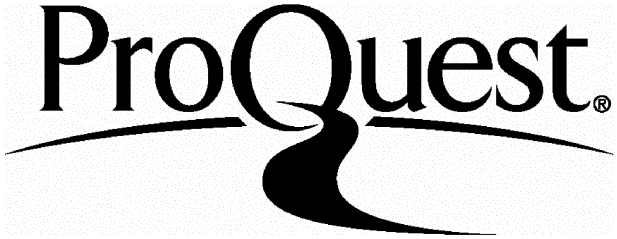
SEMIAUTOMATIC GENERATION OF CORBA INTERFACES FOR DATABASES IN MOLECULAR BIOLOGY

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

by
Kim Michael Jungfer

July 2000

Department of Biochemistry
University College London



SEMIAUTOMATIC GENERATION OF CORBA INTERFACES FOR DATABASES IN MOLECULAR BIOLOGY

ABSTRACT

The amount and complexity of genome related data is growing quickly. This highly interrelated data is distributed at many different sites, stored in numerous different formats, and maintained by independent data providers. CORBA, the industry standard for distributed computing, offers the opportunity to make implementation differences and distribution transparent and thereby helps to combine disparate data sources and application programs. In this thesis, the different aspects of CORBA access to molecular biology data are examined in detail. The work is motivated by a concrete application for distributed genome maps. Then, the different design issues relevant to the implementation of CORBA access layers are surveyed and evaluated. The most important of these issues is the question of how to represent data in a CORBA environment using the interface definition language IDL. Different representations have different advantages and disadvantages and the best representation is highly application specific. It is therefore in general impossible to generate a CORBA wrapper automatically for a given database. On the other hand, coding a server for each application manually is tedious and error prone. Therefore, a method is presented for the semiautomatic generation of CORBA wrappers for relational databases. A declarative language is described, which is used to specify the mapping between relations and IDL constructs. Using a set of such mapping rules, a CORBA server can be generated automatically. Additionally, the declarative mapping language allows for the support of ad-hoc queries, which are based on the IDL definitions.

ACKNOWLEDGEMENTS

I wish to thank everybody who has contributed to the success of this thesis. Special thanks go to my supervisors Patricia Rodriguez-Tomé and Janet Thornton who have made this work possible. I am grateful to my previous supervisor Tomas Flores who has encouraged me to work on CORBA interfaces for molecular biology databases. Finally, I would like to thank Jeroen Coppieters, Richard Göbel, Carsten Helgesen, Ulf Leser, Philip Lijnzaad, Jeremy Parsons, Martin Senger, and Anastasia Spiridou for our discussions from which I gained many ideas.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 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. Research groups have collected genome-related data for the last 20 years, during which the amount of data has grown exponentially. Medical, pharmaceutical, and agricultural researchers increasingly seek to utilise this information for the development of new treatments, drugs, and crops. There are now more than 300 publicly available collections of highly interrelated data, containing many different types of information. They include data as diverse as nucleotide sequences, protein sequences, protein structures, genome maps, taxonomic classifications, and metabolic pathways. The combination of these data collections promises new insights through new high level integrated views.

The more the size and complexity of molecular biology data grow the more important automatic tools for management, querying and analysis become. Many of the current limitations in using this wealth of information are not due to missing technology but to a lack of standardisation. Biologists utilise every possible hardware platform, operating system, database management system and programming language. Therefore a large proportion of the available programming resources have to be dedicated to unnecessary data transformations, which could be better used concentrating on the functionality of programs. In this thesis an approach is

investigated to alleviate these problems using the Common Object Request Broker Architecture (CORBA). Using CORBA, it is possible to hide the implementation details of data sources and application programs and make them readily accessible via a network such as the internet. This approach promises to reduce drastically the effort required for the deployment and maintenance of distributed systems and facilitates the reuse of existing components.

The rest of the introduction is organised as follows: First the current situation in molecular biology is analysed together with the most important systems and technologies used. Then the concepts of interfaces and components are introduced and the question is discussed of how they can facilitate the building of future distributed applications. Finally, the CORBA standard itself is introduced and compared with related technologies.

1.1 Data Collections in Molecular Biology

Data management in molecular biology is characterised by a situation where historic development and organisational obstacles have prevented the definition and proliferation of standards. End users are confronted with an overwhelming diversity in data formats, query languages and access methods. In this section the most important of the currently used systems and technologies are surveyed. These are flat files for the storage of data, the World Wide Web for the access of biocomputing resources, and several proprietary systems for the management of molecular and genomic information.

1.1.1 Flat Files

Molecular biology data has been traditionally stored in simple text files, often referred to as flat files. Flat files are popular because no database management system is needed and they can be distributed easily using ftp or CD-ROM. Furthermore, flat-files can be read and understood directly by humans. This fact has lead biologists to identify an entry of a data collection with its

flat file representation. The advent of the World Wide Web strengthened this view. Flat files can be transformed easily to hypertext by turning references into hypertext links. The Sequence Retrieval System SRS [Etzold 96] is a well-known example for this approach. Flat files became the centre of the data flow in molecular biology. Every data collection has to provide a flat file version in order to distribute the data and most analysis programs use flat files as their data source.

This central role of flat-files has several disadvantages. The most obvious problem is that flat-files are difficult to use. Writing a parser is a non-trivial task, which is further complicated by imprecisely specified and frequently changing formats. Even though some standard flat file formats exist, such as the EMBL/DDBJ/GenBank format, biologists find it in general difficult to agree on a single model of their data. Another major drawback is that flat files can lead to an immense waste of computing resources. Different programs often expect different flat-file formats so that the same site needs to keep multiple copies of the same data. For example sequence comparison programs, such as BLAST [Altschul 90] or FASTA [Pearson 90], have their own format, which is different from the format of the primary sequence databases. Finally, the wish of human readability results in the attempt to keep all information associated with an entry together. Since this attempt conflicts with the goal of normalisation, flat-files tend to be redundant. For instance the address of an author might appear in all entries of that author.

1.1.2 The World Wide Web

The World Wide Web [Berners-Lee 94] has revolutionised the way molecular biology data is accessed today. All large providers of molecular biology resources, such as the National Center for Biotechnology[1] (NCBI) and the European Bioinformatics Institute[2] (EBI), now offer a Web-based access to their data collections and tools. Hypertext links are ideally suited for the rich interconnectedness of biological data and allow for an easy browsing of distributed information.

Form-based keyword searches are capable of expressing complex queries and new technologies such as JAVA permit the development of sophisticated Web-based user-interfaces and visualisation tools. The main advantages of the Web are its ease of use and its small hard- and software requirements. A desktop computer with Internet connection and a Web browser are sufficient - no other programs need to be installed locally. In many cases this is a very good solution as long as the service provider has anticipated the specific requirements for searching, analysis and visualisation of the data. The situation is however problematic if a researcher would like to analyse the data using his own methods and tools. In this case Web interfaces are very cumbersome to use. In order to utilise data from the Web in a program, the relevant HTML pages have to be saved on the local computer individually and then parsed by a hand-written program.

1.1.3 Information Systems

The need for software, which allows for a unified access to the diverse data sources in molecular biology, has been recognised early. Many different systems for the management, integration and distribution of biological data have been developed [Bishop 98, Letovsky 99]. Some of the most important systems are examined here.

1.1.3.1 Entrez

Entrez[3] is an integrated Web front end and search engine for the databases at the NCBI. This includes the MEDLINE bibliographic citation service, the GenBank nucleotide sequence database, a protein database - which includes SwissProt and translated GenBank entries, the Molecular Modelling Database (MMDB) of 3-D structures, the Genome Database with genetic and physical maps from several different species, and a taxonomy database. Entrez maintains links between these databases and calculates neighbours of sequences using BLAST. Apart of the Web interface, NCBI also supports the so-called *Programmers Toolkit*, a C language library,

which allows accessing Entrez via a network. NCBI uses the Abstract Syntax Notation (ASN.1) to integrate its various databases. In separate text files, ASN.1 is used to express both the model of the data as well as the data itself. The toolkit includes a program, which automatically generates C language structure definitions from an ASN.1 model and object loaders, which read and write from ASN.1 files to C structures. Even though ASN.1 files are used for the internal data management, they are less suitable to be read and understood by humans than the traditional flat file formats.

1.1.3.2 SRS

The Sequence Retrieval System[4] (SRS) [Etzold 96] is another tool for the integration of molecular biology data collections. Most users will access its Web interface but it also has a C language API and a UNIX command line interface. SRS installations are replicated at many sites, thereby providing for a certain degree of fault tolerance and allowing users to select the version with the best network connection. Unlike Entrez, SRS installations contain not only well-known databases like EMBL [Stoesser 99] and SwissProt [Bairoch 99] but also a large number of smaller, highly specialised data collections. SRS can directly utilise the original ASCII text file distribution formats of the data collection. Its main strengths are its powerful parser and the maintenance of indices of links between the different data collections. SRS implements a proprietary query language, view support, and a CORBA interface [Coupaye 99].

1.1.3.3 ACEDB

ACEDB [Durbin 94] was originally developed to manage the genetic data of the nematode C.elegans. It consists of a non-standard object-oriented database management system and a large number of excellent graphical visualisation tools. Both mouse navigation and queries are supported. A proprietary text file format is used as an external data representation for reading from or writing to a file. ACEDB has proved to be very popular with many other organism

communities and became the core element of the Integrated Genome Database (IGD) project [Bryant 97].

1.1.3.4 OPM

The OPM data management tools [Markowitz 99] are based on the Object Protocol Model (OPM)[5] [Chen 95], an object model with special support for the modelling of scientific experiments (protocols). The system provides wrapper facilities for the commercial relational database management systems Sybase[6] and Oracle[7]. It supports the automatic mapping of an object model to a relational schema as well as the mapping of queries [Chen 98]. Additional retrofitting tools provide views for already existing relational databases and for structured flat files using SRS. OPM has been employed for the development and maintenance of several biological databases. Examples include the version 6 of the Genome Data Base (GDB)[8], a repository for all published mapping information generated by the Human Genome Project, and the Primary Database of German Human Resource Center (RZPD)[9].

1.1.3.5 BioKleisli

BioKleisli [Davidson et al 1999] is a system for the integration of heterogeneous, distributed data sources using the *Collection Programming Language* (CPL). Its main strength is its rich type system, its query capabilities, and the large number of different data sources supported – among others it provides drivers for ASN.1, ACEDB, SRS indexed files, and relational database systems. BioKleisli supports virtual integration using views as well as physical integration by instantiating data warehouses.

1.1.4 Componentry

The information systems discussed provide integrated user interfaces to diverse types of biological information. Each of the systems has its own focus and a different conceptual origin.

The main strength of ACEDB is, the graphical display of the data, while other systems, such as BioKleisli, concentrate on the possibility to query heterogeneous data sources. Even though all of the systems try to anticipate as many user demands as possible they are not necessarily ideally suited for a specific application. Unfortunately it is difficult to combine different parts of these systems with other programs and tools. For example it would be hard to combine the query engine of SRS with the graphical user interface of ACEDB or to use a different genome viewer together with the ACEDB database. This is of course possible but it requires a lot of knowledge and programming effort. This effort seems to be so high that many developers find it easier to implement a new system from scratch rather than to try to combine existing programs. In this sense a program like ACEDB is sometimes called a closed, monolithic application. Monolithic applications limit the possibilities for reuse and lead projects to reimplement functionality that already exists in other systems.

In [Goodman 95b], this problem is addressed and the observation is made that: "almost all genome information systems are constructed from scratch with little reuse of software developed elsewhere". According to [Goodman 95b], the reason for this situation is that the architect of a genome information system has only two possibilities: The first is to build a complete new system, thereby guaranteeing that it has precisely the needed functionality. The second is to use an already existing system and live with most of its limitations. The paper suggests the concept of componentry as a way to allow for a solution between these two extremes. A component is a piece of software, which can be used in different contexts through a well-defined interface. The idea is to replace monolithic applications by collections of smaller components, which can be mixed and matched in order to obtain a tailor made solution for the specific problem in hand. The potential advantage would be more reuse of already existing code and therefore less wasted programming effort. Because the unit of contribution is no more a whole system but only a

component, it would become easier for an individual researcher to contribute to future information systems.

Several research groups in molecular biology have been motivated by this argument and have tried to create toolkits of compatible components for graphical user interfaces. One example is the *bioTk* project [Searls 95]. BioTk consists of a set of graphical widgets for the display of biological objects, such as chromosome maps. The bioTk widgets have been implemented in the programming language Tcl/Tk, using a consistent style and standardised data structures. Another example is the *BioWidgets* project [Fischer 99], which uses a very similar approach but is based on Java beans instead of Tcl/Tk. Both projects concentrate on only one programming language to facilitate the base-level interoperation of the different components. Also, both stress the importance of keeping the model of the underlying data structures as "schema neutral" as possible – the graphic widgets should deal with concepts of the display and selection of data rather than with biological concepts. This approach makes the components more general and therefore easier to replace or reuse.

The other main trend with the aim of facilitating componentry, has been the growing interest in the CORBA standard. Unlike bioTk and bioWidgets, CORBA is not restricted to a specific programming language. The aim of CORBA is to allow components to interoperate, independent of programming language, platform, and location.

1.2 The Common Object Request Broker Architecture

In 1989, the Object Management Group[10] (OMG) was formed. Its members include a large number of major software vendors, hardware vendors, and large end-users. The stated goal of the OMG is to standardise and promote object technology. Its specifications are sanctioned by the International Standards Organisation (ISO) by reference. The core specification adopted by the OMG is the Common Object Request Broker Architecture – CORBA [OMG 99] (see [Orfali

1996], [Siegel 1996] for good introductions). CORBA combines the concept of interfaces with the distributed object-oriented programming paradigm. Among other things the standard specifies:

- The Interface Definition Language (IDL), which provides a language-independent way of describing the public interface of distributed objects.

- The Object Request Broker (ORB), a piece of middle-ware, which transparently transmits request from clients to object implementations.

The public interface describes all methods a client can invoke on a distributed object together with the necessary input and output parameters. Once the interface of a CORBA object has been implemented by a server, a client can use it like a local programming language object. It is transparent to the client where the object implementation resides or in what programming language it is implemented.

1.2.1 The Interface Definition Language

The Interface Definition Language describes the attributes and operations of CORBA objects, whereby attributes function as a shorthand for get and set methods. The input and output parameters can have basic types, such as string or float, or constructed types, such as structs and unions, or template types, such as arrays and sequences. Another important type is the object reference, which is used to reference CORBA objects. Interfaces and data types can be grouped in a hierarchical name space using modules.

A compiler takes the IDL definitions as an input and generates a programming library for a specific target language, e.g. C++ or Java. For a client, the compiler generates so-called *stubs*.

Stubs are the client side representation of distributed objects in the programming language of the client. For a server, so-called *skeletons* are generated by the IDL compiler. Skeletons contain only empty method declarations in accordance with the signature of the object to be implemented. The developer of the server has to provide the implementation of the methods. Stubs and skeletons already include the necessary code for network communication and marshalling of parameters.

