanism is to decide if the subject $s \in S$ is allowed to execute the operation $a \in A$ with regard to the granule $o \in O$.

In a *bOMS*, applications are groups of instances that invoke each other operations. The security mechanism we present is based on the control of the invocation of these operations. It is important to distinguish between two categories of operations:

- *bOMS* operations - they are available at the level of the application program which manages the object database.
- internal operations - they represent the "atomic operations" on which relies the *bOMS* operations. The internal operations correspond to the methods provided in the interfaces of the object classes.

For example, the *bOMS* operation *modifyAttribute* modifies the value of an attribute of an object in the object database. The *objectName* and the *attributeName* are specified as parameters of the *bOMS* operation. The internal operations involved are the methods in the interface of the object class *className*. This represents the *adt* from which the *objectName* object was instantiated as an *iadt*. Of course, the methods in the interfaces of other object classes may be invoked as well. Thus, any *bOMS* operation can be derived from a set of methods which are invoked during the execution of an *bOMS* operation. This implies that *the security mechanism has to provide means to control the invocation of the methods*. This leads us to consider A as the union of all the methods offered in the interface of the object classes, describing the object database schema. For each message that can invoke a method we associate an unique number named *message identifier MID*.

We consider a subject s as an instance invoking a method. All the invoking instances can be traced-back to the user on behalf of which they act. Therefore an unique number referred to as the *user identifier UID* is ultimately attached to an

invoking instance. We consider the instance being invoked as the granule o for which protection is provided.

Two complementary security mechanisms can be combined.

4.1 The object view security mechanism

The object view security mechanism defines for a given instance $iadt_o$, which instances $iadt_s$, acting as subjects, may call which methods of the instance $iadt_o$, seen as the granule o being protected. The object view is implemented using access control lists acl . The acl of a granule o , denoted by acl_o , is expressed as a set of acl entries ($aclEntry$):

$$acl_o = \{aclEntry_i \mid i = 1, \dots, n_o\}$$

where n_o represents the total number of messages that can invoke a method in the interface of the object class from which o have been instantiated.

There are two fine boundaries between which one can specify entries in the acl . At the lowest boundary, the entries are related to an user or a group of users, and only offer a file-access semantic for the possible operations (e.g. read, write, execute). In this way UNIX provides its acl entries [5]. At the highest boundary, the entries are specified with respect to a user, a group of users, an abstract data type(adt) and even individual instances ($iadt$) of an abstract data type. The operations which are considered are the methods in the interfaces of the abstract data types. This is the approach used in the framework of the BIRLIX security architecture, described in [6].

We have chosen a "middle" approach. On the one hand, the UID is the real user ID of the current process where the instance acting as a subject was created. The real user ID identifies the person who is logged in. This identity corresponds to one of the following roles, accepted by UNIX:

- the owner of the object database ($owID$),
- the group to whom the owner belongs (gID),
- the other people (pID)

On the other hand we consider as possible operations all the methods provided in the interfaces of the object classes. A *grant* (+) or *deny* (-) permission is considered for each message invoking a method.

Therefore, an $aclEntry_i$ is a tuple:

$$aclEntry_i = (aclMessageName_i, aclSubjectRights_i)$$

where $aclMessageName_i$ specifies the MID of the message invoking a method, and $aclSubjectRights_i$ is defined as follows:

$$\begin{aligned} aclSubjectRights_i &= \{Right_{i,j}\}_{j=1,2,3} \\ Right_{i,j} &= (UID_j, grant \mid deny), \\ UID_j &\in \{owID \mid gID \mid pID\} \end{aligned}$$

Let's assume that an instance acts as the subject s . It sends the message M in order to invoke method m on the object o . The following algorithm decides if the invocation of the method is granted or denied:

- the *aclMessageName* field of the access control list of the target object o , acl_o , is scanned. The $aclEntry_i$ corresponding to the invoking message M is derived.

$$acl_o.aclMessageName = M \implies aclEntry_i$$

- the user identifier UID of the subject s is looked for in the field UID_j of the rights $Right_{ij}$. The value of the field $grant | deny$ decides if the subject s may or may not invoke method m through the message M

$$Right_{ij}.UID_j = UID \implies (grant | deny)_j$$

If $(grant | deny)_j = '+'$ the subject s may invoke method m by sending the message M to the object o . Otherwise, the invocation is forbidden.

4.2 The subject view security mechanism

The subject view is associated with an instance $iadt_s$, acting as a subject. It specifies which other instances, seen as granule $iadt_o$, being protected, the instance $iadt_s$ may invoke operations on. The subject view mechanism is implemented using the subject restriction lists srl .

