Access controls by object-oriented concepts

W. Essmayr[1], G. Pernul[2] and A M. Tjoa[3]
[1]Research Institute for Applied Knowledge Processing
Softwarepark Hagenberg
Hauptstraße 99, A-4232 Hagenberg, Austria
e-mail: we@faw.uni-linz.ac.at
[2]Department of Information Systems
University of Essen
Altendorfer Straße 97, D-45143 Essen, Germany
e-mail: pernul@wi-inf.uni-essen.de
[3]Institute of Software Technology
Technical University of Vienna
Resselgasse 3, A-1040 Vienna, Austria
e-mail: tjoa@ifs.tuwien.ac.at

Abstract

This paper introduces *object-oriented access controls* (OOAC) as a result of consequently applying the object-oriented paradigm for providing access controls in object and interoperable databases. OOAC includes: (1) subjects, like users, roles etc., are regarded as first-class objects, (2) objects are accessed by sending messages, and (3) access controls deal with controlling the flow of messages among objects. OOAC are not intended to replace legacy access control mechanisms which mainly have been designed and applied in non-object environments. Instead, they provide the basis for applying these concepts in true object-oriented environments. An *object authorization language* (OAL) is proposed for specifying authorizations in a declarative manner.

Keywords

security policy, object-orientation, access controls, interoperability

Database Security XI T.Y. Lin and Shelly Qian (Eds)

1 INTRODUCTION

The importance of object-oriented database systems increased dramatically within the commercial database market in the last few years. Especially, new application domains, like multimedia or interoperable environments, illustrate the feasibility and usefulness of object-orientation. Concerning access controls, most of the existing models, for instance *discretionary access controls* (DAC), *role-based access controls* (RBAC), or *mandatory access controls* (MAC), have been originally designed for relational database systems. However, the application of these models in object-oriented systems can not be straight forward (compare Bertino and Samarati, 1993). Particular object-oriented characteristics, like object identity, encapsulation, inheritance, polymorphism, and complex objects (see Atkinson et al., 1989) require the integration of new mechanisms to legacy access control concepts.

A remarkable amount of research has been devoted to extending access control concepts for object environments (see section 1.1). All this work offers the possibility to understand the challenges and research issues concerning access controls within object-oriented systems. However, we see the need for a common ground to start from in order to develop *true* object-oriented access controls. This is mainly because of two reasons: (1) to the best of our knowledge, none of the proposed extensions to legacy access controls consider subjects (e.g. users) to be first-class objects within an object database. Instead, subjects are treated as "extra-terrestrial" entities that are completely separated from the data objects. In consequence, object-oriented features can not be applied to subjects, respectively, access control concepts can not be applied within pure data object communications. On the other hand, (2) most of the proposed extensions still regard access types as a set of elementary actions (like *read*, *write*, *execute*, etc.). This attitude ignores *messages* to be the means of communication within an object system and dramatically limits the expressiveness of the security model. The set of messages an object is able to respond to (i.e. the object's interface) exactly defines the ways an object might be accessed; a set of elementary actions can never be complete for the great variety of application domains.

The remainder of the paper is structured as follows: Section 1.1 provides an overview of work related to this paper. Section 2 and its sub-sections introduce the concepts used by object-oriented access controls. Section 2.1 locates the prerequisites for OOAC, section 2.2 describes the message communication process, and section 2.3 concentrates on the object activity stack required for access controls within object systems. Section 2.4 proposes an object authorization language to formally express authorizations within OOAC. Section 2.5 specifies basic policies concerning authorization and access control within OOAC. Finally, section 3 concludes and provides directions for future research efforts.

