An Original View Mechanism for the CORBA Middleware

Olivier Caron, Bernard Carré, Laurent Debrauwer
Laboratoire d'informatique Fondamentale de Lille, UPRESA CNRS 8022
Université des Sciences et Technologies de Lille, 59655 Villeneuve d'Ascq cedex
Email : {Olivier.Caron, Bernard.Carre, Laurent.Debrauwer}@lifl.fr

Abstract

For the last years, middleware industry, and particularly the CORBA project, have strongly integrated several aspects issued from databases: persistence, transaction management, security, etc. However, both most of object-oriented database management systems (OODBMS) and CORBA middleware have not integrated an essential component: the notion of views (also described in the ANSI-SPARC architecture) which allows various users to see and manage data from different viewpoints. Views provide a **semantic structuring** of a schema of classes. A lot of application domains are interested in this feature; data mining is a well-known example. The paper focuses on CORBA and proposes a complete software architecture in order to provide an efficient view mechanism for this middleware. This new software architecture provides solutions about evolution problems of available data. We experienced an original use of the IDL language and the IDL compiler which consists of defining for each client (view) a subset of the global server IDL description.

Keywords: CORBA, enterprise architecture, views, databases, security, software architecture, evolution.

1 Introduction

3-tier client/server architectures are a good solution to access data from different sources: for example, databases, text files, HTML documents, ...

In the CORBA world, the second tier (the server objects) is in charge of extracting data from different sources (the third tier). In the third tier, we find existing databases and legacy applications. The first tier (the client objects) and the second tier are linked by using the IDL language. All data can be described with this purely declarative language. The problem is that shared data have an unique description for all clients. Relational databases have introduced view systems [1] to resolve this well-known problem. Views allow various users to see data from different viewpoints in separate functional contexts. Views may be also a solution for evolution problems : evolution of the global schema (server side), evolution of client characteristics (client side). In the world of *object-oriented databases*, views still are an open problem and very few OODBMS provide such a system; section 2 details this aspect.

Objects with multiple descriptions are also an identified problem in the CORBA world [2]. The future CORBA 3.0 standard (scheduled for 2000) plans to provide the possibility to define objects with multiple interfaces via the future component model of this standard[1]. However, CORBA 3.0 will not allow to define a complete system of contexts. We

[1] `http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html`

need to define both the *intension* and the *extension* of a schema of classes for a given group of users. These two notions define a context. A view-schema, which is relevant to a given group of users, corresponds to a subset of the shared global schema available in the network. The *intension* of the schema is in charge of selecting classes and, inside of these, the different properties relevant to a context. The *extension* is in charge of selecting the different instances of a class for a given context.

Our claim is to show that the *current* CORBA standard already owns several aspects which can provide view mechanisms. In our approach, we experience an unconventional use of the IDL language and the IDL compiler which provides the system of views (the first tier and the second tier of the architecture) as shown in Figure 1. This new CORBA service is available for all data sources. After a description of an usual CORBA architecture in section 3, we present the different modifications and a framework to implement view mechanisms.