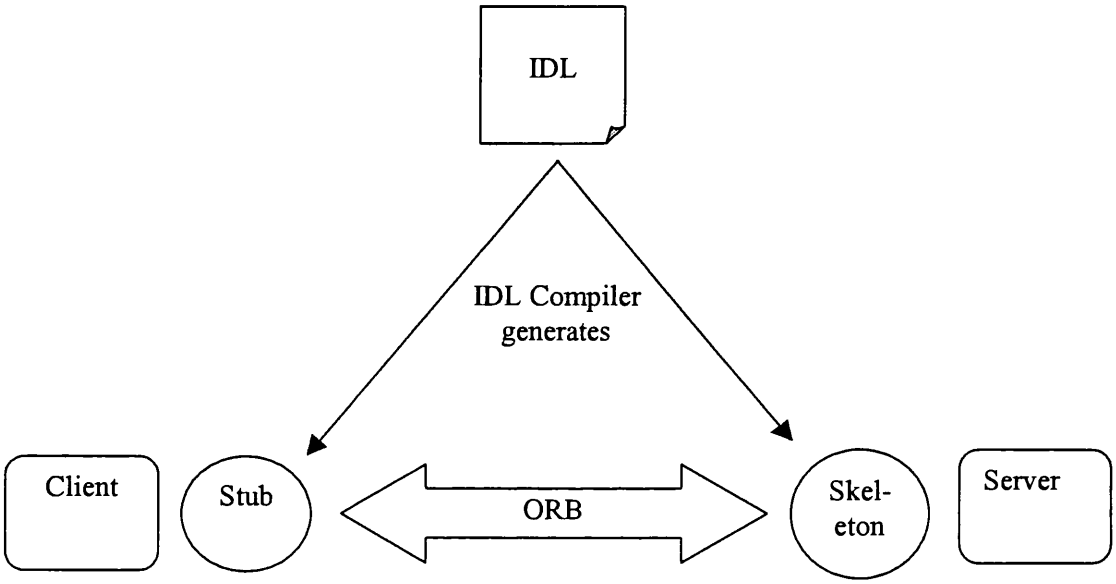


Figure 1: Stubs and Skeletons are automatically generated

The Interface Definition Language plays a central role in CORBA because it is the basis for its language independence. For each programming language, a standard mapping has been defined, which translates IDL definitions into constructs of that particular language. Without the usage of such an intermediate language it would be necessary to define a mapping for every pair of programming languages, which is clearly impractical. The CORBA standard includes language mappings for most of the important programming languages. Examples are C, C++, Java, and Smalltalk.

1.2.2 CORBA Objects and Object References

In order to invoke methods on a CORBA object the client needs to obtain an object reference for that object. The object reference serves as a proxy object on which the client can use the methods defined in IDL (Figure 2).

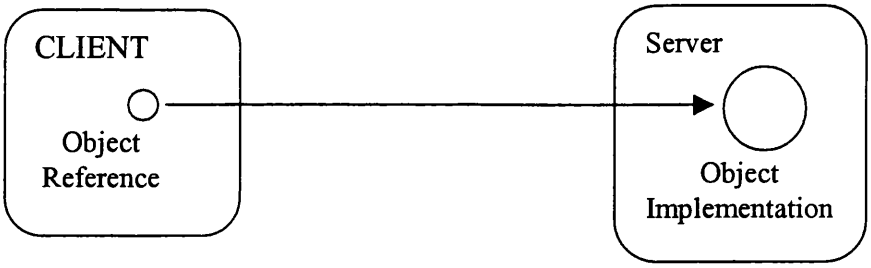


Figure 2: Object References are Proxies for Object Implementations

It is interesting to compare the CORBA approach with the traditional Remote Procedure Call (RPC) [Bloomer 92]. Using RPC a client invokes a function on a server, while in CORBA a client invokes a method on an object. Such a CORBA object is a purely logical entity whose implementation is unknown to the client. In many cases the server implements several CORBA objects as shown in Figure 3 (from [Orfali 96]).

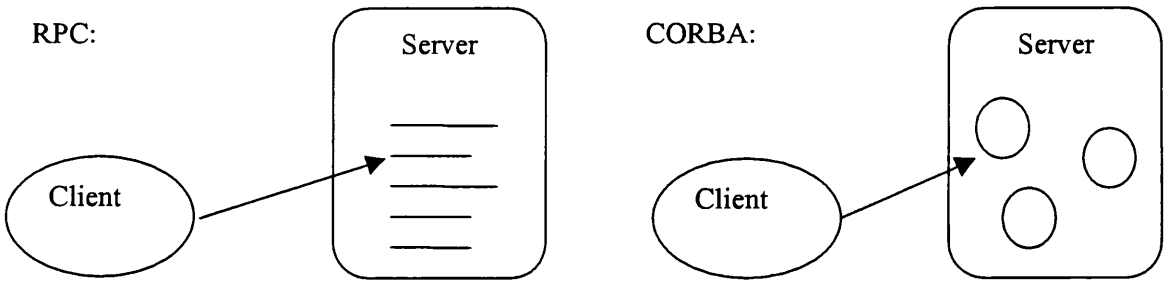


Figure 3: RPC vs CORBA

It is common to implement a CORBA server using an object-oriented language such as C++ or Java. In this case, the programming language object, which implements the CORBA object, is sometimes called the *servant.* Again, one servant can implement one or several logically distinct CORBA objects.

CORBA objects are bound to a specific implementation and only passed by reference using the object reference data type. In contrast, IDL data types such as *structs*, *unions* or *sequences*, are always passed by value and therefore copied between clients and servers.

1.2.3 The Object Management Architecture

CORBA and IDL minimise dependencies between different components by hiding implementation details like operating system, network, location, and programming language. This decoupling of clients and servers alleviates many of the common interoperability problems today and facilitates the development and maintenance of application programs. Even though this is obviously an improvement of the state of affairs it pushes the problem to another level. How can software components developed by independent groups interoperate? They can do so only if they share a set of common IDL definitions. The purpose of the Object Management Architecture (OMA) is the standardisation of IDL on several different levels (Figure 4).

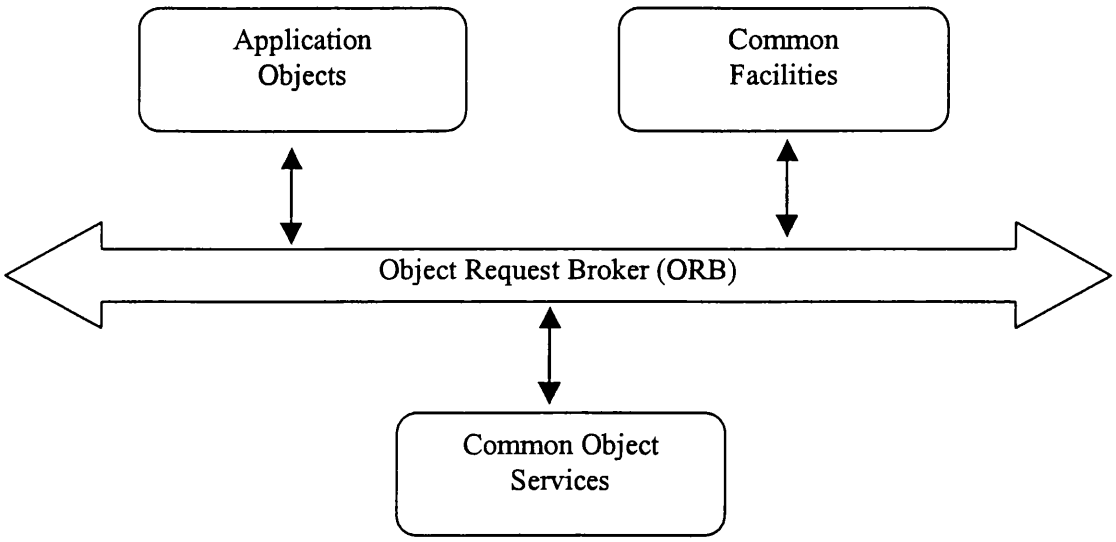


Figure 4: The Object Management Architecture (OMA)

The lowest level is formed by the *Common Object Services*. CORBA services standardise interfaces for basic functions, anticipated to be needed by many different applications across application domains. Examples are interfaces for life-cycle operations, the naming of CORBA objects and the querying of databases. The next group are the *Common Facilities*, which include higher level domain specific services, such as telecommunications or life sciences. Finally there are the application objects, which can make use of the CORBA services as well as the CORBA facilities.

1.2.4 CORBA in Molecular Biology

The advantages of the CORBA standard have been recognised by many research groups in molecular biology and there are an increasing number of CORBA applications available. Examples include CORBA servers and clients for the Radiation Hybrid Database [Rodriguez 97], HuGeMap [Barillot 99b], and SRS [Coupaye 99]. Several other applications in bioinformatics can be found at the CORBA page of EBI[11].

The OMG's Domain Task Forces (DTF) provide a formal framework for the adoption of standard interfaces for a specific application domain. In September 1998, a "Life Sciences Research" (LSR) Domain Task Force was formed whose goal is to provide a forum for everybody who wants to get involved in the creation of standards for the life sciences field. Under the umbrella of the LSR DTF, several working groups have been established. Among them are groups for gene expression, macromolecular structure, maps, sequence analysis, and visualisation and user interfaces.

1.2.5 Related Technologies

In this section CORBA is compared to related technologies. These are alternatives for distributed programming and object-oriented databases.

1.2.5.1 Distributed Programming

There are several other approaches, which can be regarded as a way of providing distributed programming capabilities. The most important of these are briefly surveyed here: sockets, CGI, RMI, and DCOM. A more thorough comparison together with a speed evaluation for JAVA can be found in [Orfali 97b].

1.2.5.1.1 Sockets

Berkeley sockets are the de-facto standard for network applications using TCP/IP. Most higher level middle-ware, including object request brokers, is built on top of sockets. Socket programming is very low level and provides no parameter marshalling. Therefore sockets require more programming effort but can achieve a better performance. According to [Orfali 97b], socket communication can be up to 50% faster in JAVA than an equivalent CORBA application.

1.2.5.1.2 HTTP/CGI

The Common Gateway Interface (CGI)[12] is part of the Hypertext Transfer Protocol (HTTP) of the World Wide Web. A CGI application is a program, which acts on data from a Web-based input form and is invoked by a Web server. The result of a CGI invocation is a HTML page, which is displayed in the client's Web browser. This means that virtual HTML pages can be created on the fly instead of being stored on the server side before hand. Another advantage is that CGI servers can be written in any programming language. The CGI/HTTP protocol is stateless even though cookies can help to save state between different invocations. The main disadvantages are that CGI is very slow and that it is only a specialised solution for Web front-ends.

1.2.5.1.3 DCOM

Microsoft's Distributed Component Object Model (DCOM) [Session 98] is an important alternative to CORBA. Shipped with Microsoft operating systems it has a large installation base. Apart from a different terminology, CORBA and DCOM offer a very similar functionality [Orfali 97b]. The performance of DCOM is similar to the performance of CORBA and like CORBA it offers parameter marshalling and language independence via DCOM IDL. Most of the arguments and methods in this thesis could be applied to DCOM as well. However, DCOM is not considered here because of its proprietary nature and lack of cross-platform support.

1.2.5.1.4 Java/RMI

Finally, the Remote Method Invocation (RMI) of Java is considered here [Harold 97]. RMI is a system for distributed programming in the Java programming language. Unlike CORBA, RMI does not try to be programming language independent. The advantage for the programmer is that Java serves as implementation language and as interface specification language. Hence, there is no mapping necessary between these two. This makes programming easier and allows for the support of specialised features of Java, which cannot be supported by a language

independent system. An example for this is RMI's dynamic downloading of class code. RMI offers both pass-by-reference and pass-by-value for Java objects. [Orfali 97a] reports a 42% performance advantage of a CORBA/Java ORB in comparison with Java/RMI at that time. However, there seems to be no good reason why there should be a big difference in performance given a similar quality of implementation. The language dependence of RMI and the proprietary nature of Java, can be regarded as the main disadvantages compared with CORBA.

1.2.5.2 Object-Oriented Database Management Systems

Even though object-oriented database management systems (OODBMSs) [Atkinson 89, Cattell 97] are not directly concerned with distributed computing, they do share some features with the approach discussed here. The main objective of an OODBMS is to achieve a seamless integration of the database language with an object-oriented programming language. This motivation – reduced impedance mismatch - is the same when CORBA IDL is used to represent data. Since many OODBMSs concentrate on only one host language – usually C++ or Java – they can achieve a better integration than would be possible with the language neutral IDL. The downside of such a system is that compromises have to be made when a database has been designed in one programming language but is used from within another language. Also, OODBMSs have no equivalent to the concept of CORBA objects as a unit of distribution. References between objects in an OODBMS may be turned into pointers of the programming language when the object is stored in memory – a process called swizzling (e.g. in [Cattell 94]). This is not possible with CORBA object references because the implementation of a CORBA object is hidden and resides potentially on another machine. Based on the concrete experiences with an attempt to implement a genome mapping system using a OODBMS, [Goodman 95] argues that the C++ type system is a poor choice as a database language because it is more complicated than a typical data modelling language. In this article, Goodman is also sceptical of implementing too many programs as part of the database schema because they tend to be very

application specific. These two points have to be considered for similar reasons when CORBA IDL is used to represent data and will be discussed later.

.

[1] http://www.ncbi.nlm.nih.gov/
[2] http://www.ebi.ac.uk/
[3] http://www.ncbi.nlm.nih.gov/Entrez
[4] http://srs.ebi.ac.uk/
[5] http://gizmo.lbl.gov/opm.html
[6] http://www.sybase.com/
[7] http://www.oracle.com/
[8] http://www.gdb.org/
[9] http://www.rzpd.de/
[10] http://www.omg.org/
[11] http://corba.ebi.ac.uk/
[12] http://hoohoo.ncsa.uiuc.edu/docs/cgi/overview.html

Chapter 2

GENOMIC MAPS

2.1 Introduction

There are now a large number of different types of genetic and physical maps available [Schuler 1996], [Dib 1996], [Hudson 1995]. Maps are interesting biological objects because they can be used to link different genes and other markers using positional information. It is, for example, possible to identify homologue regions in genomes of different species using syntenic regions. Furthermore, they are well suited for different visualisation techniques such as the highlighting of parts of a map as a result of a query.

The available maps are distributed at many different sites, each offering different kinds of access methods and viewers. They are typically accessible through Web interfaces, directly printable graphic files (e.g. Postscript) or simple text files. As discussed in the introduction, these options are not well suited for access by a program and restrict the user to predefined views, anticipated by the data provider. CORBA offers here new opportunities to overcome such restrictions. Using CORBA, clients and servers interact through public interfaces, which hide implementation details. This separation allows a flexible recombination of data sources with different viewers and applications, including Web interfaces. Several research groups have presented CORBA servers for map databases and map viewers. These systems can be found for instance in [Rodriguez-Tomé 1997, Barillot 1998, Hu 1998].