1.1 Related Work

In Bertino et al. (1991), the authors present an authorization model for next-generation database systems supporting object-oriented concepts as well as semantic data modeling concepts. Special effort is given to the development of computing implicit authorizations from a set of explicitly defined authorizations. Bertino (1992) first suggests to specify privileges for users to execute methods on objects. The authorization model enforces the concept of private and protected methods. Fernandez et al. (1993) present a method-based authorization model as well as algorithms that evaluate the proposed authorization policies concerning generalization, aggregation, relationships, abstract classes and indirect method-calls. Bertino and Samarati (1993) summarize the issues arising within discretionary authorizations for object bases. The authors provide suggestions how to successively extend a basic authorization model with object-oriented features concerning access types, security subjects and objects, access controls, and authorization. Gal-Oz et al. (1993) concentrate on the evaluation algorithms deciding whether an access should be fully or partially granted/denied. The algorithms are discussed for compiled-time as well as run-time evaluation. Fernandez et al. (1994) develop an authorization model for object-oriented databases. The model contains user access controls and administration of authorizations. It consists of a set of policies, a structure for authorization rules and algorithms to evaluate access requests against the authorization rules. Jonscher et al. (1993) discuss means for object-oriented environments to reduce the number of explicit authorizations to the number of meaningful authorizations. In this work the concept of compound subjects is introduced in order to be able to model certain real world security requirements (i.e. the four eyes principle). Brüggemann (1991) provides another early but important contribution to authorizations in object-oriented environments. Essmayr et al. (1995) discuss role-based access controls in an interoperable environment, including object-oriented as well as relational database systems. The architecture of IRO-DB, a European ESPRIT project providing interoperable access to relational and object-oriented databases, has been first described by Gardarin et al. (1994). It uses a local, a communication, and an interoperable layer and allows for specifying a global object schemata across heterogeneous and distributed local database schemata. In order to show the feasibility and scalability of the concepts proposed within this paper, they will be applied to the IRO-DB database federation within a simplified application.

2 OBJECT-ORIENTED ACCESS CONTROLS (OOAC)

2.1 Prerequisites for OOAC

OOAC requires an object database to provide a class hierarchy having a distinct origin class and a sub-hierarchy of objects that could be referenced by name. Since not every object should be an issue for access controls OOAC assume that only access to *named* objects is being controlled. Some of the object databases explicitly distinguish between *persistent* objects (objects that survive the process within they have been created) and *transient* objects (objects that live only for the execution time of the process within they have been created). Some others implement the concept of *persistency by reference* (an object is persistent as long as it is referenced by at least one other object). Anyway, only *persistent* objects should be an issue for access controls since they are the asset to protect while transient objects 'die' after the scope of a process. In this paper we use the term *protection object* for objects that could be named and should be an issue for access controls.

The set of classes offered by an object database can be grouped into several schemata, for instance, a *basic* schema, a *security* sub-schema, an *application domain* sub-schema etc. A basic schema as required for OOAC contains a root class (*Object*) holding the object identifier and offering methods to create (*new*), *delete*, and *copy* objects. Furthermore, a root class for all named objects (*NamedObject*) has to be offered, providing methods to find (*lookup*) an object and to retrieve or change its *name*. Finally, some general purpose classes could be provided that are implementing common and useful functionality (e.g. collections, strings, etc.).

In a true object-oriented environment basically everything is regarded as an object. Hence, the active entities referred to as *subjects* in security literature (e.g. users, roles, etc.) are objects, too. Role-based access controls, for instance, needs a class *Subject* (sub-class of *NamedObject*) providing special behavior concerning activity (see section 3.3) common to all security subjects. Two kinds of subjects exist, namely, *users* and *roles*. Users have to be authenticated (using a password check) and can be members of several roles. A user has to *play* (activate) a role in order to receive certain authorizations. Roles may be structured within a role-hierarchy using *subroles/superrole* relationships.

An application sub-schema contains classes relevant to a particular application domain. In this paper, we use a simple example used within the IRO-DB project (compare Gardarin et al., 1994). It focuses on a production database that maintains *parts* which are distributed over two local databases.

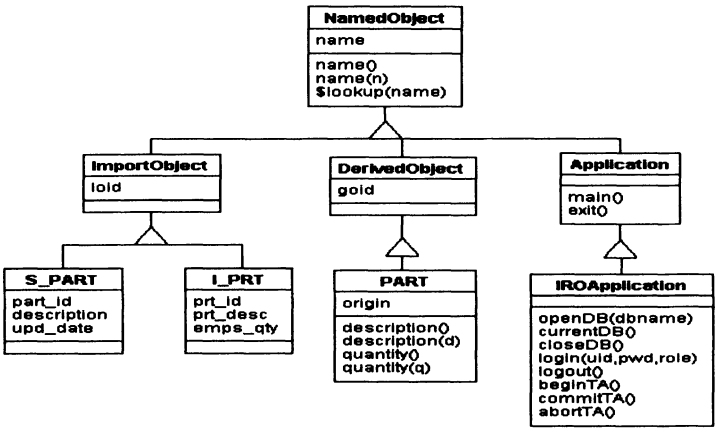

Figure 1 An example for an application sub-schema.