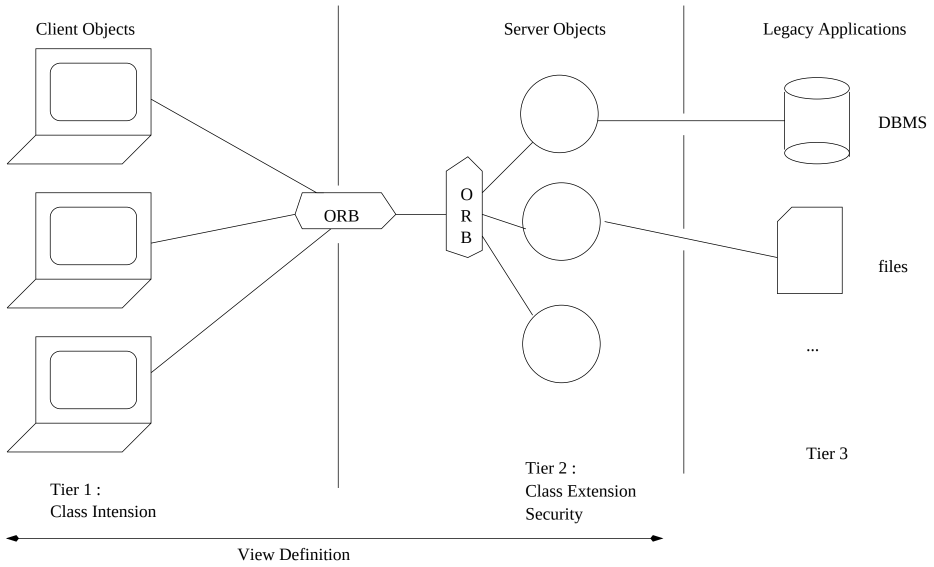

Figure 1: 3-tier Architecture

2 Views in Databases

Thanks to a three-level architecture [3] as shown in Figure 2, logical independence and physical independence of data are both main concepts in databases:

- **the internal level** is in charge of storing data.
- **the conceptual level** is in charge of structuring data.
- **the external level** is in charge of extracting and structuring data for a given group of users.

In this paper, we focus on the external level which achieves the logical independence of data.

Relational databases often use views. Views allow to see values in the database that are **computed** instead of being **stored**. View mechanisms are very powerful, it is possible to completely define the external level of the three-level architecture by views. However, a view cannot be considered as a simple relational table because view update is often impossible [4].

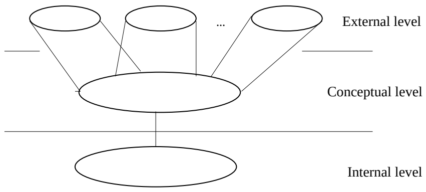

Figure 2: The three-level architecture

For this paper, we are using this very simple example as shown in the UML schema of Figure 3: a `Person` entity, defined with the `name` and `age` attributes, is employed in a `Job` of the enterprise. The `Job` entity is specified with the `name` attribute and the `salary` attribute. Two groups of users are using this database :

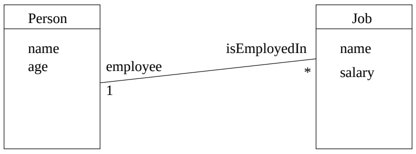

Figure 3: The global database

- users from the `accountancy` service which are mainly interested in the salary of each person of the enterprise.
- users from the `human resources` service which are interested in the employees (their age and their job).

Views of these groups are described in Figure 4.

In the world of **object-oriented databases**, a view-class is also specified by a query. However, the object model is richer than the relational model and causes many problems. Two of them are:

1. how a view-class is inserted in the class hierarchy?
2. does an instance of view-class own an identity ?

Because of these problems, most of commercial OODMBS do not provide view mechanisms [5]. However, you will find below some significant research results.

Abiteboul and Bonner in [6] use a virtual class mechanism for instantiating views in object-oriented databases. Here, classes for views are explicitly defined where the attributes of these classes are really methods that retrieve the information from where it is actually stored. Views do not preserve object identity [7].

In [8], views are specified by a set of query operators which preserve the object identity. Each view-class holds a type and an extension. [8] gives details on how to insert view-classes in the global schema.

[9] follows the work of [8] and introduces the notion of external schema or view. In [9], a view is a coherent subset of classes and view-classes from the global schema. A view is a schema which describes relevant information for a group of users.