The main aim of this chapter is to demonstrate the benefits of CORBA for distributed applications in molecular biology. It also allows to highlight the relevant issues concerning the design of such CORBA applications. For this purpose, an IDL for the distribution of genomic maps is presented together with a map viewer, which serves as a client for this IDL. The origin of the system presented here is *Mapplet*, a Java/CORBA map viewer for the Radiation Hybrid Database (RHdb), which has been implemented by the author together with a specialised CORBA server. The Radiation Hybrid Database (RHdb) contains experimental radiation hybrid data and maps derived from these experiments [Rodriguez-Tomé 1999]. This includes STS data, scores, experimental conditions and extensive cross-references. EBI and Infobiogen in Paris have later collaborated to create an *Common Map IDL* (EU Grant BIO4-CT96-0346). This is a set of interface definitions, which are general enough to accommodate both, the radiation hybrid maps of EBI, and the genetic maps of Infobiogen [Barillot 99a]. Mapplet has been adapted by the author to the Common IDL, for which it served as a proof of concept [Jungfer 98]. It is this version, which is presented here. In the meantime the effort of establishing a standard IDL for genome maps has been taken over by the Life Sciences Research Task Force of the OMG.

The main requirements for Mapplet and the Common IDL have been:

- The map viewer should be able to access and display radiation hybrid maps from EBI in England as well as genetic maps from Infobiogen in France.
- The application has to work with a reasonable speed even for large maps with several thousand markers.
- The basic functionality of Mapplet should depend only on the common IDL, while additional information for radiation hybrid maps has to be accessible as well.

2.2 Architecture

CORBA objects are used to represent the main components of the application. These components are:

- **Maps:** Having an object reference to a map object, a client can request general information about the map, such as name and type, as well as the markers on the map and their position.
- **The Map Trader:** A client can ask the trader which maps are available. Example criteria are species, chromosome or a specific marker on the map. If the trader knows about maps, which satisfy the query, it returns one or more object references for these maps. The map viewer can use the object reference to directly access the map and display it.
- **The Score Database:** If the map is a radiation hybrid map of RHdb then additional information about the markers is available. Examples are marker types and score vectors.

The Common IDL treats maps as CORBA objects. This has the advantage that map and trader can be separated. The trader can know about maps, which are not local to the trader. When a client asks for a map then the trader does not return the map itself, but a reference to the map. The client can then use the reference to access the map directly without having to know where it resides (Figure 5).

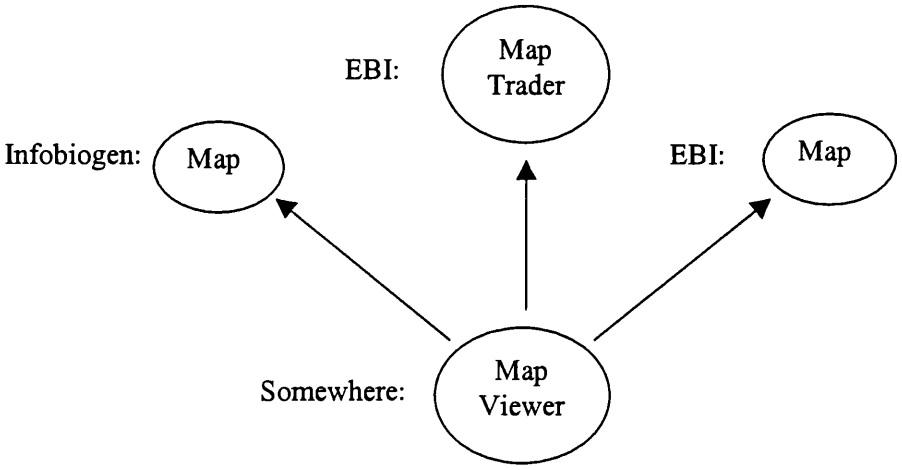


Figure 5: Architecture

2.3 The IDL

The IDL definitions for the application consist of two parts. The first part is the *Common Map IDL*, which contains methods and data available for all types of maps. The second additional part has only methods specific to radiation hybrid maps. The complete listing of the Common Map IDL can be found in appendix A. Here only those aspects of the common IDL are presented, which are necessary to understand the map viewer.

2.3.1 The Common IDL

The Common Map IDL consists of two parts: maps and traders.

2.3.1.1 Maps

The maps themselves are defined in the module maps. Their main attributes are "oid", "name", "type", "species", "chromosome" and "elements". Through the attribute "elements" it is possible to get a list of all markers, which belong to the map. Because the "MapElement" (marker) is represented as struct and not as interface it is passed-by-value. It is therefore possible to get all marker data belonging to a map with only one method call.

```
module Maps {
    ...
    struct MapElement {
        MarkerData markerData;
        float position;
        ...
    };
    typedef sequence <MapElement> MapElementList;
    ...
    interface Map {
        readonly attribute string oid;
        readonly attribute string name;
        readonly attribute string type;
        readonly attribute string species;
        readonly attribute string chromosome;
        ...
        readonly attribute MapElementList elements;
    };
};
```

2.3.1.2 Traders

The Common IDL defines the interfaces for traders and maps. The most basic method of a trader is to return an object reference given an object identifier. Since this method is not specific to map traders, it was defined in a separate trader module.

```
module Traders {
    interface Trader {
      ...
      Object getByOid(in string oid)
          raises (NotFound);
    };
};
```

A specialised version of the Trader is defined in the module "Maps". Here it is possible to specify general search criteria like species and chromosome. The result type "MapList" represents a sequence of maps, since more than one map can match a query.

```
module Maps {
    ...
    typedef sequence<Map> MapList;

    interface MapTrader : Traders::Trader {
        MapList getMapList( in string mapType,
                            in string species,
                            in string chromosome,
                            in Strings markers  )
            raises (Traders::Trader::NotFound);
    };
};
```

2.3.2 The IDL for Radiation Hybrid Data

The scores database is represented by the interface "DB" in the module "RHScores". It has two methods. The first allows to access additional information for an individual marker, and the second to get the marker types of all markers of a specified map. This second method is in theory redundant but was added for performance reasons. The map viewer allows the highlighting of all markers of a specific type. If the viewer would have to request the marker type individually for each marker then network load and response time would be unacceptable.

```
module RHScores {

    typedef sequence<string> Strings;

    struct Reference {
        string database;
```

```
        string accession;
    };
    typedef sequence <Reference> References;

    struct Score {
        string rhid;
        string panel;
        string author;
        string vector;
        string sts;
        Strings stsTypes;
    };

    typedef sequence <Strings> Types;

    exception NotFound {};

    interface DB {
        Score getScore(in string rhid) raises(NotFound);
        Types getTypes(in string mapid) raises(NotFound);
    };

}; // module RHScores
```

2.4 Mapplet: The Map Viewer

Mapplet allows a quick overview of the maps in RHdb (Figure 6.). It displays four maps side by side, which can be selected by chromosome. Each map can be scrolled and zoomed independently. Zooming can be done using the "In", "Out" and "Full" buttons.

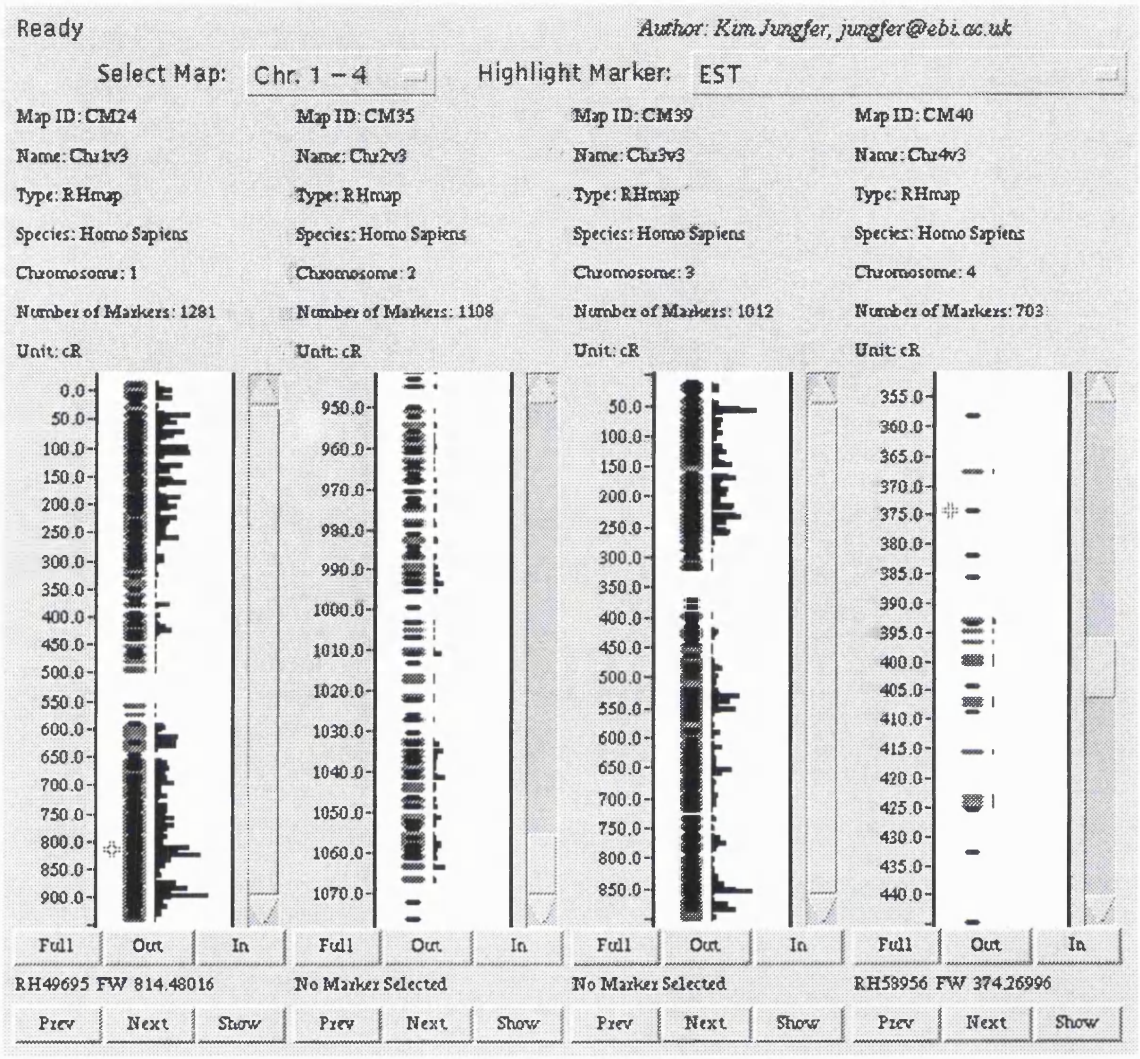


Figure 6: Snapshot

The maps are depicted in three parts:

- **A Scale**.
- **The Markers**:

  Depending on the zoom-level they generally overlap.

- **A Marker Density Histogram**:

  Depicts how many markers are mapped in a specific region.

A marker can be selected with the mouse by clicking on it. Pressing the "show" button of the corresponding map will display detailed information about the marker, like name, position, and cross-references. For Radiation hybrid maps at EBI the assay information together with the score vector is displayed. The "Next" and "Prev" button select the next or previous marker. Different marker types can be highlighted: for example, all markers, which belong to the framework or all genetic markers or all ESTs. The histogram will then show the density of the highlighted markers.

2.5 Implementation

The map viewer was implemented in Java using the CORBA 2.0 compliant ORB OmniBroker 2.0.3[1] . It can run as a standalone application as well as within a Web browser. Since no proprietary features are used, the same source code should work with any other recent, CORBA compliant Java ORB. Therefore, the applet can use the built-in ORB of Netscape's Communicator.

Unfortunately, when the client runs within a Web browser, security restrictions require the map trader, the map, and the HTTP-server from which the applet is loaded, to reside on the same machine. Other, more advanced ORBs, such as Visibroker[2], offer a solution to this problem using a daemon, which runs on the same machine as the HTTP-server and which can forward requests to object implementations on different machines.

It would have been possible for the application to utilise the normal existing CORBA servers for the Radiation Hybrid Database [Rodriguez-Tomé 1999] as a data source. However, in order to increase the performance, a special caching server was implemented, which uses these servers but avoids direct access to the underlying relational database at run-time. Since both servers use the same IDL, none of these changes affect the map viewer.

2.6 Discussion

The map viewer has been designed for use over Wide Area Networks (WANs) where round-trip response times can be slow and where granularity of data access becomes the single most important factor in browsing performance. A straightforward translation of a conceptual data model into an interface-based IDL will normally lead to inefficient solutions. An example is [Hu 1998], where the "MapElement" equivalent "Locus" is only represented as an interface. This means that in order to display a map it is necessary to access each displayed "Locus" individually by remote method calls. Such an approach is therefore limited to small maps or local network applications. In this context it is important to realise that IDL is not a data modelling language but a specification language for an API. Ultimately the concrete application determines the IDL – not an abstract data model. Using the common map IDL, the map viewer can download positions and names of all markers belonging to a map with only one method call. This is possible because the "MapElement" is modelled as a struct, which can be passed-by-value. Apart from this difference, the IDL in [Hu 1998] and the Common Map IDL show many similarities.

The Mapplet is based on the Common Map IDL. However, it was necessary to extend this IDL to implement features specific to RHdb. This is probably a typical situation. Agreement on a common interface, while desirable, is often difficult to achieve. If achieved it is often an unsatisfactory compromise. Since CORBA deals only with interfaces and not with implementations, this problem is less significant than, for example, with flat file formats. It is easy to built servers supporting different IDL definitions. Another possibility is the mechanism of inheritance for CORBA interfaces. E.g. it would be possible to define a specialised map interface for radiation hybrid maps, which inherits from the common map interface. The specialised map object itself could then give the additional information. However, this approach was not chosen here in order to keep score data and map objects separate. The main

functionality is preserved when the Mapplet is restricted to the common IDL. Specialised features are accessible using an additional set of IDL definitions.

2.7 Conclusions

Representing genomic maps and markers using CORBA IDL allows clients to access remote information like local programming language objects. This is convenient since no parsing or other internalisation process are necessary. CORBA object references provide location transparency so that the client does not need to know where the map resides. Speed of the application depends highly on the chosen IDL. Structs are faster for bulk data transfers, while interfaces can take advantage of location transparency, computed methods, and inheritance. Such speed considerations are the main reason why IDL definitions are very application specific even though IDL definitions hide many implementation details. The standardisation of IDL for specific application domains, such as genomic maps, allows for an independent developing of clients and servers. Interoperability between independent data providers and applications is only possible if agreement on standard interfaces is achieved. The common map IDL is an attempt to set such a standard. Mapplet demonstrates the usefulness of the common IDL for the purpose of a map viewer. The IDL is designed to deal with large maps of more than one thousand markers. Mapplet focuses on a quick overview of large maps.

