Constructing IDL Views on Relational Databases

Kim Jungfer[1], Ulf Leser[2] and Patricia Rodriguez-Tomé[1]

[1]EMBL Outstation, The European Bioinformatics Institute,
Wellcome Trust Genome Campus,
Cambridge CB10 1SD, United Kingdom
{jungfer, tome}@ebi.ac.uk
[2]Technische Universität Berlin, Einsteinufer 17,
D-10587 Berlin, Germany
leser@cs.tu-berlin.de

Abstract. Data collections are distributed at many different sites and stored in numerous different database management systems. The industry standard CORBA can help to alleviate the technical problems of distribution and diverging data formats. In a CORBA environment, data structures can be represented using the Interface Definition Language IDL. Manually coding a server, which implements the IDL through calls to the underlying database, is tedious. On the other hand, it is in general impossible to automatically generate the CORBA server because the IDL is not only determined by the schema of the database but also by other factors such as performance requirements. We therefore have developed a method for the semi-automatic generation of CORBA wrappers for relational databases. A declarative language is presented, which is used to describe the mapping between relations and IDL constructs. Using a set of such mapping rules, a CORBA server is generated together with the IDL. Additionally, the server is equipped with a query language based on the IDL. We have implemented a prototype of the system.

1 Introduction

Integration of data from multiple, distributed and autonomous data sources is a challenging problem in many domains. Semantic and technical heterogeneity is common and data structures are often complex and evolve over time. The field of Molecular Biology can serve as an example. Historic development and organizational obstacles have prevented the definition and proliferation of standards, leaving end users confronted with an overwhelming diversity in data formats, query languages and access methods. Several proprietary systems like SRS [4] and Entrez [15] have been developed for the integration and distribution of molecular and genomic data. A Biologist nowadays has to find an access method to the desired data source, typically on the WWW. Then he has to understand and use the interface, e.g. by typing in keywords in a form, and finally he has to analyze the results, for instance by parsing

HTML pages. Evolving data structures on the other hand, leave the developers of user interfaces with the tedious task of keeping their Web sites up-to-date.

Clearly, this approach will not be able to cope with the increasing complexity, diversity and amount of data. Several research groups, e.g. [14], have therefore started to apply CORBA to alleviate some of the problems described. CORBA [10], [16], as a middleware platform, has many advantageous features: it offers language, location and platform transparency, which means that clients such as analysis programs and visualization tools can access remote information as if they were local objects. In a CORBA environment, data can be represented using the Interface Definition Language (IDL). This allows clients to work with domain-specific data structures and permits a flexible combination of data sources with different visualization and analysis tools [7].

In this scenario, a CORBA server has to implement a mapping between database structures and their IDL representation. Usually, this mapping is implemented manually: the developer first specifies appropriate IDL definitions, lets the IDL compiler generate the skeletons, and then adds the necessary implementation, mainly code to access the database through a database gateway such as JDBC. However, implementing a new CORBA server for each application and maintaining it in the presence of evolving IDL definitions and database schemas is tiresome. Furthermore, it is completely unclear how ad hoc queries can be supported in such a setting.

Another possibility is the automatic generation of IDL and CORBA server based on the schema of the underlying database. But such an IDL is normally not what we need for our concrete application. One reason is that in order to allow for the interoperation of independently developed clients and servers it is advantageous to agree on a common IDL [1]. Another reason is - as detailed in chapter 3 - that there are many different ways to represent data in IDL. The different representations have different advantages and disadvantages, and can significantly influence the performance of a distributed system. The IDL is therefore highly application specific and cannot be derived from a database schema only.

In this paper, we describe a system that offers a partial solution to these problems for the case of relational databases. Many major and minor data collections in molecular biology, like EMBL [17] or IXDB [9], utilize a relational database management system. Furthermore, relational databases provide a powerful view mechanism, which can be exploited to facilitate the mapping task. This also allows for a higher degree of independence between the CORBA server and the schema of the database.