3 Classical Use of CORBA

CORBA [10, 11] is an efficient middleware to access data stored by object-oriented databases. Databases need a lot of services already present in CORBA (transaction

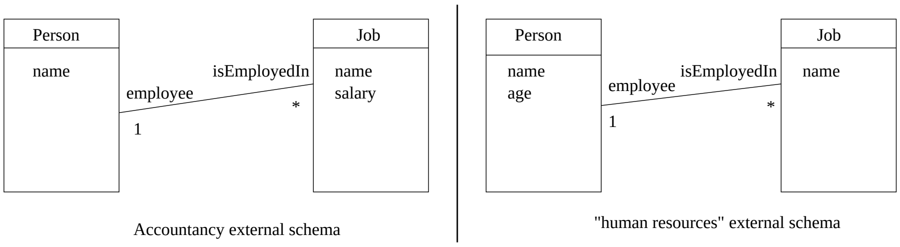

Figure 4: External schemas

management, security,...). In the network of a complex enterprise, CORBA and object-oriented databases are working together in the following manner as shown in Figure 5. A

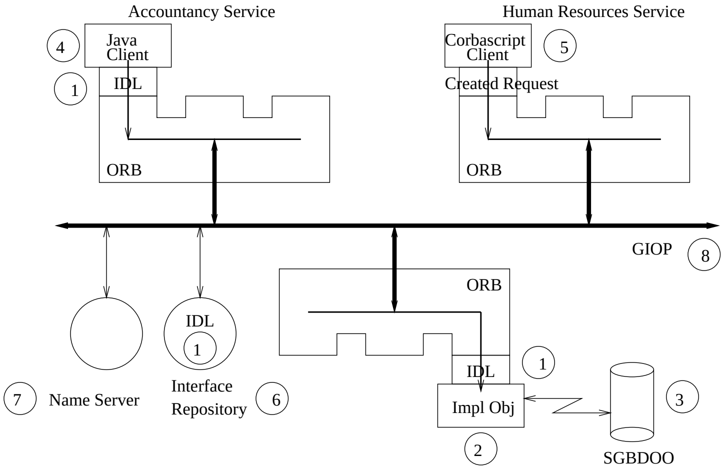

Figure 5: Example of an usual network of an enterprise

description of the Information System or, more precisely, a description of available data is specified by the IDL language in an IDL file. Client IDL stubs, like the ones on the server, are created using an IDL compiler thanks to the IDL file (see ① in Figure 5). From a client's perspective, the stub acts like a local call; it is a local proxy for a remote server object. On the server side, an object must be written according to the IDL file. This object ② can have access to data from a database ③. The IDL description for our example is the following:

```
module ISexample {
  interface Job ;
  interface Person {
    attribute string name ;
    attribute long age ;
    Job getIsEmployedIn() ;
    void setIsEmployedIn(in Job value) ;
  } ;
  typedef sequence<Person> ListOfEmployees ;
  interface Job {
    attribute string name ;
    attribute double salary ;
    ListOfEmployees getEmployee() ;
```

```
    void addEmployee (in Person value) ;
    void removeEmployee(in Person value) ;
  } ;
} ;
```

On the client side, the IDL description can be managed according to the two following manners:

- A client application written in a high-level language like C, C++, Java or Smalltalk uses the client stubs generated by the IDL compiler. These precompiled stubs define how clients invoke objects on the server. In fact, the programmer simply invokes a language method to obtain a remote service. This type of client is called a *static client*.
- You can also use specific interpreters or scripting language to dynamically generate object implementations. These tools ⑤ use the Interface Repository (IR) ⑥ which is a run-time distributed database of machine readable versions of IDL descriptions. A well-known example of such a scripting language is CorbaScript [12] based on this self-describing system.

A name server ⑦ is available on the network in order to obtain remote object references. In the case of a complex network of an enterprise, you can use inter-ORB bridges (using the GIOP protocol ⑧) to create very flexible topologies via federations of ORBs [2].