---

[1] http://www.ooc.com/ob.html
[2] http://www.inprise.com/visibroker

Chapter 3

CORBA WRAPPERS

3.1 Introduction

In the previous chapter, the advantages of accessing molecular biology data using CORBA have been discussed from the perspective of a client. The aim of this chapter is to examine the server side of such applications in a systematic way. In general, today's molecular biology databases do not support CORBA interfaces. In order to allow such a database to interact with CORBA components it is therefore necessary to build a so-called wrapper – a program, which implements IDL interfaces for already existing *legacy systems*.

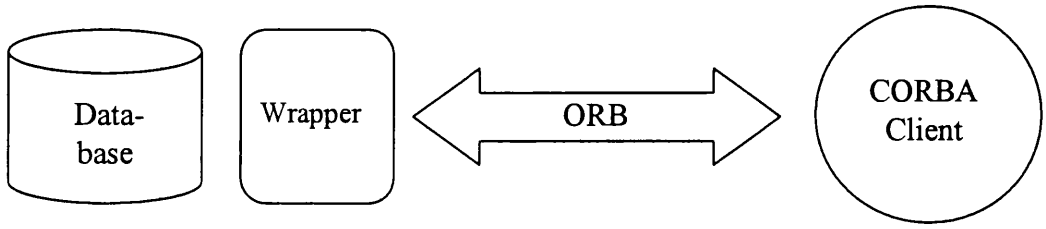


Figure 7: CORBA wrappers make existing DBs available through the ORB

The system depicted in Figure 7 is a classical three-tier architecture. The wrapper represents the middle-tier, insulating the client and data source from each other. Because the wrapper can already provide an application specific view of the data, it allows the client to concentrate on its main functionality. It is possible to speak of a *thin client* if most of the data transformations and processing is left to the middle-tier. The client remains simple and is mainly responsible for

data presentation. Thin clients are often desirable because they are easy to distribute via a network, typically executed within a Web browser. Another advantage of thin clients is that their simplicity makes it easier to replace them with the implementation from an independent provider. This issue makes them even more interesting in the context of CORBA since it is one of the main aims of CORBA to support componentry.

The two aspects addressed in this chapter are the interface and the implementation of CORBA wrappers for databases. The external interface of the wrapper defines its functionality and is specified using CORBA IDL. In the context here this means mainly data representation and access. The second aspect consists of a variety of issues concerning the actual implementation of the wrapper.

3.2 Interfacing Databases using CORBA IDL

How should data be represented in a CORBA environment? Should the IDL reflect the data model and if yes how? How can a CORBA wrapper support queries? The answer to these questions is crucial in this context and will therefore be examined in detail in this section.

3.2.1 Data Representation

Using CORBA IDL, data can be represented in many different ways. Each possibility has advantages and disadvantages depending on the type of application. These possibilities will be evaluated in this section with respect to performance, extensibility and, simplicity. There are in general four types of representations, which are summarised in Figure 8. The two principal design choices, which lead to this table are: generic types versus domain specific types and CORBA objects versus value types.

	Value-based	Object-based
Generic types	Strings, Sequence of Octets	Generic CORBA objects
Domain types	Structs, Unions, CORBA 3 Values	Domain CORBA objects

Figure 8: Types of Data Representations in IDL

3.2.1.1 Generic Types

Many existing systems use CORBA IDL only for the infrastructure of the system, while using generic types to represent the data itself [Dogac 96]. In this approach the IDL definitions do therefore not reflect the data model. This can be achieved for example by encoding the data objects in strings or byte sequences. The following string for example could represent an entry of a sequence database:

"((accession 'AC24123') (species 'human') (sequence 'ACGCGTTAATCGGC'))"

This is similar to the normal access of a relational database using an SQL prompt, where each result entry is returned as a string to the user. An example of a syntax for the encoding of objects in text files is the object interchange format OIF, defined in the ODMG 2.0 standard [Cattell 97].

The second generic possibility is to utilise a CORBA object to represent the data but to use a completely generic interface, which is independent from the structure of the represented object. The following IDL for example could be used for such a purpose:

```
interface DBObject {
    string getClassName();
```

```
    any getAttribute(in string attributeName);
};
```

The main advantage of generic approaches is the ease of maintenance. Using IDL, clients and servers have to be compiled together with the classes, which have been generated from the IDL definitions. This can become a major burden if the data model is large or changes frequently. Generic applications, which are not interested in the semantics of the data – e.g. a tool for displaying data in the form of a table or a query evaluator – are therefore often better served by generic data representations. However, for other more specialised applications, such as a map viewer, generic representations can be very cumbersome to use. In the case of the encoding the objects in sequence types the objects have to be parsed and brought into an internal representation.

3.2.1.2 Domain SpecificTypes

The second possibility is to use CORBA IDL not only for the infrastructure of the system but also to represent the data objects themselves. The main reason for representing data using domain specific IDL types is to simplify the task of writing clients. As seen above the generic approach requires the client to transform the data object into an internal representation, which can be used more conveniently. This problem is often referred to as impedance mismatch between the database language and the application programming language. If however the data model is directly expressed in CORBA IDL, then encoding, decoding and translation into an internal representation is done by the stubs and skeletons, which have been generated automatically by the IDL compiler. The client can access remote information like local programming language objects.

Similar to the generic case there are again two main possibilities depending on whether CORBA objects are used or value types. Of course, these different possibilities do not exclude each other – they simply meet different requirements of different clients.

3.2.1.2.1 Object-Based

The following example shows a possible representation of a protein sequence as a CORBA object:

```
interface ProteinSequence {
   attribute string accession;
   attribute string species;
   attribute string sequence;
};
```

This is perhaps the most natural representation. Every class of the data model is represented by a corresponding interface. Every database entry is represented as a CORBA object. The application program can use such a sequence object as if it were local. Because CORBA interfaces support directly both multiple inheritance and associations, interfaces can resemble a conceptual data model very naturally. However, this approach can be too slow for many applications, because every access to an attribute is a remote method invocation over the network. If the application program wants to get all three attribute values then three remote method invocations will occur. If the application requires access to many such objects the network load becomes unacceptable. Such fine grained data access has led many to conclude that CORBA as such is inefficient. In reality it is just one possibility, which is suitable in some situations and not suitable in others.

3.2.1.2.2 Value-Based

The second possibility is not to use CORBA objects to avoid the mentioned potential performance problems, but instead models the data as values such as IDL *structs*.

```
struct ProteinSequence {
   string accession;
   string description;
   string sequence;
```

```
}
```

In contrast to CORBA objects, structs are passed by value. Access to the individual fields is therefore local and very fast. On the other hand the application has to get the whole struct even if it is only interested in the accession field. 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 using IDL unions or anys. This problem will often occur if an extended entity-relationship model is used as the starting point for the IDL development, but interfaces are no option

The upcoming CORBA 3.0 standard has extended the IDL by a new type called *value*. CORBA 3.0 values are similar to structs but do additionally support inheritance. Even though, following the example of Java, values are restricted to single inheritance, this overcomes one of the most severe disadvantages of IDL structs.

3.2.1.3 Associations

Associations can be modelled in IDL using three different possibilities. Object references, logical keys and embedded structs. For each approach an example is given below.

3.2.1.3.1 Object References:

```
interface SwissProtSequence {
   attribute string accession;
   attribute string description;
   attribute string sequence;
   attribute EMBLSequence EMBL;
}
interface EMBLSequence { ... };
```

The main advantage of object references is their ease of use. The client can navigate from a protein sequence to an EMBL sequence easily by simply following a pointer.

3.2.1.3.2 Logical Keys:

This is the same example as above but this time only a logical key indicates, which EMBLSequence is referenced.

```
interface SwissProtSequence {
   attribute string accession;
   attribute string description;
   attribute string sequence;
   attribute string EMBLaccession;
}
```

The advantage of logical keys is that they do not only point to a specific implementation but to the appropriate EMBL sequence in any instance of the database. The disadvantage is less convenient access for the client, which has to know itself how to get the *EMBLSequence* using its accession number.

3.2.1.3.3 Embedded Structs:

Here the same example using embedded structs. Note that the *EMBLSequence* is always copied as a whole together with the parent struct *SwissProtSequence*.

```
struct SwissProtSequence {
   string accession;
   string description;
   string sequence;
   EMBLSequence EMBL;
}
struct EMBLSequence { ... };
```

3.2.1.3.4 Multiple Associations:

One-to-n and n-to-m relationships can be modelled using the same principal approaches but do additionally require the usage of IDL types such as sequences or arrays .

3.2.1.4 Evaluation

This section summarises the advantages and disadvantages of IDL structs, interfaces and CORBA 3.0 value types.

3.2.1.4.1 Time

Value based data transfer is quicker if a large amount of data has to be transferred. This applies to both structs as well as the new CORBA3.0 value types. Object based access, however, allows for a more precise selection of the downloaded data. This is an advantage when not all data of a specific object is needed and only few objects are involved.

3.2.1.4.2 Space

If value based data representations are used, the whole data is copied to the client. Therefore value based representations are in most cases less space efficient. There are however cases when the size of an CORBA object reference alone is bigger than the data contained in the object. In this case there would be no advantage from using interface based representations.

3.2.1.4.3 Extensibility

Inheritance is one of the most important technique to extend existing code. Structs do not support inheritance, while both interfaces and the CORBA3.0 value types do. CORBA 3.0 value types therefore overcome one of the most important disadvantages of structs.

3.2.1.4.4 Simplicity/Readability

Again inheritance is an important way to structure and simplify code. Therefore interfaces and CORBA3.0 values are again superior to structs in this respect.

3.2.1.5 Applications

It is also possible to view the differences of the different approaches from the perspective of different types of applications.

3.2.1.5.1 Browsing

Simple navigation and browsing of CORBA objects, is best supported by object based data representations if no large amounts of data need to be transferred.

3.2.1.5.2 Bulk Data Transfer

If large amounts of data need to be transferred then value-based representations are most efficient.

3.2.1.5.3 Data Integration

As seen in the interface example for biological sequences, CORBA object references provide an easy way of interconnecting objects from different independent and possibly heterogeneous data sources – a point which has been made for example in [Markowitz 95]. Even though this approach is easy to implement, it does support only simple navigation but no queries. It is in many respects very similar to the interconnection of Web pages using URLs instead of object references. In contrast, structs and CORBA 3.0 values do not provide a similar simple mechanism of integration of data from independent sources.

3.2.1.5.4 Object Sharing

One interesting new application opened by CORBA is the possibility to share distributed objects among several clients. An example would be a CORBA object which represents an alignment of nucleotide sequences. It is possible to support the decision about what is the correct alignment by making an assumption about a taxonomic tree of the species involved. If the tree changes, the alignment will in general change as well. This behaviour can be naturally modelled when the alignment viewer and the tree editor both hold object references to the same alignment object.

3.2.2 Data Models

When using IDL to represent domain objects it is tempting to treat it just like a data modelling language. This is especially true when using interface-based representations, which allow for a very natural one-to-one mapping between a conceptual model and IDL. It is however important to distinguish between these two. The following picture depicts the relationship of database schema, conceptual model, and the IDL.

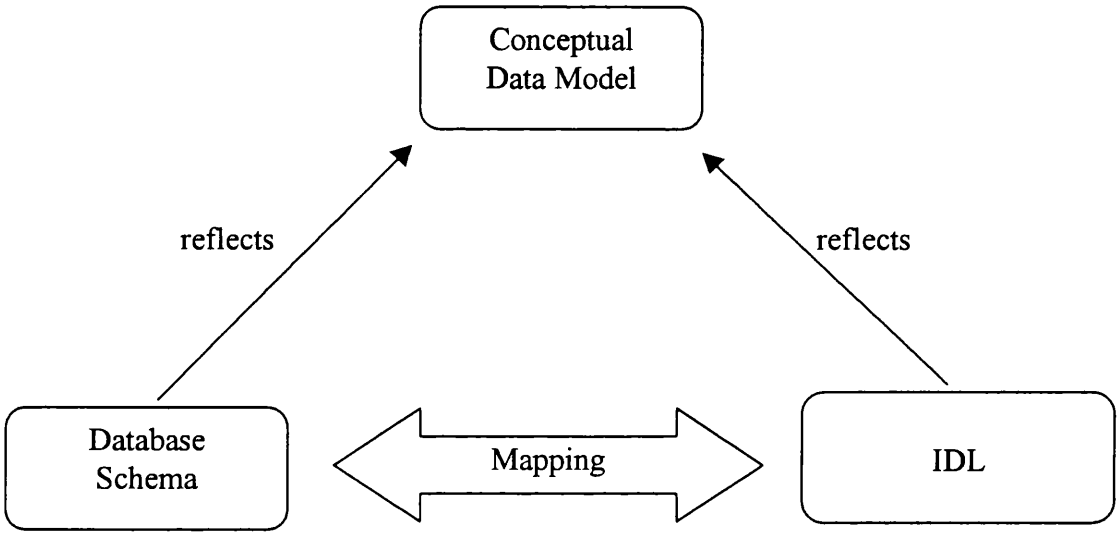


Figure 9: IDL vs. Data Model

While the conceptual model is the abstract, logical, and global structure of the domain data, the IDL merely represents an application specific API. Therefore, the IDL usually deals only with a subset of the conceptual model. Furthermore it already has the access paths encoded used by a client. Because several different access paths are often needed within the same application, IDL tends to be redundant and more complicated. It is worth remembering that similar arguments hold for the database schema, which often have to be denormalised in order to speed up specific applications. All this can result in a non-trivial relationship between database schema and IDL, which complicates the implementation of a wrapper whose task it is to implement the mapping between these two.

3.2.3 Queries

The ability to express queries is clearly an important aspect for every database. This aspect is standardised by OMG through its Query Service specification [OMG 98]. The central element of this IDL is a CORBA object of the type *Query Evaluator*. A query evaluator takes a query string as input and returns a reference to a collection object, which represents the result. This result collection gives access to its elements through another CORBA object: the Iterator. The query service does not specify how the elements of the result should look; the iterator always returns the IDL type *any*. It can therefore be combined with any of the above listed data representations. Furthermore the query service can be used with any possible query language depending on the underlying database system. Result collections can themselves support the query evaluator interface allowing to narrow a query in an iterative way.

Query support is common when generic data representations are used. JDBC is a well known example for such an API in the case of Java. However, if CORBA IDL is used to represent the data it is not obvious whether and how ad-hoc queries should be supported. Simple navigation and methods providing access to pre-canned queries are sufficient for many applications. On the other hand it seems clear that some applications would benefit from query support leaving the question how this can be done best. The main question is in what terms the query should be expressed. There are essentially three possibilities:

- In terms of the schema of the underlying database
- In terms of a conceptual data model
- In terms of the IDL itself

To base the queries on the schema of a database is probably the easiest to implement solution. The disadvantage is that the user has to know both the schema and the IDL as well as

the connection between these two. The approach is therefore only realistic if schema and IDL resemble each other closely.

Very similar arguments hold if a conceptual model is used instead of the schema. The advantage might be that a conceptual model has better chances to resemble the IDL. The reason is that unlike some database schemas – such as a relational one - it can support IDL concepts such as inheritance and associations.