The central idea of our approach is to automatically generate the server source code based on a set of rules, which describe the mapping from a relational schema to a target IDL. The generated server implements a query language, which is purely based on the IDL definitions. Therefore, the user does not need to know the schema of the relational database. The server can either be used directly, or the code can be exploited as a skeleton for further customizations.

The following example, depicted in Figure 1, will be used throughout the text. The schema represents a very simple model of a genome map. A genome map has a name, the name of the represented chromosome and a set of markers (e.g. genes). Each marker has a name, a position on the map, and the information indicating whether the marker belongs to the framework of the map or not.

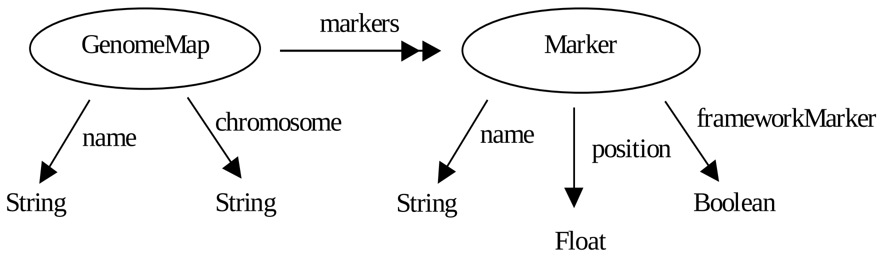

Fig. 1. Example schema for genome maps and markers.

The following IDL is *one* possibility to represent this schema. The genome map is represented by an interface, while markers are represented by structs. This has the advantage, that the application can download all markers belonging to a map with only one remote method invocation.

```
module Example {

  struct Marker {
    string name;
    float position;
    boolean frameworkMarker;
  };

  typedef sequence <Marker> Markers;

  interface GenomeMap {

    readonly attribute string name;
    readonly attribute string chromosome;
    readonly attribute Markers markers;

  };

}; // End of module Example
```

The rest of the paper is organized as follows. The architecture of the system is described in section 2, the mapping language is presented in section 3, query possibilities of the system are examined in the section 4, related work and further issues are discussed in section 5 and 6.

2 Architecture

Figure 2 depicts an overview of the architecture of the system. The process of generating a CORBA wrapper is as follows: The first step is to decide what IDL optimally serves the application. Then one or more IDL views can be defined using our specialized language. The view definitions describe the mapping between IDL constructs on one side and tables and columns on the other side. Using these rules, a generator creates both a CORBA server and a file with the implemented IDL. The generated IDL can be used to implement a CORBA client for this server.

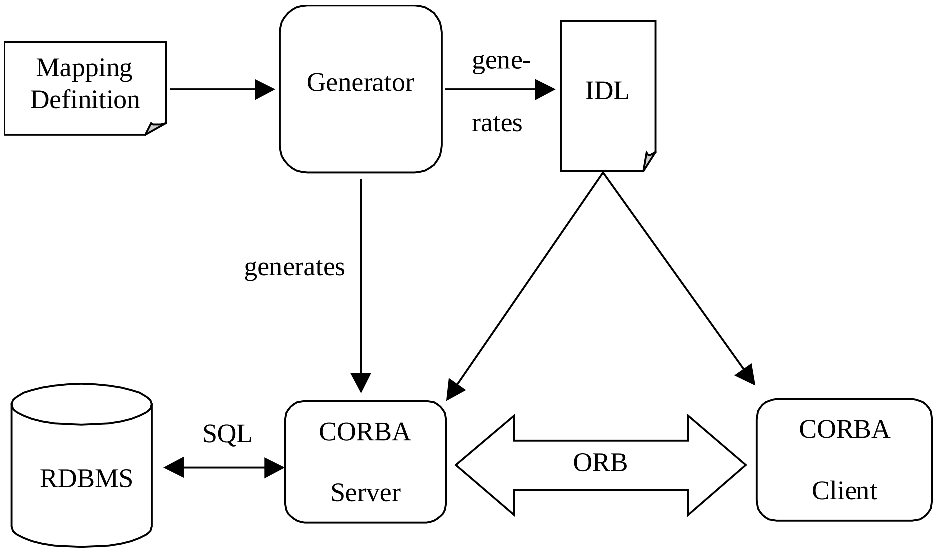

Fig. 2. Architecture

The client can query the server using a query language, which is based on the generated IDL. The queries are translated to SQL queries using the definitions of the mapping language. The results are then translated back in the required IDL representation and returned to the CORBA client.