Basically, objects at the interoperable layer of IRO-DB may be either *native* objects defined within the home database of the interoperable layer, may be *imported* from a local database, or may be *derived* from native, imported, or other derived objects. Import objects additionally maintain a local object identifier (*loid*) that is composed of the original object identifier and some local site information. Derived objects additionally maintain a global object identifier (*goid*) that contains information about the classes from which the object is derived. The example schema in Figure 1 shows two import classes, namely *S_PART* and *I_PRT* with their objects in fact located at different local databases. Both classes maintain a part identifier (*S_PART.part_id*, *I_PRT.prt_id*) and a part description (*S_PART.description*, *I_PRT.prt_desc*) about logically identical parts. However, class *S_PART* additionally maintains an update date (*upd_date*), and class *I_PRT* additionally holds the quantity (*emps_qty*) of produced parts. The derived class *PART* provides a global view on both import classes allowing to retrieve and change the part *description* as well as the *quantity* of any part stored in the distributed local databases. The native class *IROApplication* provides functionality for opening and closing IRO-DB (*openDB*, *closeDB*), for transaction management (*beginTA*, *commitTA*, *closeTA*) and for identification and authentication (*login*, *logout*) of IRO-DB users.

2.2 Messages

Objects communicate in sending messages to themselves or to other objects which react in executing a particular method that shows behavior in that it may change the state of the object, alter parameter values or provide a return value. Object-oriented

access controls simply deal with controlling the flow of messages among objects of an object database.

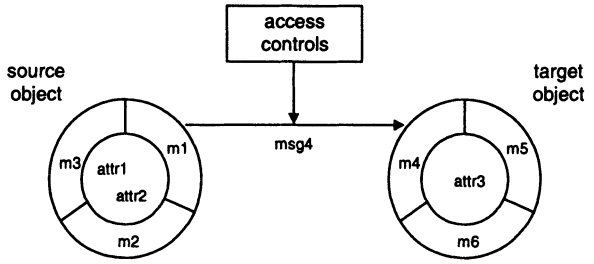

Figure 2 Controlling the flow of messages among objects.

Figure 2 illustrates the message sending and access control mechanisms. During the execution of method *m1* the source object sends message *msg4* to the target object which is expected to react in executing its corresponding method *m4*. OOAC intercepts the message sending process[1] and applies access controls in that it evaluates whether the source object is allowed to send the particular message to the target object (see section 2.5 for details about the evaluation process). If allowed, the message is sent and the target object executes the requested method. If denied, the message is blocked and an *access control exception* is raised[2]. The requested method is prevented from executing and its output parameter values as well as an optional result value are set to a distinct value, e.g. *nil*. Each application may catch the access control exceptions and handle them according to its needs. Several nested method calls may occur during program execution, e.g. *object1* sends *message1* to *object2* which in turn sends *message2* to *object3* etc. which leads to a stack of active objects as explained in the following sub-section.

2.3 Activity

OOAC require a system maintained *activity stack* which usually corresponds to the stack of method calls. Each time, an object receives a message and starts executing the corresponding method, the object becomes *active* and is pushed on to the activity stack. All subsequent messages are regarded to be sent from this object until it returns from executing the method and is popped again from the stack. The most recently activated object is the basis for access control decisions. If a

[1] Triggers could be an adequate mechanism to intercept the sending of messages among objects.

[2] We assume the existence of exceptions in the object database here since they are the most intuitive way to handle errors.

decisions is not possible the previously activated object is examined until the primarily activated object is reached which usually corresponds to a kind of default system object. Instances of class *Subject* or sub-classes of it are the only objects that have the possibility to *actively* modify the activity stack independently from the sequence of method invocations in one of the following ways:

- activate *on-behalf-of*: the subject is additionally pushed on to the activity stack which is the normal case like for any other object with the difference that the subject remains activated until it is explicitly deactivated (popped from the stack).
- activate *instead-of*: the subject replaces the previously activated object in the activity stack with the consequence that the authorizations of the replaced object do not further influence access control decisions.