3.1 Limitations of the IDL languages for implementing views

The IDL language has not been designed for implementing views. This section underlines the limitations of the IDL language. The IDL language can not be considered as a programming language and does not own the same characteristics. So, if we consider the Java interfaces mechanism [13]. When applying this mechanism to the IDL language, the following IDL example produces an error at compilation time:

```
module ISexample {
  // HumanRessources view
  interface ViewHumanRessourcesPerson {
    attribute string name ;
    attribute long age ;
  } ;
  interface ViewHumanRessourcesJob {
    attribute string name ;
  } ;
  // Accountancy view
  interface ViewAccountancyPerson {
    attribute string name ;
  }
  interface ViewAccountancyJob {
    attribute string name ;
    attribute double salary ;
  }
  // the global schema
  interface Person : ViewHumanRessourcesPerson, ViewAccountancyPerson
{ } ;
  interface Job : ViewHumanRessourcesJob, ViewAccountancyJob { } ;
} ;
```

The IDL language is just a description language, nor overriding neither overloading are supported. In the above example, the definition of the IDL interface `Person` is false because the `name` attribute is defined twice. For implementing views, the above example present an another problem: a client can see all the different views !

4 Views for CORBA Middleware

We propose here to use CORBA in a specific way in order to provide view mechanisms. On the server side, the global schema is specified by an IDL description. View-schemas, subsets of the global schema, are also specified with the IDL language according to inclusion rules (see section 4.1). This system allows view-class to preserve both the global class hierarchy and object identity.

Section 4.1 details the specification of views and the different generation rules of stubs. Section 4.2 describes the security issue. Then, section 4.3 describes the definition of an extension of a class in a given context.

4.1 View Definition: the intension of a class

The CORBA architecture of the network of the enterprise must be lightly modified (see Figure 6). These modifications are the following:

- Each ORB bus (corresponding to a group of users) have its own interface repository (see ① in Figure 6).
- For each group of users, an IDL description is defined corresponding to the external schema of the three-level architecture. This IDL file is used by the IDL compiler to only generate client stubs (for static clients) ②. This IDL file can either be stored in the local Interface Repository (for dynamic clients) ③.
- The IDL file corresponding to the global schema is used to generate server stubs ④ .

So, the client IDL descriptions are separated from the global server's one and must be subsets of it. We detail in the following sections how views are used and the different

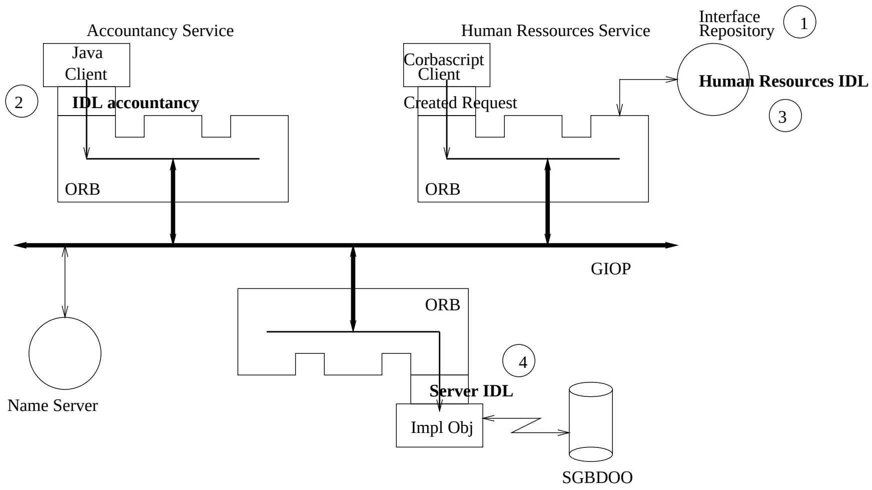

Figure 6: Example of a Corba network with views

rules to write client IDL descriptions.