3. Mapping of a Relational Schema to IDL

We can distinguish between approaches that use CORBA only for the infrastructure of the system, treating data objects and queries basically as strings or bit-streams, and others that also model the data itself in IDL. While the former is for instance followed by [3], we adopt the latter possibility. The main advantage of this solution is the reduced impedance mismatch. The ultimate goal is that clients can be developed based purely on the automatically generated stubs, without having to decode bit-streams or strings into domain objects.

There are a number of different possibilities to represent data in IDL. For example it is possible to model database entries either as structs or as interfaces. Both approaches have advantages and disadvantages. Interfaces can resemble a conceptual data model very naturally, but are often too slow since every method invocation goes over the network. Structs in contrast are copied by value and then accessed locally. Structs are therefore more suitable for bulk data transfers. On the other hand, structs neither support inheritance nor associations. If such concepts are used, they need to be circumscribed. This can for instance occur if an extended entity-relationship model is used as the starting point for the IDL development. The optimal representation of a relational schema in IDL is strictly application dependent. To offer maximum flexibility, our system supports both structs and interfaces.

We will first present the concepts of the mapping language and its grammar, then we give an example, and finally we discuss the generated IDL of the CORBA server.

3.1. The Mapping Language

In this section we describe a high-level language, which can define mappings between relations and columns on one side and different IDL types on the other. The IDL is completely specified by the mapping definitions. To facilitate the task, we restrict the language to the most often used constructs of IDL: modules, the basic types long, float, string, boolean, the constructed type struct, the template type sequence, and interfaces. Interfaces are restricted to read-only attributes. To keep the mapping language simple and without loss of generality we assume that relational views can be used to get closer to the needed IDL. This means that mapping possibilities, which can be expressed by a relational view are not considered here. More specifically:

- Every simple type in the IDL (long, float, string, boolean,) is represented directly by one table attribute. No null-values are permitted.
- All single-valued members (attributes) of a struct (interface) can be found in the same table.
- Multi-valued members (attributes) of a struct (interface) are stored in a different table, which is connected to the base table by foreign keys.

Views

The top-level construct of the mapping language is the view definition. A view has a name and is associated with a table and the mapping for an IDL type, which is either a struct or an interface (xxx`_mp` in the grammar means: mapping definition for xxx). Different views can use the same IDL types based on different tables.

```
view ::= (view <view_name>
          table <table_name>
```

```
              <interface_mp> | <struct_mp>)
```

Interfaces

The mapping definition for interfaces specifies first the IDL name of the interface together with all interface names from which it inherits. All interface names are scoped to specify the appropriate IDL module. The extended interfaces are merely necessary to define the IDL type - they do not affect the mapping. Especially they do not imply any set inclusion properties between different views represented by these interfaces. Then the primary key for the table is given. The key is necessary to allow the CORBA object adapter to keep track of the connection between object references and database entries. For each attribute, the attribute name and the mapping for the attribute type is given. All attributes have to be specified here, including those inherited from other interfaces.

```
interface_mp ::= (interface <scoped_interface_name>
                  extends (<scoped_interface_name>*)
                  keys (<column_name>*)
                  (<attribute_name> <data_type_mp>)+)
```

Structs

For every struct the scoped struct name is specified as well as the mapping for each member. Since structs are passed by-value, there is no need for keys.

```
struct_mp ::= (struct <scoped_struct_name>)
               (<member_name> <data_type_mp>)+)
```