Table 1 shows the changes to an activity stack during the execution of a simple example application that uses the schema illustrated in Figure 1. The IRO-DB application (IROApplication[1], the characters '[' and ']' denote instantiation, i.e. the object named "1" of class *IROApplication*) offers the following functionality: (1) identify and authenticate a user, (2) let the user play a particular role, (3) let the user choose a particular part, and (4) change the part description in both of the local databases from which parts are derived.

Table 1 Changes within the activity stack while executing an example application

Source Object (Activity Stack)	*Target Object*	*Message (Method Call Stack)*
system	IROApplication[1]	main(1,"1")
IROApplication[1]	IROApplication[1]	openDB("example")
IROApplication[1]	IROApplication[1]	login("7","abc","2")
IROApplication[1]	User[7]	authenticate("abc")
User[7]	User[7]	activate(OnBehalf)
User[7]	Role[2]	play
Role[2]	Role[2]	activate(InsteadOf)
Role[2]	PART[15]	description("Part15")
PART[15]	S_PART[15]	description.set("Part15")
PART[15]	I_PRT[15]	prt_desc.set("Part15")
Role[2]	IROApplication[1]	logout
IROApplication[1]	IROApplication[1]	closeDB
IROApplication[1]	IROApplication[1]	exit
system		

After the big-bang (i.e. the process that represents IROApplication[1] is loaded into memory) a default *system* object immediately passes control to IROApplication[1] in sending the message *main*. During the execution of method *main*, the application

first opens the IRO-DB database "example" and tries to *login* a user, i.e. it identifies User[7] and authenticates him/her using password "abc". The method *authenticate* actively changes the activity stack in that it lets an authenticated user work *on-behalf* of the calling application. Next, the method *login* identifies that the user wants to *play* Role[2] and sends the corresponding message *play* to the particular role. The method *play* actively changes the activity stack again in that it lets the requested role work *instead-of* the active user. Third, the application identifies that the user wants to modify the description of PART[15] to the value "Part15" and sends the corresponding message *description("Part15")* to the particular part object which in turn sets the description attributes of both import objects S_PART[15] and I_PRT[15] from which PART[15] has been derived.
After completing the task the application performs a *logout*, *closes* IRO-DB and *exits*, the corresponding process is terminated. The following sub-section goes into detail for specifying authorizations within OOAC and illustrates some means to facilitate this process, for instance, using template or conditional authorizations.

2.4 An Object Authorization Language (OAL)

Authorization in OOAC specifies the messages a source object may send to a target object. In order to describe authorizations in a declarative manner we propose an *object authorization language* (OAL). 3 kinds of authorizations are possible within OAL, namely *template*, *conditional*, and *negative* authorizations.

Template Authorization

The OAL contains mechanisms to relieve the administrational effort of specifying authorizations. The characters '*' (*any*) and '$' (*arbitrary*) can be used to specify sets of objects and/or messages respectively to specify an arbitrary object or message that can be referred to from other points within an authorization. The latter concept is especially useful for conditional authorizations. Using the template characters '*' and '$', an object, either source or target, may be specified in one of the ways shown in Figure 3. Both, the X and Y axis show the possible types of specifications which can be one of *particular* (by name), *arbitrary* (by character '$' followed by an identifier) or *any* (by character '*'). The X axis corresponds to object definitions, the Y axis to class definitions. Thus, a complete object specification may be a *particular object* of a *particular class* (AClass[inst]), an *arbitrary object* of a *particular class* (AClass[$inst]), *any object* of a *particular class* (AClass[*]), an *arbitrary object* of an *arbitrary class* ($AClass[$inst]), *any object* of an *arbitrary class* ($AClass[*]), and *any object* of *any class* (*).

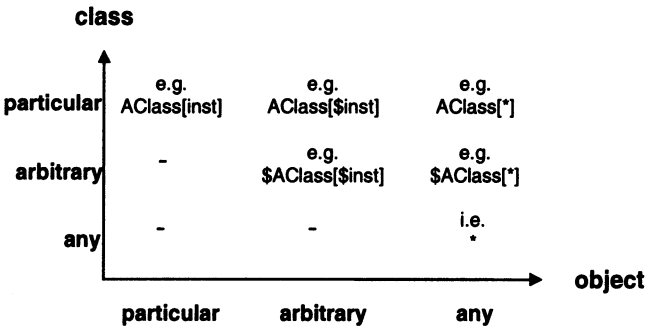

Figure 3 Template object specifications within OAL.