4.1.1 Dynamic Client

Thanks to the name server, the dynamic client obtains a object reference and uses it to retrieve the object's interface and dynamically constructs the request. The dynamic client obtains the necessary information from the metabase of the Interface Repository. Then, the created request is invoked on the object server. IDL client description (an external schema) and IDL server description (the global schema) are working together if all information stored in the client metabase are available on the server side. The metabase contains module definitions (`ModuleDef`). An instance of `ModuleDef` can contain interface definitions (`InterfaceDef`). An instance of `InterfaceDef` can contain attribute definitions (`AttributeDef`), method definitions (`OperationDef`),... We distinguish two kinds of objects: `Container` objects and `Contained Objects`. For example, instances of the `InterfaceDef` class are `Containers` and instances of `AttributeDef` are `Contained`.

In order to get view mechanisms, each `Container` element of the client metabase must be a subset of the corresponding `Container` element on the server metabase as illustrated in Figure 7.

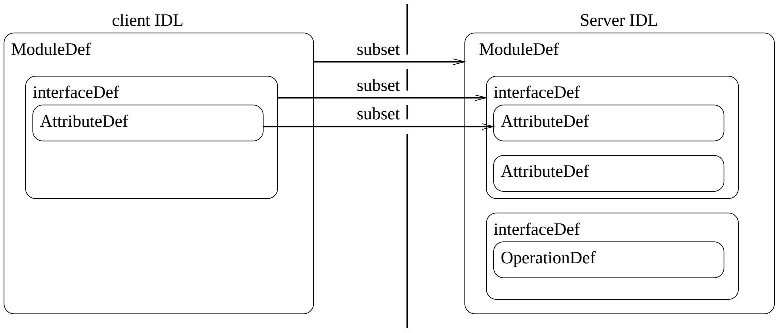

Figure 7: Rules between client IDL and server IDL

If we respect the above rule, dynamically created requests are accepted by the server object. For our example, here is the IDL description for the human resources service:

```
module ISexample {
  interface Job ;
  interface Person {
    readonly attribute string name ;
    Job getIsEmployedIn() ;
  } ;
  typedef sequence<Person> ListOfEmployees ;
  interface Job {
    readonly attribute string name ;
    ListOfEmployees getEmployee() ;
  } ;
} ;
```

In this external schema, `age` and `salary` information are not "visible". For this view, data can not be modified (`readonly` clause).

4.1.2 Static Client

For static client, the above rule for generating IDL client description is also required. The only difference is that we use this IDL file to generate client stubs as shown in Figure 8.

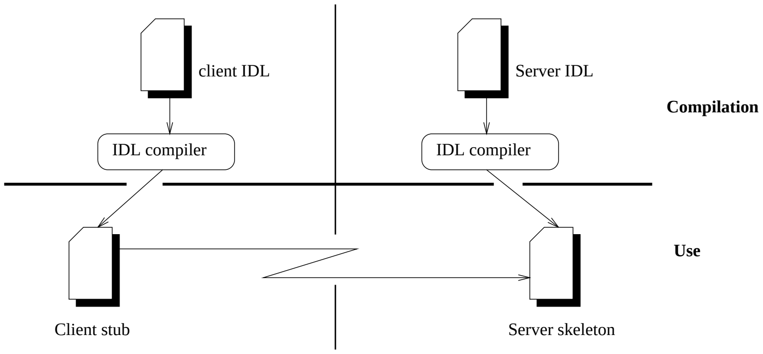

Figure 8: Compilation and Use Steps

On the server side, server stubs (CORBA calls them skeletons) are generated by the IDL compiler from the IDL file corresponding to the global schema.

4.2 View Definition: Security

Multiple client IDL descriptions provide **view intensions**. However, security is not guaranteed by this mechanism. For example, a client can use a client IDL file corresponding to an another context and then obtain non available information for this client. In our architecture, it is the second tier (the server objects) which is in charge of checking the correct use of views. According to the characteristics of the selected programming language, security implementation may differ. However, we propose an implementation schema which is independent from any language (see section 4.2.1) and a Java implementation (see section 4.2.2).