Data Types

The type of an attribute or struct member is either a basic type or an object reference or a struct or a sequence.

```
data_type_mp ::=   <basic_type_mp>
                 | <reference_mp>
                 | <struct_mp>
                 | <sequence_mp>
```

Basic Types

Every type mapping is defined in the context of a table. All basic types are directly represented by one of the table's columns. The only exception is the type boolean, which does not exist in some relational databases. In this case it is necessary to give additionally the value which represents true.

```
basic_type_mp ::=  (boolean <column_name> <true_value>)
                 | (long    <column_name>)
                 | (float   <column_name>)
                 | (string  <column_name>)
```

References

This type represents object references. The corresponding interface has to be defined in a separate view. In this case it is necessary to specify the connection between the current table and the table of the referenced view. This is done by giving the primary / foreign keys of the two involved tables.

```
reference_mp ::= (reference <view_name>
                  keys (<column_name>+)
                       (<column_name>+))
```

Sequences

The data, which belongs to a sequence, is multi-valued and therefore stored in a different table. Again we have to specify the connecting columns of the two tables. In the case where the subtype is an object reference, it is sufficient to give the view name instead of a complete reference mapping as described above. The reason is that the connecting columns are already specified in the sequence mapping. Since sequences are ordered it is additionally necessary to specify an order-by-clause.

```
sequence_mp ::= (sequence <scoped_sequence_name>
                 table <table_name>
                 keys (<column_name>+) (<column_name>+)
                 <sequence_type_mp>
                 order_by <order_by_clause>)

sequence_type_mp ::=   <struct_mp>
                     | <sequence_mp>
                     | <basic_type_mp>
                     | <view_name>
```

3.2 Example

The example is based on the schema and IDL given in the introduction. We define a mapping using the tables of the relational schema depicted in Figure 3. The attribute *map_id* in the table *map_markers* is the foreign key of the table *maps*.

Table: maps

id	chromosome

Table: map_markers

id	position	fw	map_id

Fig. 3. Relational schema.

The mapping definition directly reflects the nested interface-sequence-struct structure of the IDL given in section 1. Note that in the opposite case, where a struct contains a reference to an interface, the interface would be defined in a separate view. Also note that the nesting of IDL modules does not have to be the same as the nesting of the mapping language. For example, it is possible that a struct contains another struct from a completely different module.

```
( view GenomeMaps
  table maps
  ( interface Example::GenomeMap
    extends ()
    keys (id)
    ( name (string id) )
    ( chromosome (string chromosome) )
    ( markers
      ( sequence Example::Markers
        table map_markers
        keys (id) (map_id)
        ( struct Example::Marker
          ( name (string id) )
          ( position (float position) )
          ( frameworkMarker (boolean fw T) ) )
        order_by position
      )
    )
  )
)
```

3.3 The Generated IDL

Given the mapping definitions in section 3.2, a set of IDL definitions can be automatically generated. The first part of this IDL represents the data as specified in the mapping language. In our example it is identical with the IDL given in the introduction. The second part specifies the API for the database itself and defines methods for querying and data retrieval. This works as follows: The *Evaluator* interface has a *get* method defined for each view. The client can specify here a where-clause similar to SQL queries (see next section). The evaluator returns a reference to an iterator. Again, there is a separate iterator specified for each view. The iterator has a *next* method, which returns object references or structs of the type defined in the view. Additionally there are methods *count* and *next_n* to allow the client to optimize the data retrieval. For our example the following IDL is generated:

```
module Views {

  exception NoMoreElements {};
  exception InvalidQuery {};
```

```
    interface Iterator {
      boolean more();
      void close();
    };

    typedef sequence<Example::GenomeMap> GenomeMapSeq;

    interface GenomeMaps: Iterator {
      Example::GenomeMap next() raises(NoMoreElements);
      GenomeMapSeq next_n(in long n);
    };

    interface Evaluator {
      long count(in string viewName, in string where)
           raises (InvalidQuery);
      GenomeMaps get_GenomeMaps(in string where)
                 raises (InvalidQuery);
    };

  }; // End of module Views
```

Note that the *count* method can be used for all views implemented by the server whereas each *get* method is defined for only one view.