The OAL also allows to specify template messages. Thus, a complete message specification may be a *particular message* (e.g. $msg1(AClass[*],*), parameters may optionally be specified as a set of object specifications placed within parenthesis and separated with commata), an *arbitrary message* ($aMessage) of the object's interface, and *any message* (*) of the object's interface.
The authorizations necessary for the simple example application mentioned in the previous section can be expressed in OAL as follows:

```
1. ALLOW system SENDING main, exit TO Application[*];
2. ALLOW Application[*] SENDING authenticate TO User[*];
3. ALLOW User[$u] SENDING play TO User[$u].roles[*];
4. ALLOW Role[2] SENDING description(String) TO PART[*];
5. ALLOW PART[$p] SENDING * TO PART[$p].origin[*];
```

The set of authorizations specifies that the default *system* object may send messages *main* and *exit* to any application (1) which in turn may authenticate any of the known users (2). Each particular user may *play* those roles (s)he is a member of (3). Role[2] may change the *description* of any PART object (4), and a particular PART object may send any message to those objects the PART object has been derived from, i.e. the *origin* of the derived object (5).

Conditional Authorization

In many cases it is desirable to specify constraints on authorizations, for instance, dependencies on attribute values, time, location, or even on other authorizations (like within the concept of implied authorization, see for instance Bertino et al., 1991). OOAC allow to specify two kinds of conditional authorizations:

- dependent on *object state*, and
- dependent on *other authorizations*.

Suppose the following:

```
IF Date.now()<User[$u].expirationDate() THEN
    ALLOW User[$u] SENDING play TO Role[Subscriber];
END
```

In this case, the user objects are designed to have an attribute holding an expiration date. The message *expirationDate* returns this date which is then compared to the current date (message *now* is a class member of *Date* returning the current date). The authorization specifies that any user of the database is allowed to *play* Role[Subscriber] if the user's validation date has not expired.
The authorization below is dependent on another authorization and realizes a mutual exclusion constraint in that Role[2] is denied for what Role[1] is authorized to.

```
IF ALLOWED Role[1] SENDING $m TO $AClass[$inst] THEN
    DENY Role[2] SENDING $m TO $AClass[$inst];
END
```

Summarizing, the OAL provides possibilities to specify authorizations that depend on object state and/or other authorizations. The former allows to realize any kind of value dependency which is possible using the database objects and messages (including time, location, etc.). The latter can be used to realize particular policies e.g. for the concept of implied authorization or for mutually exclusive roles.

Negative Authorization

Usually, authorization models allow to specify positive authorizations (permissions) based on a *closed world* assumption saying that any access is denied unless explicitly permitted. Authorization models have been enriched with the concept of negative authorizations (prohibitions) for more flexibility combined with an *open world* assumption (any access is allowed unless explicitly prohibited). A mixture of positive as well as negative authorizations is feasible enabling to specify exceptions from general specifications.
Within OOAC, authorization may be either based on an *open* or a *closed* world assumption. Furthermore, an access may be either *allowed*, or *denied*, respectively, a previously allowed/denied access may be *revoked*. The sequence of authorizations is relevant with respect to conflicts either due to template authorizations or due to coexistence of permissions and prohibitions. Consider the following sequence of authorizations:

```
1. ALLOW User[*] SENDING description() TO PART[*];
2. DENY User[47] SENDING description() TO PART[*];
3. DENY User[*] SENDING description(String) TO PART[*];
4. ALLOW User[11] SENDING description(String) TO PART[*];
```

The sequence specifies that principally any user is *allowed* to retrieve the description of PART objects (1) except for User[47] (2). On the other hand, any user is principally *denied* to change the description of PART objects (3) with the exception of User[11] (4).

2.5 Authorization and Access Control

A set of policies characterizes the basic attitudes of OOAC. We try to minimize the number of policies in order to emphasize the flexibility offered by *template* and *conditional* authorizations. Nevertheless, some fundamental policies are specified in this sub-section which are useful and are intended to reflect the principle nature of object orientation.