4.2.1 Secured Application Scenario

In the following scenario, three objects are working together: the client object, the server object and an instance of the `ContextManager` class which can be considered as a runtime distributed database of contexts. A context is simply identified by its name. Here is a step-by-step description of the object interactions shown in Figure 9:

1. *Client identification and authentication.* Client object invoke `setContext` method on instance of the `ContextManager` class. Arguments of this method are the name of the context and a password. The `ContextManager` object is in charge of checking the password.
2. *Client object invoke the remote operation* on the server object. For each method invocation, the server object:
   (a) asks the `ContextManager` object for the context of the current client.
   (b) checks access rights before executing the asked method.

4.2.2 Security Implementation in Java

A lot of security features are being implemented for Java 1.2 [13]. We use some of them in order to implement view security and we provide a Java API consisting of two classes : the `ContextSecurityManager` class which is a subclass of `java.lang.SecurityManager` and the `ContextPermission` class which is a subclass of java.security.Permission class. Server objects are running under control of this specific security manager. Moreover, a

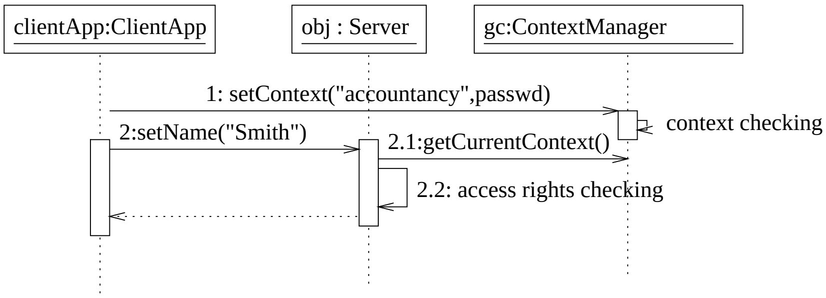

Figure 9: view security: UML sequence diagram

security policy defined in a text file describes the different permissions. You specify this policy file when invoking execution of the server:

```
java -Djava.security.policy=ISexample.policy ServerMain
```

The policy file (called 'ISexample.policy' in the above example), recognized by Java 1.2, indicates what permissions of the `ContextPermission` class are allowed for code from specified code sources. More precisely, you specify the method name of a class (`ClassName.methodName`) and the different allowed contexts for this method. Here is a section of this file:

```
grant  codebase "file:/corba/views/example/-" {
  permission ContextPermission "Person.setName" ,"accountancy";
  permission ContextPermission "Person.getName" ,"accountancy",
                                                 "humanResources" ;
  ...
```

The different methods must explicitly check the permissions. Here is an example for the `setName` method of the `Person` class.

```
public void setName(String value) {
  //  begin generated code
  ContextSecurityManager sm ; ContextPermission cp ;
  sm =(ContextSecurityManager) System.getSecurityManager() ;
  cp=new ContextPermission("Person.setName",sm.getCurrentContext())
  sm.checkPermission(cp) ;
  //  end of generated code
  this.name=new String(value) ;
  }
```

This section of code is always the same for each method (except underlined code). This code can be generated by a specific tool.

The security manager throws a security exception in the case of a non available method invocation for the current context. For each remote invocation, the `checkAccept` standard method of `java.lang.SecurityManager` class is called and checks the context of the current client as described in section 4.2.1.

The great advantage of this system is that we can modify access rights **without modifying the code**: we just need to update the policy file.

4.3 View Definition: the Extension of a Class

A view provides both the intension and the extension of a class for a given group of users. Like security implementation, the second tier of the architecture (the server objects) is in charge of providing the extension of a class for a given context (of group of users).