4. Queries

Our approach maps a relational schema into IDL, thereby alleviating the infamous impedance mismatch between application code and relational database. Query results are always represented by a predefined type, either structs or object references. This is naturally achieved by class specific *get* methods and iterators as described above. Using this approach, we can avoid the usage of the generic IDL type any. Anys are less efficient for the data transfer and inconvenient to use in client programs. However, using fixed result types inevitably restricts the query power, as arbitrary joins and projections have to be disallowed. In practice this restriction is of little significance and shared by many other applications such as digital libraries.

The *get* methods of the *Evaluator* interface takes as input parameter a string, which is comparable to a SQL where-clause. The predicates of the query are formulated using attribute names and member names of the IDL interfaces and structs. Client code depends therefore only on the IDL and is immune against most schema changes in the database.

4.1 The Query Language

We introduce the language informally using some examples. Conditions on basic types can be specified using the predicates ‘<’, ‘>’, ‘<=’, and ‘>=’. Predicates can be combined using ‘and’, ‘or’ and ‘not’. If a member or attribute contains a sequence then the quantifiers ‘exists’ and ‘all’ can be used. Queries can contain nested subqueries to specify embedded structs or referenced objects.

Examples:
We assume that a view for markers exists for the IDL in the main example. The following where-clauses could be specified in the *get_Markers* method. Note that in this case structs and not object references would be returned. If, as in Q4, several member conditions are specified, then all conditions have to be true.

Q1: “All markers“
→ `“”`

Q2: “The markers with the name ‘RH2345’ and ‘RH5432’ “
→ `(name (or ‘RH2345’ ‘RH5432’))`

Q3: “All markers except the marker with the name ‘RH2345’ “
→ `(name (not ‘RH2345’))`

Q4: “All non-framework markers with a position greater than 100“
→ `(frameworkMarker false) (position (> 100))`

Q5: “All markers with a position between 20 and 30“
→ `(position (and (>= 20) (<= 30)))`

For the view *GenomeMaps*, as defined in the last section, the following queries are possible. The *GenomeMap* attribute *markers* contains a sequence of structs. At this place the quantifiers *exists* and *all* can be used. Inside the quantifier a specification for the struct has to be given, which is a list of member conditions as in Q1-Q5.

Q6: “All maps which contain the marker with the id ‘RH3456’ “
→ `(markers (exists (name ’RH3456’)))`

Q7: “All maps which contain only framework markers“
→ `(markers (all (frameworkMarker true)))`

4.2 Query Mapping

The translation of our query language to SQL is based on the mapping rules. As these rules always associate each struct or interface with one table, this translation is fairly straightforward. Nested queries are translated to nested SQL statements using the predicate ‘in’. We give the translation of queries Q2 and Q6 as examples. Again we assume the relational schema in 3.2.

```
Q2: select distinct id, position, fw from map_markers
    where id=‘RH2345’ or id=‘RH5432’

Q6: select distinct id from maps
    where id in (select map_id from markers
                 where id=’RH3456’))
```

Note, that in Q2 all information on the markers is retrieved whereas in Q6 only the key. The reason is that in Q2 a struct is returned, which has to contain all data, while in Q6 only an object reference is returned.

5. Related Work

We are not aware of any other project that follows our track of generating CORBA servers and IDL based on a set of declarative mapping rules. However, a number of research areas share problems. For instance, mapping relations to IDL interfaces is related to object-relational mapping (e.g. [12], [18], [20]). The mapping step, consequently called “semantic enrichment“ in [6], can in general not be automated because the relational schema simply does not carry the necessary information. Hence, the mapping rules must be specified by a human operator, as done in our approach.