Fundamental Policies

The first policy has already been mentioned before. It reduces the number of access controls within OOAC in providing a base class *NamedObject* which has to be sub-classed for objects that could be named and should be an issue for access controls (i.e. protection objects).

> (protection-object): access controls are applied to messages sent to named objects.

In case that a non-protection object initiates a message that object has to work *on-behalf* of a protection object in order to be possibly authorized for the particular message. Any message that is sent to a non-protection object can only be controlled at the programming language level.
The second fundamental policy serves the principle of encapsulation in that it lets any object use its own interface, freely.

> (object-interface): any object is allowed to send any possible message to itself.

This policy can also be disabled if desired in using the following template authorization: "DENY $AClass[$inst] SENDING *". The negative authorization does not specify a target object which is then supposed to be equal to the source object. It says that an arbitrary object of an arbitrary class must not send messages (defined for that class or inherited, see inheritance 1 and 2 policies below) to itself.

Object/Class Methods

A method may either be defined for individual objects or for all objects of a class at once. *Object methods* are the regular case and can be executed independently on any of the class' objects. Authorizations for object methods may include an *instantiation clause* ("[...]") and thus be specified for individual objects. A *class method* can only be executed on all objects of a class at once, not independently on a particular object of a class. An example for a class method could be *NamedObject.lookup* which takes a name and returns the corresponding object from within the extent of the regarded class, e.g. *User.lookup("user1")* returns the instance of class *User* that is named "user1" (the expression is equal to User[user1]

within the OAL). Consequently, an authorization for a class method does not allow to specify an *instantiation clause* for the target object within the OAL. However, not all of the object systems distinguish between object- and class-methods.

Overloading

Overloaded methods have the same name but different signatures within one class. Examples for overloaded methods are *name()* and *name(String)* of class *NamedObject* again taken from the basic schema. The former returns the name of an object, the latter allows to change the name of an object. Overloaded methods are distinguished using their signature, i.e. the number and types of parameters as well as the type of an optionally returned result. Note, that most object systems ignore the result type due to technical issues.

Inheritance

Methods may be inherited, that is, specified and implemented in a direct or indirect super-class of the regarded class. With respect to access controls, all methods that are inherited by an object are regarded as components of the object's interface which leads to the following two policies:

> (inheritance 1): authorizations may be specified for messages that are inherited from super-classes of the target object's class.

Thus, an authorization like "ALLOW User[1] SENDING name() TO DerivedObject[*]" could be specified to allow User[1] to retrieve the name of derived objects, for instance, although the particular method is defined and implemented in class *NamedObject*.

> (inheritance 2): an authorization to send *any* message defined for object *o* includes those messages that are inherited from any of the super-classes of *o*.

For instance, the authorization "ALLOW User[1] SENDING * TO PART[1]" allows User[1] to send any message defined for class *PART* (i.e. *description*, and *quantity*) as well as any message inherited from the super-classes (i.e. *name*, *lookup*, *new*, *copy*, and *delete*) to PART[1].

We do not want to go into detail for multiple inheritance since this is an optional feature of object databases (compare Atkinson et al., 1989). Nevertheless, if an object system supports multiple inheritance, OOAC has to deal with the possibility of ambiguous messages (i.e. messages that cannot clearly define which method of the super-classes has to be executed). Furthermore, an object may combine methods inherited from both, protection objects and non-protection objects, which requires to additionally control some messages sent to non-protection objects.

Polymorphism

Inherited methods may be *overridden*, i.e. the implementation of the method may be changed or extended without changing the name and signature of the method. The method that should be executed can be determined by examining the *type* of the regarded object. If the decision depends on the *dynamic* type of an object the concept is also called *late binding*, since the dynamic type can only be determined at run-time. The following two policies dealing with the polymorphism property of objects can thus be stated:

> (polymorphism 1): an authorization to send message *m* to object *o* additionally holds for any possible form (polymorphism) of *o* that can respond to *m*.

For instance, the authorization "ALLOW User[1] SENDING name() TO PART[1]" additionally allows User[1] to send *name()* to PART[1] in the form of a *DerivedObject* or a *NamedObject* instance since these super-classes of *PART* can respond to message *name()*.

> (polymorphism 2): an authorization to send message *m* to *any* object that is an instance of a particular class *C* include those objects that are instances of sub-classes of *C* which thus *could* be instances of *C* as well (due to polymorphism).