For each context, we would like to determine the set of instances of the classes which participate in this context. We propose to use predicates as in the view mechanisms of the databases [9]. When this predicate is satisfied by an instance, then this instance participates in the given context. The participation predicate of an instance is supplied by its class. The extension of a class in a context is then determined by the set of the instances of its extension, which satisfy the participation predicate. To implement this predicate, the IDL description on the server side is slightly modified:

```
module ISexample {
  interface Object {
    boolean participate(in string contextName) ;
  } ;
  interface Employee : Object {
  ...
```

In the server IDL description, all defined interfaces have a same root interface: the `Object` interface. The implementation of this interface is an abstract class. So, sub-classes must all implement the `participate` method. This method is in charge of return the predicate participation according to the given context.

In our example, all instances of the `Job` class are participating in the context of the `accountancy` service. Instances of the `Job` class are participating in the context of the `human resources` service only if the salary is less than 1000 Euros (The human resources service does not consider the boss's salary !). The following section of Java code shows the implementation of `participate` method of the `Job` class:

```
  public boolean participate(String contextName) {
    if (contextName.equals("humanResources") {
      return salary < 1000.0 ;
    else if (contextName.equals("accountancy") {
      return true ;
    else return false ;
  }
```

In the same way than security implementation, each method body of the different classes insert the following line of Java code:

```
  if (!this.participate(sm.getCurrentContext()) throw new ContextException() ;
```

Here, the method can throw an exception. That goes in the same sense that a OMG request for proposal (RFP) which allow to specify exceptions when defining attributes like the following IDL line:

```
  attribute string name raises (ContextException) ;
```

5 Conclusion

The proposed software architecture to implement a complete view mechanism presents multiple advantages. First, we don't need any modifications at the level of data sources. Second, view mechanisms for the CORBA middleware are easy to implement. The additional code that must be inserted at the beginning of each method, can be entirely generated by a specific tool. Third, evolution is a main aspect of the proposed architecture: we just need to give clients a new IDL file (corresponding to the new external schema) and to update the new policy file (see section 4.2). We don't need to modify neither second-tier (the server objects) nor third-tier of the 3-tier client–server architecture in order to update external schemas. On server side, new methods may have been added and you don't necessarily have the ability to update all your clients (it is often the case for large scale applications with an important number of clients); with our proposed architecture, old clients and new clients can communicate with the new server.

Evidently, object-oriented databases are very interested in this system. However, thanks to the CORBA client/server architecture, this new CORBA service can be used by many other data sources. View mechanisms bring them the possibility to structure their data according to different clients.

References

[1] R.G.G. CATTELL. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, 1994.

[2] O.M.G. Home Page, `http://www.omg.org`, 2000.

[3] D. TSICHRITZIS and A. KLUG. The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems. Technical report, Information Systems, 1978.

[4] J. ULLMAN. *Principles of Database and Knowledge-Based Systems*. Computer Science Press, 1989.

[5] L. DEBRAUWER. *Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, décembre 1998.

[6] S. ABITEBOUL and A. BONNER. Object and Views. In *Proceedings of SIGMOD*, pages 238–247, May 1991.

[7] O.A. BUKHRES and A.K. ELMAGARMID. *Object-Oriented Multidatabase Systems*. Prentice Hall, 1996.

[8] M.H. SCHOLL, C. LAASCH, and M. TRESCH. Updatable Views in Object-Oriented Databases. In *Proceedings of the Deductive and Object-Oriented Databases, Second International Conference*, pages 189–205, December 1991.

[9] E.A. RUNDENSTEINER. A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB'92)*, pages 187–198, August 1992.

[10] J.M. GEIB, C. GRANSART, and P. MERLE. *CORBA: des Concepts à la Pratique*. Inter-Editions, 1997.

[11] R. ORFALI and D. HARKEY. *Client/Server Programming with Java and Corba*. Wiley, 1998.

[12] P. MERLE, C. GRANSART, J.F. ROOS, and J.M. GEIB. CorbaScript: A Dedicated CORBA Scripting Language. In *CHEP'98 Computing in High Energy Physics*, Chicago, Illinois, USA, 1998.

[13] Java Home Page, `http://www.javasoft.com`, 2000.