The subject view is implemented in order to guaranty the enforcement of the least privilege policy. When new subjects are considered in the system, they have to be restricted in their access rights to exactly the number of granule they need. If only the *acl* are supported by the system, whenever a new subject is added to the system, all *acls* have to be scanned if they would allow the subject to access the corresponding object.

An *srl* is associated with every subject in the system. An *srl* of a given subject s , (srl_s) is a set of *srl*-entries and a flag (F). The flag is a Boolean variable, yielding *True* if the new generated subject may access all the granule in the object database. If the flag is set *False* the subject is restricted to the granule specified in the set of *srl*-entries. The *srl_s* can be formalized as follows:

$$\begin{aligned} srl_s &= (srlSet, F) \\ srlSet &= \{srlEntry_i \mid i \in \{1, \dots, n\}\} \end{aligned}$$

A *srlEntry* in the *srlSet* is the name of an object in the object database which subject s can access.

5 Implementation Hints

In this section we use the terminology agreed in the framework of the ONTOS object database management system. ONTOS provides a reliable persistent storage facility for C++ objects. Thus, (1) it allows objects denoted by C++ program variables to have a lifetime that is longer than that of the program that created them, and (2) allows C++ programs to retrieve persistent objects (objects created by other programs) into their program variables [7]. We consider the object database *SocialGroup*. It stores objects which model persons in a social environment. The user's database schema consists of only one Type denoted *Person*. A type is a kind of object that defines the abstract state and behavior of its instances. The abstract state of the type *Person* is represented by the properties as name, age, etc. The set of procedures describing the behavior of the type consists of: constructors/destructors, accessor to read and write the state of the objects. In ONTOS database schema Types correspond to C++ "persistent classes". The persistence of the class is provided by inheritance from a class belonging to the Kernel Database Schema of ONTOS, denoted by *Object*. In C++ the term *class* describes the state and behavior of objects by defining *data members* and *member functions* used by each instance of the class. The class describing the *Person* type is presented in the left side of Figure 2.

The application program that manages the object database defines *bOMS* operations like:

- *createObject(objName)* - it creates a new object in the database,
- *deleteObject(objName)* - it deletes an object from the database,
- *readAttribute(objName, attrName)* - it gets the value of the attribute *attrName* for the object denoted by *objName*,
- *modifyAttribute(objName, attrName, newValue)* - it modifies the value of the attribute *attrName* according to the value indicated by *newValue*.

Furthermore, an invocation of an *bOMS* operation, let's say *readAttribute(Person1, age)*, relies on the invocation of the method *intPerson :: get_age()* in the interface of the persistent class *Person* on the granule *Person1*. Therefore we have to control the invocation of the *get_age* accessor.

The main idea in order to achieve the access control mechanism is to transform the structure of the initial *Person* class. Thus the traditional constructors, destructors and accessors - which previous represented the behavior of this class - have to be committed as "protected". A new interface of the type class is defined, thus the signature of each method embeds a particular identification/authentication information about the subject. The code implementing the method contains a ramification statement which decides if the subject is authorized to call the method or not. If the subject is eligible, the method is executed by calling the corresponding protected method (which in fact act over the data member). Otherwise, the access to the protected