The authorization "ALLOW User[1] SENDING name() TO DerivedObject[*]" allows User [1] to send message *name()* to any object that is an instance of class *DerivedObject* (which in this case will be an empty set since *DerivedObject* is an abstract class). Additionally the message *name()* may be sent to those objects that *could* be instances of class *DerivedObject*, i.e. that are instances of one of the sub-classes of *DerivedObject*, namely *PART*.

Complex Objects

Complex objects are objects that are composed of simpler ones. In this paper, we assume an object system that completely encapsulates the attributes of an object, that is, the attributes can only be accessed by other objects in sending messages that can be controlled. Only the object itself is allowed to manipulate its attributes directly. Each attribute value in fact is a reference to an object. Some objects are collections and can be used to aggregate other objects (e.g. *Collection*). The concept is orthogonal in the sense that a collection may hold any object and thus may hold other collections, too.

In Essmayr et al. (1996), we proposed some policies concerning implied authorization for components of complex objects, saying that if a subject (e.g. a user) is authorized to access a complex object, the subject should be implicitly authorized to access each component of the object and thus the complex object as a

whole. OOAC do not use any automatic authorization mechanisms. Instead, each object has to be authorized for the actions it wants to execute which is orthogonal for subjects since they are themselves objects in OOAC. For retrieving the name of a derived PART object, for instance, the PART object has to be authorized to retrieve the name of any (import) object the PART has been derived from. An adequate authorization in OAL could be formulated as "ALLOW PART[$p] SENDING name() TO PART[$p].origin[*]" authorizing an arbitrary PART object to retrieve the name of its origin objects.
Another problem is that some object systems provide types that are system defined (e.g. int, float, etc.) and do not represent first-class objects. Furthermore, attributes might be declared public, that is, they may be directly accessed by other objects without using the object's interface. Assume the attribute *description* of class *S_PART* to be a public *String*. If strings are non-protection objects the policy concerning protection objects has to be extended as follows:

> (protection-object'): access controls are applied to messages sent to protection objects and to those non-protection objects that are declared as public parts of a protection object.

The following authorization could be specified which allows User[1] to assign descriptions to any object of the origin of PART[1]:

```
ALLOW User[1] SENDING set TO PART[1].origin[*].description;
```

System defined types that are declared as public attributes are handled by OOAC as quasi objects having the minimum interface *get* (the attribute value) and *set* (the attribute value) which allows to specify authorizations for these two messages.

3 CONCLUSIONS AND FUTURE WORK

In this paper we introduced a new concept for access controls that is especially tailored for true object-oriented environments. The concept called *object-oriented access controls* (OOAC) is based upon the following assumptions: (1) everything within the object-oriented environment is regarded as an object, (2) thus, security subjects (e.g. users, roles, etc.) are regarded as first-class objects, too, (3) messages are the only means for communicating to other objects.
In consequence, OOAC deal with controlling the flow of messages among the objects of an object database. An *object authorization language* (OAL) has been proposed that allows to specify the set of messages an object is allowed or denied to send to other objects in a declarative manner. The OAL provides means to specify *template* authorizations (relevant to a *set* of objects and/or messages), *conditional* authorizations (depending on object state or other authorizations) and *negative* authorizations (denying access rather than allowing it). Furthermore, we presented a minimal set of policies that corresponds to those properties commonly accepted to

be inherent to object-oriented systems. The policies address some characteristics concerning the object interface, inheritance, polymorphism, and complex objects. On the other hand, OOAC do not impose a particular kind of access control policy (e.g. discretionary, role-based, or mandatory access controls). Instead, any known policy or even any policy developed in future may be implemented using OOAC since the structure and behavior of database objects exactly determine the ways an object may be accessed as well as the ways an object can be protected against unauthorized access. The feasibility of OOAC has been demonstrated in applying the concept to IRO-DB, a database federation that provides interoperable access between relational and object-oriented database systems.

Future research efforts will concentrate on implementing ownership-based (e.g. DAC), role-based (RBAC) or mandatory (MAC) access controls within OOAC. Furthermore, administration issues will be addressed assuming the existence of meta classes (like *Class*, *Attribute*, *Message*, *Method*, *Schema*, *Authorization*, etc.) as parts of the application schema allowing to apply OOAC for controlling schema modifications and/or security administration in the same way as controlling simple object access.