The translation of object-oriented queries into a query against a semantically equivalent relational schema is covered in depth in [5] and [13]. The approach of [5] is similar to ours in that they also assume that each (object-oriented) class is represented by exactly one (relational) table. However, our query language is only a subset of theirs, as we do not treat path expressions. [13] considers extensional relationships in inheritance hierarchies by mapping the translation into DATALOG programs, which are used as a mediator between the query and the database. In contrast, for our mapping we do not require nor guarantee any relationships between extents of interfaces that are in a specialization relationship.

Another related research area is the integration of database systems in a CORBA framework. [8] discusses several design issues in this context, including the consequences of using structs or interfaces for object representation. They clearly point out that it is in general very difficult to achieve full relational query power through CORBA, mainly due to the static type system. The OMG itself has contributed to this area through the “Object Query Service Specification“ [11]. However, as detailed in [8] and [19], this specification has severe pitfalls, and, to our knowledge, has not been implemented by any of the commercial ORB vendors. For instance, it does not support any representation of domain objects on the CORBA level.

There are only few commercial tools available, which support the generation of CORBA access layers for relational databases. For example Persistence TM[1] defines an object-oriented schema on top of a relational schema. A programming library is generated which makes the data accessible through a set of C++ classes. Additionally the tool can generate a CORBA server, which maps the OO schema into IDL and uses the library to access the database. The main problem with this tool is the limited influence the developer has in the choice of the generated IDL. It is purely interface-based with no support for struct-based representations, which are essential to ensure sufficient performance. Hence, in real-life applications, it is necessary to change the generated CORBA server to a great degree by hand. But these changes are not visible for the query processor. A similar approach is taken be the OPM project [2]. We believe that our method is a more direct and flexible way of achieving a CORBA interface to an existing relational database.

6. Discussion

We have presented a method for the semi-automatic generation of CORBA wrappers for relational databases. Compared to the two other major approaches – hand-coding or completely automatic generation – our system offers many advantages. CORBA views can be defined easily, allowing many applications to share data, each with its own IDL. It is straightforward to generate redundant IDL definitions, for instance containing both a struct and an interface for the same data. This leaves it to the client application to choose the most convenient access method.

The server is equipped with a query language, which can express complex conditions. Usage of this query language does not require any knowledge of the schema of the underlying database, but is entirely based on the IDL itself. The client code is therefore completely independent of schema changes, provided that the mapping rules are adjusted. Although, it is clear that our query language can only express a limited set of queries, it proved to be sufficient in our application domain.

Using a set of mapping rules, the system generates JAVA source code for the server. We have chosen this compilation strategy for several reasons. Firstly, it offers a considerably better performance compared to an interpretation of the rules at run-time. Secondly, the code can be used as a template for further customizations. Finally, it allows the usage of skeleton code generated by the IDL compiler, which significantly simplifies the code generation task. The disadvantage is that every change in the mapping rules requires the regeneration of the server. However, the choice between interpretation and compilation is an implementation detail, which does not touch the principal of our approach.

Some problems remain when specifying a query based on IDL definitions. They stem from the fact that IDL was designed to specify an API and not to model data. An

[1] http://www.persistence.com

example is the usage of inheritance. If an interface A specializes an interface B then a query against B does not necessarily return a superset of the same query against A. Such a behavior can be enforced using the mapping language and appropriate relational views, but it is not visible from the IDL alone. Other problems can occur when the requirements for querying are not identical to the requirements for data retrieval. For instance we might not want to retrieve the information indicating whether a marker belongs to the framework of a map but still be able to use it in a query. These problems would vanish if we use the schema and query language of the underlying relational database and IDL merely represents query results. The disadvantage would be that then the user has to know the relational schema, the IDL and the mapping between the two.

Future investigations will go into extensions of the mapping language. For example we will include a possibility to express simple inheritance on structs through the use of unions. We also aim to improve the query language by adopting a more SQL like syntax.