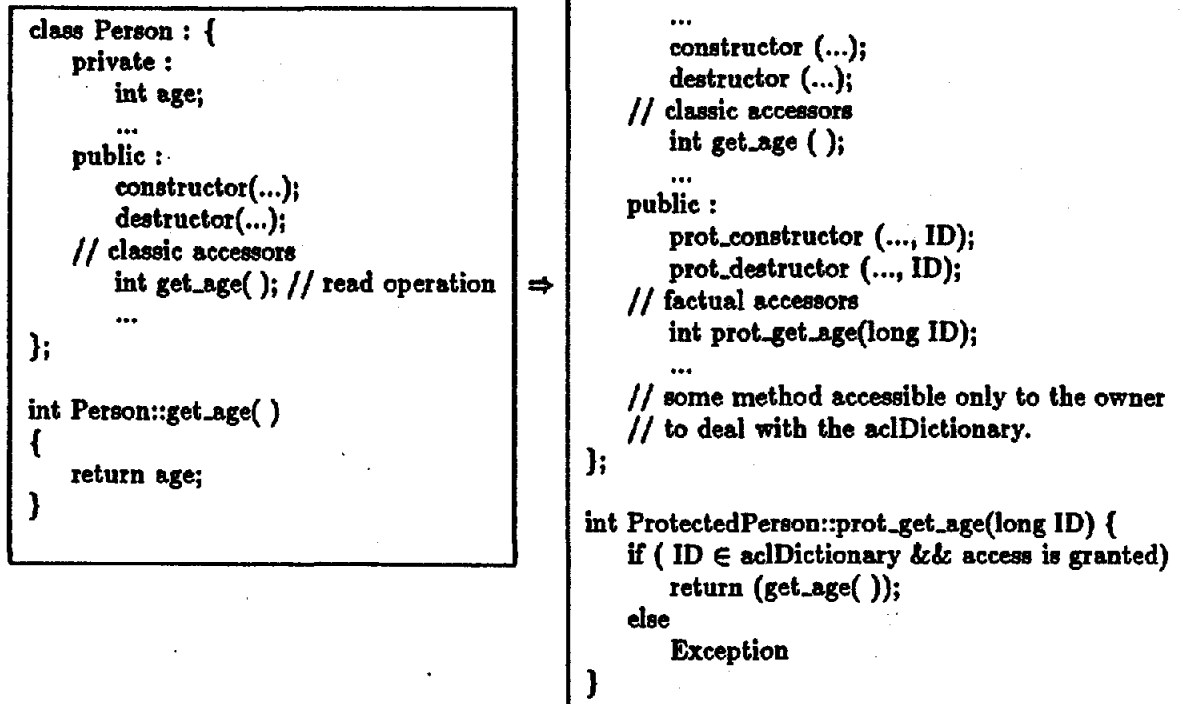


Figure 2: Pseudo code.

method is denied. At the same time an exception is risen, informing the subject about the refusal. This can also be recorded in the journal of access violations, which can be read by the administrator. The form of the resulting persistent class, that we have denoted by *ProtectedPerson*, is shown in the right side of Figure 2.

The authorization checking relies on a search operation in a dictionary of access rights. For this purpose a reference to a dictionary aggregate object is added to the protected data members of the type class. The dictionary is tagged with respect to the number of the messages which can invoke methods of the interface, and the element corresponding to each tag represents an access control list (acl) dedicated to that method. Each entry in the acl consists of the identifier of the subject and the permission(+) or denial(-) of the access to the corresponding method. The creation and deletion of this dictionary as well as the insertion and removing of tags and elements in the dictionary are executed only by the owner of the database. He controls the granting of access rights to other subjects. Correspondingly, the *bOMS* operations *SetAclEntry(objectName, SubjectID, AccessValues)* and *GetAclObject(objectName)* are provided at the level of the application program.

6 Conclusion

The security mechanism we have presented relies mainly on the encapsulation and communication paradigm between objects. These features characterize the behaviorally object-oriented databases. The implementation introduces a modality to access a hidden method through an identification part. The requesting subject is first looked for in the access list kept with the object. This list is realized by means of a dictionary. At the same time, each subject has a list of objects it has access to. We didn't stress out this part in the implementation, due to the complexity of the software solution herefore. This could be a subject for future research. We will also extend our work in order to cover both the class composition hierarchy and class hierarchy dimensions.

References

- [1] F. Rabitti, E. Bertino, W. Kim and D. Woelk, "A Model for Authorization for Next-Generation Database Systems," *ACM Transaction on Database Systems*, Vol. 16, No. 1, March 1991, pp. 88-131.
- [2] E.B. Fernandez, R.C. Summers and C. Wood, "Database Security and Integrity," Addison-Wesley, Reading, Mass., 1984.
- [3] R. Unland and G. Schlageter, "Object-Oriented Database Systems: Concepts and Perspectives," *Database Systems of the 90s*, A. Blaser (Ed.), LNICS 466, pp.154-197, Springer, 1990.
- [4] K.R. Dittrich, M. Härtrig and H. Pfeferlee, "Discretionary Access Control in Structurally Object-Oriented Database Systems," *Database Security, Status and Prospects*, C.E. Landwehr (Ed.), pp. 105-122, Elsevier Science Publisher B.V. (North-Holland).
- [5] D.A. Curry, "Improving the Security of your UNIX System," *SRI International Tech. Report ITSTD-71-FR-90-21*, April 1990.
- [6] H. Härtig, O. Kowalski and W. Kühnhauser, "The BIRLIX Security Architecture," *Journal of Computer Security*, No. 2, 1993, pp. 5-21.
- [7] "ONTOS Developer's Guide", *ONTOS, Inc.*, Release 2.1, 1991.