4 REFERENCES

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S. (1989) The Object-Oriented Database System Manifesto. *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan.

Atwood, T., Duhl, J., Ferran, G., Loomis, M., and Wade, D. (1993) *The Object Database Standard: ODMG-93, Release 1.1*. Morgan Kaufmann Publishers, San Francisco, California.

Bertino, E., Kim, W., Rabitti, F. and Woelk, D. (1991) A Model of Authorization for Next-Generation Database Systems. *ACM ToDS*, Vol. 16/1.

Bertino, E., Martino, L. (1991) Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, April 1991, pp33-47.

Bertino, E. (1992) Data Hiding and Security in an Object-Oriented Database System. *Proc. 8th IEEE Int. Conf. on Data Engineering*, Phoenix, Arizona.

Bertino, E., Samarati, P. (1993) Research Issues in Discretionary Authorizations for Object Bases. *Workshop in Computing: B Thuraisingham, R. Sandhu, T.C. Ting (Eds.) Security for Object-Oriented Systems*, Washington DC.

Brüggemann, H. H., (1991) Rights in an object-oriented environment. *Proc. 5th IFIP 11.3 Working Conference on Database Security*. Shepherdstown, WV, 1991.

Busse, R., Fankhauser, P., Huck, G., Klas, W. (1994) IRO-DB An object-oriented approach towards federated and interoperable DBMS. *Proc. of the International Workshop on Advances in Databases and Information Systems (ADBIS'94)*, Moscow, Russia, Russian Academy of Sciences.

Busse R., Fankhauser P., Neuhold E.J. (1994a) Federated Schemata in ODMG, *Proc. 2nd Int. East/West Database Workshop*, Klagenfurt, Austria.

Castano, S., Fugini, M., Martella, G., Samarati, P. (1995) *Database Security*. Addison-Wesley.

Essmayr, W., Kastner, F., Pernul, G., Preishuber, S., and Tjoa, A M. (1995) Access Controls for Federated Database Environments. *Proc. Joint IFIP TC 6 and TC 11 Working Conf. on Communications and Multimedia Security*, Graz, Austria.

Essmayr, W., Kastner, F., Pernul, G., Preishuber, S., and Tjoa, A M. (1996) Authorization and Access Control in IRO-DB. *Proc. of the 12th Int. Conf. on Data Engineering*, New-Orleans, Louisiana, USA.

Essmayr, W., Kastner, F., Pernul, G., Tjoa, A M. (1996a) The Security Architecture of IRO-DB. *Proc. 12th IFIP Int. Conf. on Information Security*, Island of Samos, Greece.

Fernandez, E.B., Larrondo-Petrie, M.M., Gudes, E. (1993) A Method-Based Authorization Model for Object-Oriented Databases. *Workshop in Computing: B Thuraisingham, R. Sandhu, T.C. Ting (Eds.) Security for Object-Oriented Systems*, Washington DC.

Fernandez, E.B., Gudes, E. and Song, H. (1994) A Model for Evaluation and Administration of Security in Object-Oriented Databases. *IEEE Trans. on Knowl. & Data Eng.*, Vol. 6/2.

Gal-Oz, N., Gudes, E., Fernandez, E. B. (1993) A Model of Methods Access Authorization in Object-Oriented Databases. *Proc. 19th VLDB Conference*, Dublin, Irland.

Gardarin G., Gannouni S., Finance B., Fankhauser P., Klas W., Pastre D., Legoff R., Ramfos A. (1994). IRO-DB: A Distributed System Federating Object and Relational Databases. In Bukhres O. and Elmargarmid A.K., *Object-Oriented Multidatabase Systems*, Prentice Hall.

Jonscher, D., Moffett, J. D., Dittrich, K. R. (1993). Complex subjects or: The Striving for Complexity is Ruling our World. Proc. 7th IFIP 11.3 Working Conference on Database Security. Huntsville, AL, 1993.

Pernul, G.(1994) Database Security. In: Advances in Computers, Vol.38, pp. 1-72. (M. C. Yovits, ed.). Academic Press.

Sheth, A.P. and Larson, J.A. (1990) Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, Vol.22/3.