In contrast, to express queries in terms of the IDL itself has the advantage that the user needs to know the IDL only. The disadvantage is however that – as discussed before – the IDL tends to be more complicated and less readable than a conceptual model. This approach might therefore be better suited for small specialised applications.

3.3 Implementation

The implementation and maintenance of CORBA wrappers for biological databases can be a challenging problem. This section discusses several issues, which need to be addressed in this context.

3.3.1 Generation of CORBA Wrappers

When using CORBA IDL to represent the domain objects of an application, it is one of the main tasks of the CORBA wrapper to implement the mapping between the database structures on one hand and the corresponding IDL constructs on the other. This can be done in several different ways.

3.3.1.1 Manual Implementation

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. This is mainly code to access the database through a database gateway such as JDBC. The problem is that 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 IDL based ad hoc queries can be supported in such a setting.

3.3.1.2 Automatic Code Generation

Another possibility is the automatic generation of IDL and CORBA server based on the schema of the underlying database. The problem with this approach is that the automatically generated IDL is usually not what we need for a 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 [Barillot 99a]. Another reason is, as detailed above, 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.

3.3.1.3 Semi-automatic Code Generation

Finally, it is possible to try to find a compromise between the completely manual and the completely automatic implementation. This approach will be called semi-automatic here. The central idea is as follows: First the developer decides on an IDL that ideally suits his application. Then he specifies a set of rules, which describe the customised mapping between database schema and the target IDL. Using the mapping rules the CORBA wrapper can now be generated automatically. The semi-automatic server generation has the advantage, that the developer has a large degree of freedom when choosing the target IDL but still leaves the most difficult part of the implementation to an automatic code generator. Also, it is still possible to support ad hoc queries, which are expressed in terms of the IDL data types.

3.3.1.4 Query Support

It is relatively easy to support queries by simply giving access to the underlying query system of a CORBA wrapped database. However it is more difficult to support ad hoc queries, which are based on the IDL definitions of the server. Especially in the case of a manually implemented CORBA wrapper it is not obvious whether and how such queries could be supported. The situation is better in the case of automatically and semi-automatically generated servers. The mapping rules are in these cases known explicitly and can be used to translate queries and results from and to the IDL representation.

3.3.1.5 Code Maintenance

As discussed before, manually implemented CORBA wrappers are difficult to maintain if either the schema of the database or the target IDL change. However, automatically and semi-automatically generated CORBA servers also have their problems with respect to code maintenance. The automatically generated server will in most cases not provide the IDL needed for a specific application. It is therefore usually necessary to modify it by hand in order to adapt it. This will not happen as often in the semi-automatic case but even there it is sometimes desirable to add the implementation of convenience methods. Therefore in both cases manually and automatically generated code have to coexists. When the automatic part of the code is regenerated because of a schema change, it has to be recombined with the hand coded modifications. To avoid error prone code merging it is in the experience of the author better to use either inheritance or delegation to combine the two types of code.

3.3.2 Registering Large Numbers of Objects

Databases can contain millions of objects. If the CORBA wrapper used interface based representations, the ORB has to be able to keep track of the connection between these database objects and the corresponding CORBA object references. It is therefore not practical in this case

to use the standard individual object registration provided by CORBA. Instead it is necessary to use an implicit registration method. The general idea is to have the object identifier of the database object encoded in the CORBA object reference. Many ORBs implemented an own proprietary way of implicitly registering large numbers of objects. The recently by OMG adopted Portable Object Adapter (POA) now standardises this possibility.

3.3.3 Caching

Typically, every CORBA object is represented by one corresponding object of the implementation language of the server; e.g. a C++ object. In the case of a database wrapper this means that the C++ server contains a copy of the database entry (Figure 10a). If a client invokes for the first time a method on an object reference the corresponding database entry is loaded and the C++ object is instantiated. It stays in memory for subsequent method invocations until the server actively removes it to prevent the server from maintaining a copy of the whole database. This approach can be fast because the CORBA server is a cache for database entries but it makes a policy for garbage collection necessary.

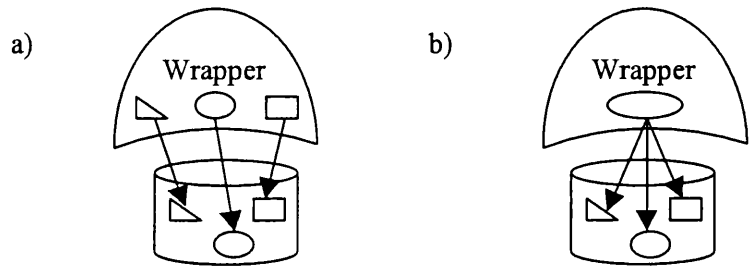


Figure 10: CORBA wrappers with a) and without state b)

In the second possibility (Figure 10b) the CORBA server is stateless. One language object implements a set of different CORBA objects each representing another database entry. For every request the server extracts the object identifier from the request in order to find out which

CORBA object it has to emulate. In this approach no data eviction strategy is needed but every request requires access to the underlying database.

3.3.4 Object-Relational Mapping

Relational database management systems (RDBMS) are still the standard today: many major and minor data collections in molecular biology, like EMBL [Stoesser 99] or IXDB [Leser 98b], utilise a relational database management system. Since the relational model is very different from an object-oriented IDL, object-relational mapping [Wiederhold 86] becomes the central problem for CORBA wrappers. There are different tools, which can help to define an object model on top of a relational database. An example is the object-relational mapping tool "Persistence". It follows the philosophy of object-oriented databases in the sense that it makes database entries directly available as C++ objects. Persistence is therefore well suited to implement CORBA servers, which cache database entries in main memory. The main disadvantage of this tool is that the developer has not much freedom to define object schemas. The Persistence object schema is very close to the original table structure of the database. Relational views and a certain amount of hand coding are often necessary to do the required mapping. Another tool for object-relational mapping is OPM [Chen 95]. In contrast to Persistence it supports the mapping of object-oriented queries to SQL queries. It is therefore better suited for stateless wrappers and for the implementation of a CORBA query service.

3.4 Conclusion

Usually, existing data collections do not provide CORBA interfaces. In order to make such data collections available in a CORBA environment it is necessary to implement a so-called CORBA wrapper. The most important design question, which a developer of such a wrapper faces, is the question of how to represent data in IDL. The answer to this question depends very much on the

concrete application, and will have a major impact on the performance of the distributed application. One way to find the best IDL is often by a use-case analysis [Jacobson 92].

The implementation of CORBA wrappers by hand is tedious and error prone. Automatically generated wrappers on the other hand are too inflexible in the choice of the implemented IDL. Semi-automatic wrapper generation offers here a good compromise. The IDL can largely be influenced by the developer while the most complicated part of the code is automatically generated.

Chapter 4

A MAPPING LANGUAGE FOR RELATIONAL DATABASES

4.1 Introduction

In this section a high-level language is presented, designed to describe mappings between relations on one side and CORBA IDL types on the other side. The IDL types are completely specified by the mapping definitions. The mapping language is declarative and is intended to guide automatic query translation and data transformations. In order to facilitate the task, the IDL has been restricted to most often used constructs of the interface definition language of CORBA V2.0:

- Modules
- Interfaces and object references
- The template type sequence
- The constructed types enumeration and struct
- The base types long, float, string, and boolean

Interfaces are restricted to read-only attributes. Other types such as arrays and unions can easily be incorporated using a similar method.

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:

- The simple values long, float, string, boolean and enum are represented directly by exactly 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.

The here presented mapping language has been completely developed by the author. An earlier and less general version of the language has been published in [Jungfer 99a].

4.2 The Mapping Language

4.2.1 Views

The top-level construct of the mapping language is the definition of an IDL view. A view has a name and is associated with a table and the mapping for an IDL type. The IDL type can be in this case 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.

```
<idl_view> ::= ( VIEW <view_name>
                 TABLE <table_name>
```

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

4.2.2 Interfaces

The mapping definition for interfaces specifies first the IDL name of the interface. Then all interfaces from which it inherits are specified in the "EXTENDS" clause. 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. The following "SUB" clause allows the specification of non-overlapping "sub-views" (See the examples in section 3.2 of this chapter for the usage of this construct). Then the primary key for the main table of this view 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>* ) ]
                     [ SUB ( <view_name> ) ]
                     [ KEYS ( <column_name>+ )
                       ( <attribute_name> <data_type_mp> )+ ]
                   )
```

4.2.3 Data Types

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

```
<data_type_mp> ::=   <base_type_mp>
                   | <enum_mp>
                   | <reference_mp>
                   | <struct_mp>
```

```
                    | <sequence_mp>
```

4.2.4 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> )+ )
```

4.2.5 View References

Inside the mapping definition of an IDL view it is possible to refer to an already existing view. Depending on the type of the referenced view, this is either used to represent an embedded struct or an object reference. While an embedded struct can also be defined directly inside the outer type, a view reference is the only possibility to specify a CORBA object reference using this mapping language. For a view reference 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>+ ) )
```

4.2.6 Sequences

The data, which belongs to a sequence, is multi-valued and therefore stored in a different table. This table together with the connecting columns is specified in the "table_mp" construct of the mapping language. Another possibility is the usage of a view reference where the new table is already specified by the referenced view. An exception is the case where the subtype of the sequence is an object reference. In this case the new table is already specified by the referenced

view (see the examples in 4.3.1 for the usage of these two possibilities). Since sequences are ordered it is additionally necessary to specify an order-by-clause.

```
<sequence_mp> ::= ( SEQUENCE <scoped_sequence_name>
                    <table_mp> | <reference_mp>
                    ORDERBY <order_by_clause> )

<table_mp> ::= ( TABLE <table_name>
                 KEYS ( <column_name>+ ) ( <column_name>+ )
                 <data_type_mp> )
```

4.2.7 Base 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.

```
<base_type_mp> ::=  ( BOOLEAN <column_name> <true_value> )
                  | ( LONG    <column_name> )
                  | ( FLOAT   <column_name> )
                  | ( STRING  <column_name> )
```

4.2.8 Enumerations

The mapping for an enumeration is specified by giving the scoped name of the enumeration and a list of specifications for every possible value of the enumeration type (enumerator). Each enumerator is associated with a column name and the value, which represents the enumerator in the database.

```
<enum_mp> ::= ( ENUM <scoped_name>
                COLUMN <column_name>
                ( <enumerator> <enum_value> )+ )
```

4.3 Examples

The examples are divided in two parts. The first part contains the basic mappings for interfaces, structs, enumeration and sequences. The second part deals with the representation of inheritance.

4.3.1 Basic Mappings

The following conceptual model of genome maps and markers is used throughout this section (Figure 11). It is a simplified version of the model in the second chapter. A map has a name, a type and contains a possibly large number of markers. Each marker, which is assigned to a map has a position on this map, a rank, and a boolean flag indicating whether it belongs to the framework of the map or not. Each marker on its own has a name and a type and can occur on several different maps.

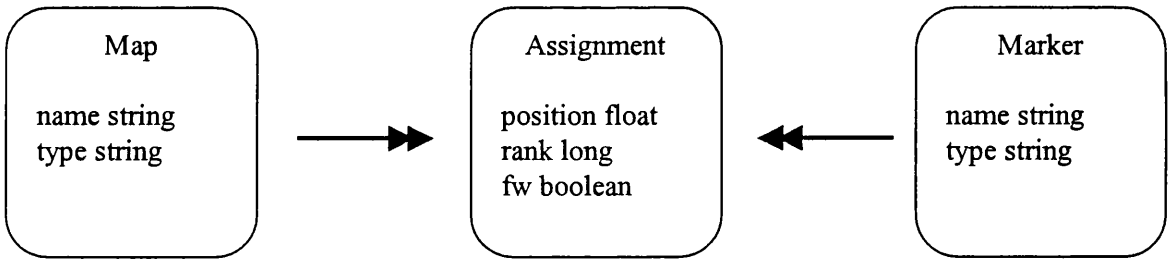


Figure 11: Conceptual Model

The following relational schema (Figure. 12) represents this model. The attribute "ID" is the primary key of the tables "MAPS" and "MARKERS". They correspond to the foreign keys "MAP_ID" and "MARKER_ID" of the table "MAP_MARKERS". The types have been omitted but are assumed to be compatible with the types of the conceptual model.

MAPS:

ID	TYPE

MARKERS:

ID	TYPE

MAP_MARKERS:

MAP_ID	MARKER_ID	FW	RANK	POSITION

Figure 12: Relational Schema.

4.3.1.1 Struct

The following CORBA IDL is one possibility to represent the assignment class of the conceptual model in figure 11.

```
module Genome {
  struct Assignment {
    string map_name;
    string marker_name;
    float position;
    long rank;
    boolean fw;
  };
};
```

This simple IDL could be mapped using the following view definition:

```
( VIEW Assignments
  TABLE MAP_MARKERS
  ( STRUCT Genome::Assignment
    ( map_name    ( STRING MAP_ID ) )
    ( marker_name ( STRING MARKER_ID ) )
```

```
    ( position      ( FLOAT POSITION ) )
    ( rank          ( LONG RANK ) )
    ( fw            ( BOOLEAN FW Y ) )
  )
)
```

In this example, a view with the name "Assignments" has been defined. The data for the view can be found in the table "MAP_MARKERS". The view is represented by the IDL struct "Assignment" in the module "Genome". The struct member "map_name" is represented by the table attribute "MAP_ID" and has the IDL type string. Note that in the case of the boolean struct member "fw", it is additionally necessary to specify the value which represents true in the relational database; in this case "Y".

4.3.1.2 Interface

Mapping specifications for interfaces are very similar to those of structs. Here an example for the class "Map" of the conceptual model.

```
module Genome {
  interface Map {
    readonly attribute string name;
    readonly attribute string type;
  };
};
```

The following mapping specification implements this IDL:

```
( VIEW Maps
  TABLE MAPS
  ( INTERFACE Genome::Map
    EXTENDS ()
    KEYS ( ID )
    ( name ( STRING ID ) )
    ( type ( STRING TYPE ) )
```

```
  )
)
```

Mapping definitions of structs and interfaces are identical apart from the additional specification of extended interfaces and keys. In this case no other interface is extended by "Genome::Map". The primary key of the table "MAPS" is specified in the "KEYS" clause and can be used by the object adapter of the object request broker.

4.3.1.3 Nested Struct – One Table

In this example, the same information as in example 4.3.1.1 is represented but this time using two nested structs. The example is a bit artificial but demonstrates the point.

```
module Genome {
  struct Names {
    string map_name;
    string marker_name;
  };
  struct Assignment {
    Names names;
    float position;
    long rank;
    boolean fw;
  };
};
```

The corresponding mapping definition directly reflects the nested IDL structs.

```
( VIEW Assignments
  TABLE MAP_MARKERS
  ( STRUCT Genome::Assignment
    ( names     ( STRUCT Genome::Names
                  ( map_name    (STRING MAP_ID ) )
                  ( marker_name (STRING MARKER_ID ) ) )
    ( position ( FLOAT POSITION) )
    ( rank      ( LONG RANK ) )
```

```
    ( fw        ( BOOLEAN FW ) )
  )
)
```

4.3.1.4 Nested Struct – Two Tables

In this example, the "Assignment" struct contains not only the marker name but the full marker information in an own struct.