Acknowledgements

We would like to thank Richard Göbel, Philip Lijnzaad, Jeremy Parsons, and Anastasia Spiridou for their ideas and comments. Ulf Leser was supported by the German Research Society, Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

References

1. Barillot E., Leser U., et al.: A proposal for a standard CORBA interface for genome maps. Bioinfomatics, **15**. Oxford University Press. (1999) 157-169
2. Chen I.M.A., Kosky A.S., et al.: Advanced Query Mechanisms for Biological Databases. Proc. of the 6th Int. Conf. on Intelligent Systems for Molecular Biology (ISMB), Montreal, Canada. AAAI Press, Menlo Park (1998)
3. Dogac A., Dengi C., et al.: A Multidatabase System Implementation on CORBA. 6th Int. Workshop on Research Issues in Data Engineering: Nontraditional Database Systems, New Orleans, Louisiana (1996)
4. Etzold T., Ulyanov A., Argos P.: SRS: Information Retrieval System for Molecular Biology Data Banks. Methods in Enzymology, Vol. 266. Academic Press (1996) 114-128
5. Fahl G., and Risch T.: Query Processing over Object Views of Relational Data. The VLDB Journal **6**(4). Springer-Verlag (1997) 261-281
6. Hohenstein U.: Using Semantic Enrichment to Achieve Interoperability of Relational and ODMG Databases. International Hong Kong Computer Society Database Workshop (1996) 210-232
7. Jungfer K., and Rodriguez-Tomé P.: Mapplet: A corba-based genome map viewer. Bioinformatics, **14**(8). Oxford University Press (1998) 734-738

8. Leser U., Tai S., Busse S.: Design Issues of Database Access in a CORBA Environment. Workshop on Integration of Heterogeneous Software Systems, Magdeburg, Germany (1998)
9. Leser U., Wagner R., et al.: IXDB, an X Chromosome Integrated Database. Nucleic Acids Research., **26**(1). Oxford University Press (1998) 108-111
10. Object Management Group: The Common Object Request Broker: Architecture and Specification. John Wiley & Sons, New York (1995)
11. Object Management Group: CORBAServices: Common Object Service Specification: Query Service Specification. http://www.omg.org (1997)
12. Papazoglou M., Tari Z., and Russell N.: Object-Oriented Technology for Interschema and Language Mappings. Object-Oriented Multidatabase Systems: A Solution for Advanced Applications, Bukhres O.A., Elmagarmid A.K. (eds.) Prentice Hall, New Jersey (1996) 203-250
13. Qian X., and Raschid L.: Query Interoperation among object-oriented and relational databases. 11th Int. Conference on Data Engineering, Tapei, Taiwan. IEEE Computer Soc. Press (1995)
14. Rodriguez-Tomé P., Helgesen C., Lijnzaad P., and Jungfer K.: A CORBA server for the Radiation Hybrid Database. Proc. of the 5th Int. Conf. on Intelligent Systems in Molecular Biology (ISMB). AAAI Press (1997) 250-253
15. Schuler G.D., Epstein J.A., et al.: Entrez: Molecular Biology Databases and Retrieval System. Methods in Enzymology, Vol. 266. Academic Press (1996) 141-162
16. Siegel J.: CORBA Fundamentals and Programming, John Wiley & Sons (1996)
17. Stoesser G., Moseley M.A., et al.: The EMBL Nucleotide Sequence Database. Nucleic Acids Research, **26**(1). Oxford University Press (1998) 8-15
18. Tari Z., Stokes J.: Designing the Reengineering Service for the DOK Federated Database System. Proc. of the IEEE Int. Conf. On Data Engineering (ICDE'97), Birmingham (1997) 465-475
19. Wells D. L., and Thompson C. W.: Evaluation of the Object Query Service Submissions to the Object Management Group. IEEE Quarterly Bulletin on Data Engineering, **17**(4). (1994) 36-45
20. Wiederhold G.: Views, Objects and Databases. IEEE Computer, Vol. **19**(12). (1986) 37-44