```
module Genome {
  struct Marker {
    string name;
    string type;
  };
  struct Assignment {
    string map_name;
    Marker marker;
    float position;
    long rank;
    boolean fw;
  };
};
```

The following mapping definition creates two CORBA views; one for the markers and one for the assignments. The "Assignments" view refers to the "Markers" view using the "REFERENCE" construct of the mapping language.

```
( VIEW Markers
  TABLE MARKERS
  ( STRUCT Genome::Marker
    ( name ( STRING ID ) )
    ( type ( STRING TYPE ) )
  )
)
( VIEW Assignments
  TABLE MAP_MARKERS
```

```
  ( STRUCT Genome::Assignment
    ( map_name ( STRING map_id ) )
    ( marker   ( REFERENCE Markers
                 KEYS (MARKER_ID) (ID) )
    ( position ( FLOAT POSITION) )
    ( rank     ( LONG RANK ) )
    ( fw       ( BOOLEAN FW ) )
  )
)
```

4.3.1.5 Object Reference

This example represents the same information as the previous example but uses an interface for the markers.

```
module Genome {
  interface Marker {
    readonly attribute string name;
    readonly attribute string type;
  };
  struct Assignment {
    string map_name;
    Marker marker;
    float position;
    long rank;
    boolean fw;
  };
};
```

Again, two views are defined. The only difference to the previous example is that the view "Markers" is now represented by an interface. The same reference construct can be used as before but this time resulting in a CORBA object reference.

```
( VIEW Markers
  TABLE MARKERS
  ( INTERFACE Genome::Marker
```

```
    EXTENDS ()
    KEYS (ID)
    ( name ( STRING ID ) )
    ( type ( STRING TYPE ) )
  )
)
( VIEW Assignments
  TABLE MAP_MARKERS
  ( STRUCT Genome::Assignment
    ( map_name ( STRING map_id ) )
    ( marker   ( REFERENCE Markers
                 KEYS (MARKER_ID) (ID) )
    ( position ( FLOAT POSITION) )
    ( rank     ( LONG RANK ) )
    ( fw       ( BOOLEAN FW ) )
  )
)
```

4.3.1.6 Sequence

Here the struct "Map" contains a sequence of strings. The contents of a sequence is always on a separate table.

```
module Genome {
  typedef sequence <string> Markers;
  struct Map {
    string name;
    string type;
    Markers markers;
  };
};
```

The following view definition implements this IDL:

```
( VIEW Maps
  TABLE MAPS
```

```
  ( STRUCT Genome::Map
    ( name   ( STRING ID ) )
    ( type   ( STRING TYPE ) )
    ( markers ( SEQUENCE Genome::Markers
                ( TABLE MAP_MARKERS
                  KEYS (ID) (MAP_ID)
                  ( STRING MARKER_ID )
                  ORDERBY POSITION ) )
  )
)
```

The mapping specification of the sequence contains the connecting columns of the two involved tables, the mapping definition of the sub-type of the sequence, and an order-by clause.

4.3.1.7 Sequence of Structs – Two Tables

In this example the struct "Map" contains a sequence of "Assignment" structs. Note that the member "map_name" of the struct "Assignment" has been omitted in this example because it would be redundant.

```
module Genome {
  struct Assignment {
    string marker_name;
    float position;
    long rank;
    boolean fw;
  };
  typedef sequence <Assignment> Assignments;
  struct Map {
    string name;
    string type;
    Assignments assignments;
  };
};
```

Here the corresponding mapping definition if only one view is used:

```
( VIEW Maps
  TABLE MAPS
  ( STRUCT Genome::Map
    ( name         ( STRING ID ) )
    ( type         ( STRING TYPE ) )
    ( assignments ( SEQUENCE Genome::Assignments
                    ( TABLE MAP_MARKERS
                      KEYS (ID) (MAP_ID)
                      ( STRUCT Genome::Assignment
                        ( marker_name ( STRING MARKER_ID ) )
                        ( position    ( FLOAT POSITION ) )
                        ( rank        ( LONG RANK ) )
                        ( fw          ( BOOLEAN FW Y ) )
                      )
                      ORDERBY POSITION ) ) )
  )
)
```

Alternatively, if a separate view for assignments is wanted:

```
( VIEW Assignments
  TABLE MAP_MARKERS
  ( STRUCT Genome::Assignment
    ( marker_name ( STRING MARKER_ID ) )
    ( position    ( FLOAT POSITION ) )
    ( rank        ( LONG RANK ) )
    ( fw          ( BOOLEAN FW Y ) )
  )
)
( VIEW Maps
  TABLE MAPS
  ( STRUCT Genome::Map
    ( name         ( STRING ID ) )
    ( type         ( STRING TYPE ) )
    ( assignments ( SEQUENCE Genome::Assignments
                    ( REFERENCE Assignments
```

```
                    KEYS (ID) (MAP_ID) )
                  ORDERBY POSITION ) ) )
  )
)
```

In this case the "REFERENCE" construct is used instead of the "TABLE" construct.

4.3.1.8 Sequence of Structs – Three Tables

Here the struct "Map" contains a sequence markers instead of a sequence of assignments as in the previous example.

```
module Genome {
  struct Marker {
    string name;
    string type;
  };
  typedef sequence <Marker> Markers;
  struct Map {
    string name;
    string type;
    Markers markers;
  };
};
```

Note that in contrast to the previous example, the "TABLE" construct as well as the "REFERENCE" construct have to be used in the mapping definition of the view "Maps".

```
( VIEW Markers
  TABLE MARKERS
  ( STRUCT Genome::Marker
    ( name ( STRING MARKER_ID ) )
    ( type ( STRING TYPE ) )
  )
)
```

```
( VIEW Maps
  TABLE MAPS
  ( STRUCT Genome::Map
    ( name    ( STRING ID ) )
    ( type    ( STRING TYPE ) )
    ( markers ( SEQUENCE Genome::Markers
                ( TABLE MAP_MARKERS
                  KEYS (ID) (MAP_ID)
                  ( REFERENCE Markers
                    KEYS (MARKER_ID) (ID) ) )
                ORDERBY POSITION ) )
  )
)
```

4.3.1.9 Interface Containing a Sequence of Nested Structs

Here "Map" is an interface containing a sequence of nested "Assignment" and "Marker" structs.

This example could be a realistic IDL for the complete relational schema.

```
module Genome {
  struct Marker {
    string name;
    string type;
  };
  struct Assignment {
    Marker marker;
    float position;
    long rank;
    boolean fw;
  };
  typedef sequence <Assignment> Assignments;
  interface Map {
    readonly attribute string name;
    readonly attribute string type;
    readonly attribute Assignments assignments;
  };
};
```

Here the corresponding mapping definitions using three separate views.

```
( VIEW Markers
  TABLE MARKERS
  ( STRUCT Genome::Marker
    ( name ( STRING ID ) )
    ( type ( STRING TYPE ) )
  )
)
( VIEW Assignments
  TABLE MAP_MARKERS
  ( STRUCT Genome::Assignment
    ( map_name ( STRING map_id ) )
    ( marker   ( REFERENCE Markers
                 KEYS (MARKER_ID) (ID) )
    ( position ( FLOAT POSITION) )
    ( rank     ( LONG RANK ) )
    ( fw       ( BOOLEAN FW ) )
  )
)
( VIEW Maps
  TABLE MAPS
  ( INTERFACE Genome::Map
    EXTENDS ()
    KEYS (ID)
    ( name (STRING ID) )
    ( type (string TYPE)
    ( assignments ( SEQUENCE Genome::Assignments
                    ( REFERENCE Assignments
                      KEYS (ID) (MAP_ID) )
                    ORDERBY POSITION ) )
  )
)
```

4.3.1.10 Enumeration

In the previous examples the type of a map was represented by a string. If there is only a small number of possible different types then an enumeration could be used instead.

```
module {
  enum MapType { genetic, radiation_hybrid }
  interface Map
    readonly attribute string name;
    readonly attribute MapType type;
  };
};
```

In the mapping definition, each possible enumerator is associated with exactly one value in the database.

```
( VIEW Maps
  TABLE MAPS
  ( INTERFACE Genome::Map
    EXTENDS ()
    KEYS ( ID )
    ( name ( STRING ID ) )
    ( type ( ENUM Genome::MapType
             COLUMN TYPE
             ( genetic genetic )
             ( radiation_hybrid rh ) ) )
  )
)
```

4.3.2 Inheritance

In the examples so far, genome maps had a type, represented by a string or an enumeration. If the map types form a hierarchy, possibly having different attributes for different types of maps, then a different conceptual model would be more natural. Figure 12 depicts a conceptual model

where "Physical Map" and "Genetic Map" are sub-classes of "Map". Genetic maps have the additional attribute "lod_score".

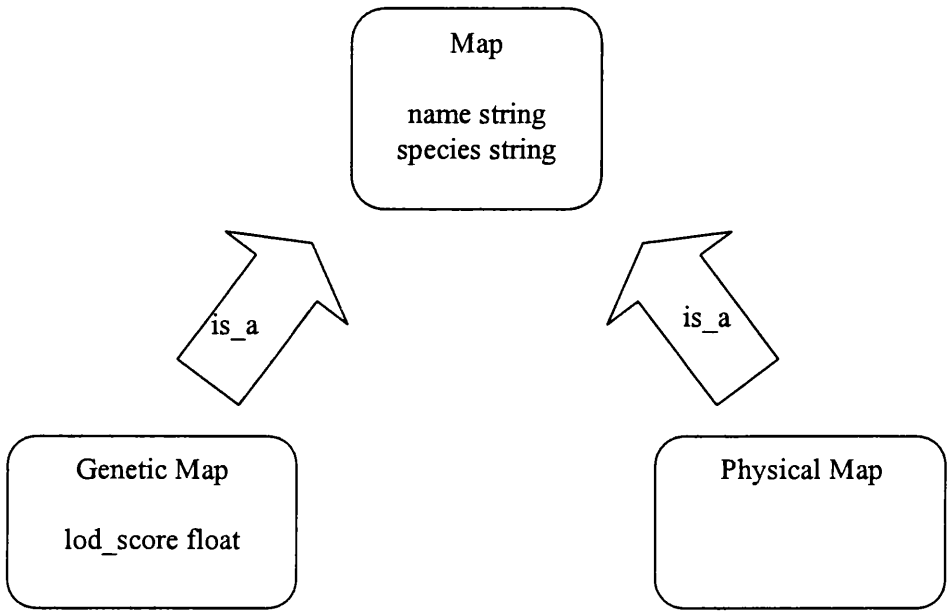


Figure 13: Conceptual Model - Inheritance

There are two common ways how to represent such a hierarchy in a relational database: *horizontal partitioning* and *vertical partitioning* [Ceri 84].

4.3.2.1 Horizontal Partitioning

A schema using horizontal partitioning is depicted in Figure 14. The table "MAPS" contains only entries for maps, which are neither assigned to be genetic nor physical maps. The table "GENETIC_MAPS" contains all genetic maps and the table "PHYSICAL_MAPS" contains all physical maps. Advantage of this possibility is an efficient access to the sub-classes; disadvantage is that an access to the super-class "Map" requires a union of three tables. Horizontal partitioning is the only directly supported representation of the mapping language.

MAPS:

ID	SPECIES
M1	Mouse

GENETIC_MAPS:

ID	SPECIES	LOD_SCORE
H2	Human	2.25

PHYSICAL_MAPS:

ID	SPECIES
R3	Rat

Figure 14: Horizontal Partitioning.

4.3.2.2 Vertical Partitioning

Figure 15 depicts a schema using vertical partitioning. The table "MAPS" contains entries for all maps, while the tables "GENETIC_MAPS" and "PHYSICAL_MAPS" contain only keys and additional attributes. This time access to the super-class "Map" is efficient, whereas access to one of the sub-classes requires a costly join operation. Vertical partitioning is not directly supported by the mapping language; it is instead necessary to use relational views to define the mapping.

MAPS:

ID	SPECIES
M1	Mouse
H2	Human
R3	Rat

GENETIC_MAPS:

ID	LOD_SCORE
H2	2.25

PHYSICAL_MAPS:

ID
R3

Figure 15: Vertical Partitioning

4.3.2.3 Interfaces

The following IDL is one possibility to represent the conceptual schema in Figure 13 using interfaces:

```
module {
  interface Map {
    readonly attribute string name;
    readonly attribute string species;
  };
  interface GeneticMap : Map {
    readonly attribute float lod_score;
  };
  interface PhysicalMap : Map {
  };
};
```

If horizontal partitioning is used (Figure 14), the mapping is straightforward. Note that the "EXTENDS" clause only specifies the complete IDL type of an interface, while the "SUB" clause defines the subset relationship of the involved views.

```
( VIEW Maps
  TABLE MAPS
  ( INTERFACE Genome::Map
    SUB ( GeneticMaps PhysicalMaps )
    KEYS ( ID )
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
  )
)
( VIEW GeneticMaps
  TABLE GENETIC_MAPS
  ( INTERFACE Genome::GeneticMap
    EXTENDS ( Genome::Map )
    KEYS ( ID )
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
    ( lod_score ( FLOAT LOD_SCORE ) )
  )
)
( VIEW PhysicalMaps
  TABLE PHYSICAL_MAPS
```

```
  ( INTERFACE Genome::PhysicalMap
    EXTENDS ( Genome::Map )
    KEYS ( ID )
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
  )
)
```

4.3.2.4 Structs

CORBA IDL structs do not support inheritance. If however, performance considerations require the usage of structs to represent a class hierarchy, then it is still possible to mimic some of the relevant behaviour in a crude way. There are two separate issues: the type hierarchy and the sub/super-set relationship. The best one can do with respect to the type hierarchy is "cut and paste inheritance" as in the following IDL.

```
module Genome {
  struct Map {
    string name;
    string species;
  };
  struct GeneticMap {
    string name;
    string species;
    float lod_score;
  };
  struct PhysicalMap {
    string name;
    string species;
  };
};
```

Even though each struct has the appropriate members, a compiler would of course not allow to use a "GeneticMap" in a place where a struct of the type "Map" is expected. The following mapping definition implements the IDL above:

```
( VIEW Maps
  TABLE MAPS
  ( STRUCT Genome::Map
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
  )
)
( VIEW GeneticMaps
  TABLE GENETIC_MAPS
  ( STRUCT Genome::GeneticMap
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
    ( lod_score ( FLOAT LOD_SCORE ) )
  )
)
( VIEW PhysicalMaps
  TABLE PHYSICAL_MAPS
  ( STRUCT Genome::PhysicalMap
    ( name    ( STRING ID ) )
    ( species ( STRING SPECIES ) )
  )
)
```

In contrast to the mapping of interfaces, there is no "SUB" clause available for the mapping of structs. The reason is that the different struct types are not compatible with each other so that a CORBA view using one struct type can not just include another CORBA view using another struct type. Therefore SQL views have to be used to achieve the required one-to-one relationship between a struct and a relational table. Figure 16 depicts such views for the mapping definitions above.

MAPS:

ID	SPECIES
M1	Mouse
H2	Human
R3	Rat

GENETIC_MAPS:

ID	SPECIES	LOD_SCORE
H2	Human	2.25

PHYSICAL_MAPS:

ID	SPECIES
R3	Rat

Figure 16: SQL Views for the Mapping of Structs

Using the virtual tables of Fig. 15, all maps are included in the view "Maps" *and* the tables for genetic and physical maps contain all attributes belonging to that map. The disadvantage of this solution, in comparison with the previous interface example, is that a client, which retrieves a struct using the view "Maps" can not directly decide whether this struct has a more specific sub-type. Also such a struct cannot be narrowed to a more specific type as it is the case for object references.

4.4 Discussion

In this section the advantages and disadvantages of the chosen mapping language are discussed and compared to alternative approaches.

4.4.1 Two-stage Mapping using Relational Views

The examples of *vertical partitioning* and *inheritance for structs* in section 3.2 have demonstrated two cases where the presented mapping language is not sufficient to express the required mapping. Instead it was necessary to use relational views as an intermediate mapping step (Figure 17).

Figure 17: Two-stage Mapping using SQL Views

There are of course many other cases where this is true especially since the aim of the approach is to allow for a maximal freedom in the choice of the relational schema and in the choice of the IDL. The advantages of splitting of the responsibilities between relational views and mapping language are:

- A powerful existing mechanism is utilised instead of duplicating functionality.
- Reuse is facilitated. Not only can different IDL views share the same relational views but also other applications can do so. In fact, in many cases the relational views will already exist when an additional CORBA access layer is added.
- CORBA servers are better insulated from schema changes in the relational database.
- A simple and pragmatic criterion is provided to decide the question of what should be part of the mapping language and what should be excluded.

The disadvantages on the other hand are:

- The mapping information has to be maintained at two different places.
- Even though the here presented approach is intended to provide a read-only access to the database, it is of course possible to employ the same strategy to do update operations. Update operations however, might benefit from more explicit control over the possible mappings.

An alternative would be to try to enrich the mapping language to allow for other common mappings such as the mentioned vertical partitioning. There is probably no reason to be too dogmatic about such extensions as long as the language becomes easier to use. Such constructs would be orthogonal to the here presented ones but not provide anything new in the context of this thesis since they have nothing to do with CORBA IDL and merely reimplement existing SQL features. The problem of updating relational views is a research topic in its own right and is treated for example in [Date 86].

4.4.2 Usage of an Intermediate Object Model

Another option is the usage of an object model as an intermediate mapping step (Figure 18).

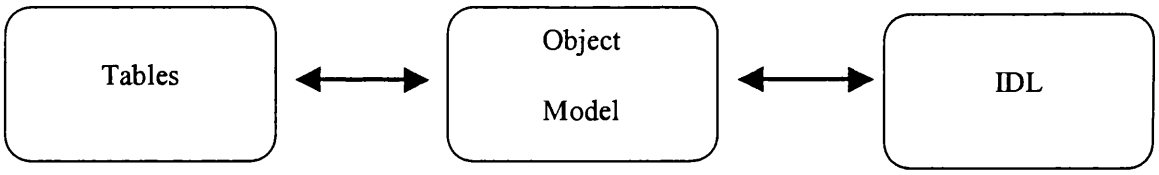


Figure 18: Two-stage Mapping Using an Intermediate Object Model

One problem is that the object oriented model is more complex than the relational model. Therefore an equivalent mapping language for the mapping between an object-oriented model and CORBA IDL is necessarily more complex. This approach is only justified if the object model and the mapping to the object model already exist and the final mapping step from the object model to IDL is simple.

4.4.3 Logic as a Mapping Language

Instead of the here presented mapping language, other possibilities could have been employed. The most general method to describe the mapping between CORBA IDL and tables of a relational database would be the direct usage of logic, whereby queries could be evaluated by theorem proving [Lloyd 1987]. While allowing a greater freedom in the choice of possible mappings, it is likely that in this case there would be no efficient method available for the evaluation of queries. Another disadvantage is that general logic is less easy use and understand by the CORBA developer than the small, predefined set of constructs of the specialised mapping language. The here presented method presents a good compromise between efficiency, ease of use, and a large number of mapping possibilities.

4.5 Conclusion

A declarative mapping language has been presented in this chapter, which allows to specify mappings between tables of relational databases and IDL types. The language is intended to guide automatic query translations and the necessary data transformations for the retrieval of data. To allow for a maximal freedom in the choice of the IDL and relational schema it is necessary to rely on relational views as an intermediate mapping step. This approach of two mapping steps simplifies the mapping language, facilitates reuse and insulates the CORBA server from changes in the relational database. The mapping language represents a good compromise with respect to efficiency, ease-of-use, and generality in comparison with other approaches such as hand-coding or usage of logic as a mapping specification.

Chapter 5

IDLVIEWS – A CORBA WRAPPER GENERATOR

5.1 Introduction

In the previous chapter a mapping language has been presented. IDLViews is a system which can use this language to generate CORBA wrappers for relational databases. It thereby serves as a proof of concept for the mapping language and allows to discuss the issues of chapter three from a practical point of view.

5.2 Architecture

Figure 19 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 the mapping 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.

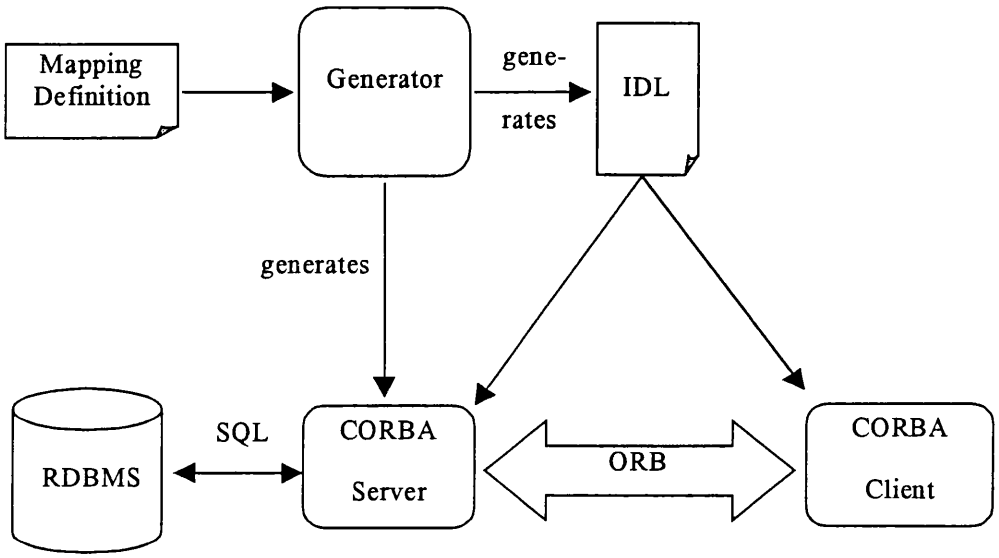


Figure 19: Architecture of IDLViews

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.

5.3 Interfaces

The IDL implemented by the CORBA server consists of two parts. The first part represents the data needed by the application. The second part specifies the API for the database itself and defines methods for querying and data retrieval. The following IDL represents the data of an application and is used in this chapter as an example.

```
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 second part of the IDL specifies methods for the querying and retrieval of the data. The basic idea of this API is similar to examples such as JDBC or the CORBA Query Service. A query evaluator allows to specify a query. The evaluator returns an iterator, which then gives access to the individual result elements. However, unlike in JDBC and the Query Service, the iterator is specific for a certain type. Even though this requires the definition of an own iterator interface for each possible return type, the advantage is that the usage of the generic type any is avoided. Anys are less efficient for the data transfer and require the conversion of the data on the client side to the actual type. If a view for the maps in the data IDL above has been defined, the following additional server 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
```

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 an object reference to an iterator. 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 optimise the data retrieval. 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.

5.4 Queries

The approach chosen for the *IDLViews* system 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.

5.4.1 The Query Language

We introduce the language informally using some examples, the grammar for this language can be found in the appendix. 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.

5.4.2 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"

No condition has to be specified.

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)))
```

5.4.3 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.

5.5 Implementation

The generator has been implemented in the programming language Java on a Solaris UNIX system. The generated server accesses an Oracle relational database using a JDBC driver. The used object request broker is Omnibroker. At the time of the implementation, Omnibroker did

not support the Portable Object Adapter (POA) and offered no other possibility to implicitly register CORBA objects. Therefore each object is registered individually when accessed the first time. The object is then loaded completely into the memory of the server and stays there until it is actively removed. This approach works fine as long as there are only few CORBA objects but clearly a different method would be necessary if a more fine grained approach is used.

5.6 Related Work

The author is not aware of any other project that follows the presented method 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. [Papazoglou 96], [Tari 97], [Wiederhold 86]). The mapping step, consequently called "semantic enrichment" in [Hohenstein 96], 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 [Fahl 97] and [Qian 95]. The approach of [Fahl 97] is similar to the one in this thesis in that they also assume that each (object-oriented) class is represented by exactly one relational table. However, the query language presented here is only a subset of theirs, as it does not treat path expressions. [Qian 95] 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 the mapping

language presented here, no relationships between extents of interfaces that are in a specialisation relationship are required or guaranteed.

Another related research area is the integration of database systems in a CORBA framework. [Leser 98a] 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" [OMG 89]. However, as detailed in [Leser 98a] and [Wells 94], this specification has severe pitfalls. 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 by the OPM project [Chen 98].

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

5.7 Discussion

A method has been presented for the semiautomatic 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 the query language can only express a limited set of queries, it proved to be sufficient for most applications.

Using a set of mapping rules, the system generates Java source code for the server. This compilation strategy has been the chosen 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 customisations. 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.

As detailed in chapter four, 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 example is the usage of inheritance. If an interface A specialises an interface B then a query against B does not necessarily return a superset of the same query against A. Such a behaviour 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 one 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.

5.8 Conclusion

The wrapper generator *IDLViews* has been presented in this chapter, which implements the mapping language in chapter four, thereby serving as a proof of concept. The *IDLViews* system is well suited to support the implementation of applications such as the standard map IDL in chapter two. It allows for the quick deployment of CORBA wrappers, which can support simple specialised client programs.

Chapter 6

CONCLUSIONS

The emergence of the CORBA standard has created many new opportunities for the combination of heterogeneous and distributed components in the field of Bioinformatics. This thesis has focused on one of the most important aspects in such an environment - the representation and distribution of molecular biology data. The main advantages of CORBA are the hiding of implementation details and the ability to access remote information like local programming language objects. The result is a simplified development of clients for CORBA wrapped data sources, such as visualisation and analysis tools.

Wrappers implement CORBA interfaces for already existing traditional data sources such as flat-files or relational databases. The most important question the developer of a wrapper faces is how to represent the data in CORBA IDL. The decision whether to represent data by value types or by CORBA objects has a major impact on the performance of the distributed system. The developer also has to choose between generic representations and domain specific representations. For both questions, the better choice depends very much on the concrete application. This text, concentrates on domain specific representations, because they are easier to use by the client and because they are more interesting in the context of this thesis.

The main problem of CORBA wrappers is the considerable effort necessary for their development and deployment. Coding the wrapper by hand, the most commonly used approach, is tedious and leads to many maintenance problems. In contrast, the automatic generation of CORBA wrappers, based on the schema of the underlying data source, is almost effortless. However, for many applications such a server is not useful, either because the performance is not sufficient or because an external standard imposes a different IDL. This thesis suggests a third option, which tries to combine the advantages of the previous two. The developer specifies a set of rules, which define the mapping between IDL constructs and schema of the data source. Once this mapping has been defined, the CORBA wrapper can be automatically generated, removing most of the burden of the server development. Additionally, such a CORBA server can support a query language, which is based on the IDL definitions. In this thesis such a mapping language has been presented for the important case of relational databases and the language has been implemented by the wrapper generator *IDLViews*. *IDLViews* is well suited for the devlopment of thin clients, because of the large degreee of freedom the developer has when choosing an IDL for his application.

Even though IDL is often used to model data, it is an API and not a data modelling language. To ensure sufficient performance, application specific access paths have to be encoded in the IDL. IDL definitions are therefore often redundant and more complex, and more difficult to understand than a data model. A similar point has been made about object-oriented databases in [Goodman 95]. In this paper the authors claim that C++ is a poor choice for modelling data, because the code tends to be application specific and complex.

The method of semiautomatic server generation is very general and can be used in many different contexts. It could just as well be used for the generation of DCOM servers or for the

generation of Enterprise Java Beans (EJBs). Different data sources such as flat-files or object-oriented databases could be used instead of relational databases. However the implementation would be more complex in these cases and it would be more difficult to support ad-hoc queries.

Standardisation of interfaces for common applications facilitates interaction of components developed by independent research groups. However, agreement on standards is a painful and difficult process – not easier in the case of IDL than in the case of flat-file formats. It might be even more difficult for CORBA interfaces because there are more possibilities to represent data and access methods using IDL and the aspect of distribution adds a new dimension to the problem. On the other hand it is relatively easy to support several different IDLs or to extend an existing one for a specialised purpose without breaking existing code.

The usage of CORBA as a middle-ware is more complex than programming within a single distributed language such as Java. However, when language and platform independence are required, there are currently no viable alternatives to CORBA. This is almost certainly true for the large public data providers such as the EBI, which would have difficulties to impose a specific programming environment on its users.

In the past four years the awareness of the CORBA standard and its possibilities has grown in the Bioinformatics community. Its advantages in an inherently distributed and heterogeneous environment have been recognised and there are an increasing number of applications available. The involvement of the EBI and the work, which has led to this thesis, have contributed to this development.

BIBLIOGRAPHY

[Atkinson 89]
M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. "The Object-Oriented Database System Manifesto", *First International Conference on Deductive and Object-Oriented Databases*, 1989, pp. 40-57.

[Altschul 90]
S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: "Basic Local Alignment Search Tool", *Journal of Molecular Biology*, **215**, 1990, pp. 403-410.

[Bairoch 99]
A. Bairoch, R. Apweiler: "The SWISS-PROT protein sequence data bank and its supplement TrEMBL in 1999", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 119-122.

[Baker 98]
P. G. Baker, A. Brass, S. Bechhofer, et al.: "TAMBIS – Transparent Access to Multiple Bioinformatics Information Sources", *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB-98)*, AAAI Press, 1995, pp. 25-34.

[Barillot 99a]
E. Barillot, U. Leser, P. Lijzaad, et al: "A Proposal for a Standard CORBA Interface for Genome Maps", *Bioinformatics*, **15**(2), Oxford University Press, 1999, pp. 157-169.

[Barillot 99b]
E. Barillot, S. Pook, et al.: "The HuGeMap Database: Interconnection and Visualization of Human Genome Maps", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 119-122.

[Barker 99]
W. C. Barker, J. Garavelli, et al.: *The PIR-International Protein Sequence Database*. Nucleic Acids Research, **27**(1), Oxford University Press, 1999, pp. 39-43.

[Bergeman 95]
E. R. Bergeman, M. Graves, C. B. Lawrence: *Viewing Genome Data as Objects for Application Development*. Proceedings of the 5th International Conference on

Intelligent Systems for Molecular Biology (ISMB-95), AAAI Press, 1995, pp. 48-56.

[Benson 99]
D. A. Benson, M. S. Boguski, D. J. Lipman, et al.: "GenBank", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 12-17.

[Berners-Lee 94]
T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret: "The World Wide Web", *Communications of the ACM*, **37**(8), 1994, pp. 76-82.

[Bishop 98]
M. J. Bishop, ed.: *Guide to Human Genome Computing*, second edition, Academic Press, 1998.

[Blake 99]
J. Blake, J. E. Richardson, et al.: "The Mouse Genome Database (MGD): genetic and genomic information about the laboratory mouse", *Nucleic Acids Research*, **27**(1), Oxford University Press , 1999, pp. 95-98.

[Bloomer 92]
J. Bloomer: *Power Programming with RPC*, O'Reilly & Associates, 1992.

[Cattell 94]
R. G. G. Cattell: *Object Data Management: Object-Oriented and Extended Relational Database Systems*, rev. ed., Addison-Wesley Publishing Company, 1994.

[Cattell 97]
R. G. G. Cattell, et al.: *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[Ceri 84]
S. Ceri and G. Pelagatti: *Distributed Databases: Principles and Systems*, McGraw Hill, 1984.

[Chen 95]
I. A. Chen and V. M. Markowitz: "An Overview of the Object-Protocol Model (OPM) and the OPM Data Management Tools", *Information Systems*, **20**(5), 1995, pp. 393-418.

[Chen 98]

I. A. Chen, A. S. Kosky, V. M. Markowitz, et al: "Advanced Query Mechanisms for Biological Databases", *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB-98)*, AAAI Press, 1998, pp. 43-51.

[Coupaye 99]
T. Coupaye: "Wrapping SRS with CORBA: from Textual Data to Distributed Objects", *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (ISMB-98)*, AAAI Press, 1998, pp. 43-51.

[Crabtree 99]
J. Crabtree, S. Fischer, M. Gibson, G. C. Overton: "Biowidgets: Reusable Vizualisation Components for Bioinformatics", *Bioinformatics: Databases and Systems*, S. Letovsky (ed), Kluwer Academic Publishers, 1999, pp. 255-263.

[Darwen 95]
H. Darwen and C. J. Date: "The Third Manifesto", *SIGMOD Record*, **24**(1), 1995, pp. 39-49.

[Davidson 99]
S. B. Davidson, O. P. Bunemann, J. Crabtree, V. Tannen, G. C. Overton, and L. Wong: "BioKleisli: Integrating Biomedical Data and Analysis Packages", *Bioinformatics: Databases and Systems*, S. Letovski (ed), Kluwer Academic Publishers, 1999, pp. 245-254.

[Date 86]
C. J. Date: "Updating Views", *Relational Databases: Selected Writings*, Addison-Wesley, 1986.

[Date 95]
C. J. Date: *An Introduction to Database Systems*, 6th edition, Addison-Wesley, 1995.

[Dib 96]
C. Dib, et al: (1996) "A comprehensive genetic map of the human genome based on 5,264 microsatellites", *Nature*, **380**, 1996, pp. 152-154.

[Dogac 96]
A. Dogac, C. Dengi, et al: "A Multidatabase System Implementation on CORBA", *6th Int. Workshop on Research Issues in Data Engineering: Nontraditional Database Systems*, New Orleans, Louisiana, 1996.

[Durbin 94]
R. Durbin, and J. Thierry-Mieg: "The ACEDB GenomeDatabase", *Computational Methods in Genome Research*, ed. S. Suhai, Plenum Press, 1994, pp. 45-55.

[Etzold 96]
T. Etzold, A. Ulyanov, and P. Argos: "SRS: Information Retrieval System for Molecular Biology Data Banks", *Methods in Enzymology*, **266**, Academic Press, 1996, pp. 114-128.

[Fahl 97]
G. Fahl, and T. Risch: "Query Processing over Object Views of Relational Data", *The VLDB Journal*, **6**(4), Springer-Verlag, 1997, 261-281.

[Fischer 99]
S. Fischer, J. Crabtree, B. Brunk, M. Gibson, and G.C. Overton: "bioWidgets: data interaction components for genomics", *Bioinformatics*, **15**(10), Oxford University Press, 1999, 837-846.

[FlyBase 99]
The FlyBase Consortium: *The Flybase Database of the Drosophila Genome Projects and community literature.* Nucleic Acids Research, **27**(1), Oxford University Press, 1999, pp. 185-88.

[Goodman 95]
N. Goodman: "An Object-Oriented DBMS War Story: Developing a Genome Mapping Database in C++", *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim (ed), ACM Press, 1995, pp. 217-237.

[Goodman 95b]
N. Goodman, S. Rozen, L. Stein: "The Case for Componentry in Genome Information Systems",
http://www-genome.wi.mit.edu/ informatics/componentry.html, 1995.

[Harold 97]
E. R. Harold: *JAVA Network Programming*, O'Reilly, 1997.

[Hohenstein 96]
Hohenstein U.: "Using Semantic Enrichment to Achieve Interoperability of Relational and ODMG Databases", *International Hong Kong Computer Society Database Workshop*, 1996, pp. 210-232.

[Hu 98]

J. Hu, C. Mungall, D. Nicholson, A. L. Archibald: "Design and Implementation of a CORBA-based Genome Mapping System", *Bioinformatics*, **14**, 1998, pp. 112-120.

[Hudson 95]
T. Hudson, et al: "An STS-Based Map of the Human Genome", *Science*, **270**, 1995, 1945-1954.

[Jacobson 92]
I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard: *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

[Jungfer 98]
K. Jungfer, and P. Rodriguez-Tomé: "Mapplet: A CORBA-based Genome Map Viewer", *Bioinformatics*, **14**(8), Oxford University Press, 1998, pp. 734-738.

[Jungfer 99a]
K. Jungfer, U. Leser, and P. Rodriguez-Tomé: "Constructing IDL Views on Relational Databases", *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, Springer Verlag, 1999, pp. 255-268.

[Jungfer 99b]
K. Jungfer, G. Cameron, and T. Flores: "CORBA and the EBI Databases", *Bioinformatics: Databases and Systems*, S. Letovski (ed), Kluwer Academic Publishers, 1999, pp. 245-254.

[Karp 95]
P. D. Karp: *A Strategy for Database Interoperation*. Journal of Computational Biology, **2**(4), Oxford University Press, 1995, pp573-586.

[Karp 99]
P. D. Karp, M. Riley, et al.: *Encyclopedia of Escherichia coli genes and metabolism*. Nucleic Acids Research., **27**(1), Oxford University Press, 1999, pp. 55-58.

[Leser 98a]
U. Leser, S. Tai, S. Busse: "Design Issues of Database Access in a CORBA Environment", *Workshop on Integration of Heterogeneous Software Systems*, Magdeburg, Germany, 1998.

[Leser 98b]

U. Leser, R. Wagner, et al: "IXDB, an X Chromosome Integrated Database", *Nucleic Acids Research*, **26**(1). Oxford University Press, 1998, pp. 108-111.

[Letovsky 99]
S. I. Letovsky, editor: *Bioinformatics: Databases and Systems*, Kluwer Academic Publishers, 1999.

[Lloyd 87]
J. W. Lloyd: "Deductive Databases", *Foundations of Logic Programming*, second extended edition, Springer-Verlag, 1987, pp. 141-172.

[Maltchenko 98]
S. Z. Maltchenko: "The BioObjects project. Part I: The Object Data Model core elements", *Bioinformatics*, **14**(6), Oxford University Press, 1998, pp. 479-485.

[Markowitz 95]
V. M. Markowitz, and O. Ritter: "Characterizing Heterogeneous Molecular Biology Data Systems", *Journal of Computational Biology*, **2**(4), Oxford University Press, 1995, pp. 547-556.

[Markowitz 99]
V. M. Markowitz, I. A. Chen, A. S. Kosky, and E. Szento: "OPM: Object-Protocol Model Data Management Tools'97", *Bioinformatics: Databases and Systems*, S. Letovski (ed), Kluwer Academic Publishers, 1999, pp. 187-199.

[Mewes 99]
H. W. Mewes, K. Heumann, et al: "MIPS: a database for genomes and protein sequences", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 44-48.

[Ohkawa 95]
H. Ohkawa, J. Ostell, and S. Bryant: *An ASN.1 Specification for Macromolecular Structure*. Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology (ISMB-95), AAAI Press, 1995, pp. 259-267.

[OMG 98]
Object Management Group: *CORBAservices: Common Object Services Specification*, OMG publication 98-12-09, http://www.omg.org/, 1998.

[OMG 99]
Object Management Group: *The Common Object Request Broker: Architecture and Specification (Revision 2.3)*. OMG publication 98-12-01, http://www.omg.org/, 1999.

[Orfali 96]
R. Orfali, D. Harkey, J. Edwards. *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, 1996.

[Orfali 97]
R. Orfali, D. Harkey: *Client/Server Programming with JAVA and CORBA*, John Wiley & Sons, 1997.

[Papazoglou 96]
M. Papazoglou, Z. Tari, and N. Russell: "Object-Oriented Technology for Interschema and Language Mappings" *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, O.A. Bukhres, A.K. Elmagarmid (eds.), Prentice Hall, New Jersey, 1996, pp. 203-250

[Pearson 90]
W. R. Pearson: "Rapid and Sensitive Sequence Comparison with FASTP and FASTA", *Methods in Enzymology*, 183, 1990, pp. 63-98.

[Qian 95]
X. Qian, and L. Raschid: "Query Interoperation among object-oriented and relational databases", *11th Int. Conference on Data Engineering*, Tapei, Taiwan. IEEE Computer Soc. Press, 1995.

[Rodriguez 97]
P. Rodriguez-Tomé, C. Helgesen, P. Lijnzaad, and K. Jungfer: "A CORBA server for the Radiation Hybrid Database", *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology (ISMB-97)*, AAAI Press, 1997, pp. 250-253.

[Rodriguez 99]
P. Rodriguez-Tomé, and P. Lijnzaad: "The Radiation Hybrid Database", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 115-118.

[Schuler 96a]
G. D. Schuler, et al: "A Gene Map of the Human Genome", *Science*, **274**, 1996, pp. 540-546.

[Schuler 96b]
G. D. Schuler, J. A. Epstein, et al: "Entrez: Molecular Biology Databases and Retrieval System", *Methods in Enzymology*, **266**, Academic Press, 1996, pp. 141-162

[Searls 95]
D.B. Searls: "bioTk: Componentry for Genome Informatics Graphical User Interfaces", *Gene*, **163**(2), pp. GC1-16, 1995.

[Sessions 96]
R. Sessions: *Object Persistence: Beyond Object-Oriented Databases*, Prentice Hall, 1996, pp. 235-239.

[Sessions 98]
R. Sessions: *COM and DCOM: Microsoft's Vision for Distributed Objects*, John Wiley & Sons, 1998.

[Siegel 96]
J. Siegel.: *CORBA Fundamentals and Programming*, New York, John Wiley & Sons, 1996.

[Skupski 99]
M. P. Skupski, M. Booker, A. Farmer, et al.: *The Genome Sequence DataBase; towards an integrated functional genomics resource.* Nucleic Acids Research, **27**(1), Oxford University Press, 1999, pp. 35-38.

[Stoesser 99]
G. Stoesser, M. A. Tuli, R. Lopez, and P. Sterk: "The EMBL Nucleotide Sequence Datbase", *Nucleic Acids Research*, **27**(1), Oxford University Press, 1999, pp. 35-38.

[Stonebraker 90]
M. Stonebraker et al.: *Third Generation Database System Manifesto.* ACM SIGMOD Record **19**(3), 1990.

[Tari 97]
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, pp. 465-475.

[Wells 94]

D. L. Wells, and C. W. Thompson: "Evaluation of the Object Query Service Submissions to the Object Management Group", *IEEE Quarterly Bulletin on Data Engineering*, **17**(4), 1994, pp. 36-45.

[Wiederhold 86]
G. Wiederhold: "Views, Objects and Databases", *IEEE Computer*, **19**(12), 1986, pp. 37-44.

[Wiederhold 92]
G. Wiederhold: "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, 1992, pp. 38-49.

APPENDIX A

The Common IDL

```
module Traders {

  interface Trader {
     exception NotFound {};
     Object getByOid(in string oid)
       raises (NotFound);
  };
};

module Maps {

   typedef sequence <string> Strings;

   struct CrossReference {
     string   database;
     string   accession;
   };
   typedef sequence < CrossReference > CrossReferenceList;

   struct MarkerData {
     string oid;
     string species;              // of marker.
                                  // Can place e.g. mouse marker
                                  // on a human map.
     string chromosomeSegment;    // Position at chromosome:
                                  // chr + arm + band
     CrossReferenceList crossReferences;
```

```
    };
    typedef sequence < MarkerData >  MarkerList;

    struct MapElement {
      string     oid;             // unique object id.
      MarkerData markerData;      // Associated marker
      float      position;        // distance from top of map
      boolean    frameworkMarker; // Is a framework marker?
    };
    typedef sequence < MapElement > MapElementList;

    interface Map {
      readonly attribute string   oid;
      readonly attribute string   name;
      readonly attribute string   type;
      readonly attribute string   species;
      readonly attribute string   chromosome;
      readonly attribute float    length;
      readonly attribute string   unit;
      readonly attribute long     size;
      readonly attribute MapElementList  elements;
    };
    typedef sequence < Map > MapList;

    interface MapTrader : Traders::Trader {
      MapList getMapList( in string mapType,
                          in string species,
                          in string chromosome,
                          in Strings markers);
          raises (Traders::Trader::NotFound);
    };
};
```

APPENDIX B

Grammar of the IDLViews Query Language

```
<where_clause> ::= <condition>*

<condition> ::=   <base_spec>
                | <slot_spec>
                | <sequence_spec>
                | <compound_cond>

<compound_cond> ::=   (&& <condition>+)
                    | (|| <condition>+)
                    | (! <condition>)

<sequence_spec> ::=   <exists>
                    | <all>
<exists>   ::= (exists <where_clause>)
<all>      ::= (all <where>_clause)

<slot_spec>   ::= (<slot_name> <where_clause>)

<base_spec>   ::= <base_value> | <range_spec>

<range_spec> ::=   <gt_spec>
                 | <ge_spec>
                 | <lt_spec>
                 | <le_spec>

<gt_spec>    ::= (>  <base_value>)
<ge_spec>    ::= (>= <base_value>)
```

```
<lt_spec>     ::= (<  <base_value>)
<le_spec>     ::= (<= <base_value>)

<base_value> :==   <string>
                 | <int>
                 | <float>
                 | <boolean>

<boolean> ::= true | false
```