A CORBA-BASED MEDIATION SYSTEM FOR THE INTEGRATION OF WRAPPED MOLECULAR BIOLOGY DATA SOURCES

Anastassia Spiridou

Ph.D. Thesis
University College London
2002

Abstract

Integration of data from disparate, heterogeneous and autonomous data sources is a common problem encountered in different domains, including the domain of Molecular Biology. Mediator-based architectures have been developed to deal with integration of information from heterogeneous and autonomous data sources, and views have been used to restructure data representation.

CORBA can resolve some of the problems involved in data integration by providing programming language, platform and network transparency. In CORBA, it is advantageous to model data itself in IDL, essentially creating IDL schemas. Integration of data served by different CORBA servers and modelled in IDL requires resolving schematic heterogeneity between the different IDL schemas. That involves mapping from one or more source IDL schemas to a preferred target IDL schema. Manual implementation of the mapping is possible but tedious.

The system described in this thesis offers creation of customised representations of data and data integration on CORBA-wrapped data sources. Views are employed to restructure data representation. The system supports semi-automatic generation of target CORBA servers based on the specification of source to target IDL mapping in a specially developed language. The mapping language has a high-level notation for expressing mappings easily and concisely, as well as procedural features to support complex cases. The mediation system is applied to the integration of bacterial genome data from two independently developed CORBA wrapped data sources.

Acknowledgements

I want to thank my supervisors Patricia Rodriguez-Tomé, Janet McDonnell, and Janet Thornton for their continuous support during my work and their useful comments. I would also like to thank Kim Jungfer, Philip Lijnzaad, Jeremy Parsons and Martin Senger for our fruitful discussions, as well as Thena Rosahl for proof reading. Last, but not least, special thanks to Paulo and Helena for their invaluable support and patience.

To my father, with love.

Table Of Contents

Abstract....2

Acknowledgements....3

Chapter 1: INTRODUCTION....8

1.1 Data Integration in Molecular Biology....9

1.1.1 Data characteristics....10

1.1.2 Data storage/management....10

1.1.3 Data access....11

1.1.4 Information Systems....12

1.2 The Common Object Request Broker Architecture (CORBA)....16

1.2.1 The Object Management Architecture (OMA)....17

1.2.2 The Interface Definition Language (IDL)....19

1.2.3 Important ORB features....21

1.2.4 CORBA 3-tier architecture for data access....21

1.3 Mediators....23

1.4 CORBA in Molecular Biology....25

1.5 A CORBA-based Mediation System....26

1.6 Outline of Thesis....29

Chapter 2: CORBA MEDIATORS 30
2.1 Architecture 30
2.2 Common model and schema definition language 32
2.3 Target object derivation 39
2.4 Publishing initial CORBA objects 41
2.5 Wrapper query support 42
2.6 Mediator query support 46
2.7 Conclusions 47
Chapter 3: THE MAPPING LANGUAGE 48
3.1 General requirements 48
3.2 Design 49
3.2.1 Common model and schema definition language 50
3.2.2 Target object derivation 52
3.2.3 Publishing initial CORBA objects 52
3.2.4 Wrapper query support 53
3.2.5 Mediator query support 54
3.3 Description 54
3.3.1 Include directive 56
3.3.2 Server declaration 56
3.3.3 Module 58
3.3.4 View 58
3.3.5 Location 61
3.3.6 Definition 61

3.3.7 Source references 63

3.3.8 Mapping 63

3.3.9 User methods 69

3.3.10 Names and Scope 69

3.4 Conclusions 73

Chapter 4: THE MEDIATOR SYSTEM 75

4.1 Architecture 75

4.2 The IDL View Generator 77

4.2.1 Use of the IDL View Generator 77

4.2.2 Mediator generated files 78

4.2.3 Implementation 79

4.3 Mediators 79

4.3.1 Object composition and the adapter design pattern 79

4.3.2 Interface implementation 82

4.3.3 Inheritance implementation 85

4.4 Conclusions 88

Chapter 5: APPLICATION IN BACTERIAL GENOMES 89

5.1 Source CORBA Servers 90

5.1.1 SubtiList 90

5.1.2 EMBL 94

5.2 Object Model 96

5.3 Mediator CORBA Server IDL 100

5.4 View Definition 102

5.5 A Client Application 116

5.6 Conclusions 121

Chapter 6: CONCLUSIONS 123

Appendices 130

Appendix A: The mapping language specification in BNF 131

Appendix B: The SubtiList CORBA server IDL 135

Appendix C: The bacterial genomes objects model 149

Appendix D: Generated mediator code 160

Appendix E: Client application code 176

References 180

Chapter 1: INTRODUCTION

The various genome projects have produced a large amount of data, which is growing rapidly. Integrating selected interrelated genome data can be advantageous in many cases enabling the discovery of new relationships between data and analysis of the integrated data. Genome-related data is inherently complex, highly diverse, with a structure that is continuously modified as biological theories evolve. It is stored in different formats, managed by a variety of systems distributed around the world, and accessed through different methods. Integration of data from distributed, heterogeneous and autonomous data sources is a challenging problem, commonly encountered in many domains. In every case, the heterogeneity present at the different levels needs to be resolved, in order to achieve the goal of data integration.

A number of systems have been developed to integrate data from different molecular biology data sources. They try to resolve data sources heterogeneity at all levels, i.e. programming language, hardware platform, operating system, and data representation. At the same time, they try to offer different ways of accessing and utilising the data served, including a range of visualisation and analysis tools, as well as supporting access over the Internet. This is a large task that often leads to duplication of programming effort by different systems and to proprietary solutions, unless available technologies that promote standardisation are used.

In this thesis, an approach is examined that attempts to solve some of the problems involved in molecular biology data integration using the Common Object Request

Broker Architecture (CORBA) within a mediation system. CORBA, which is based on standardisation, handles heterogeneity at the programming language and platform levels, and provides network transparency. This allows CORBA-based data integration approaches to focus on resolving heterogeneity at the level of data representation. Mediator-based architectures have been used to integrate information from autonomous and heterogeneous data sources. They resolve heterogeneity at the level of data representation. The approach examined in this thesis is a mediation system that utilises CORBA. It supports the integration of data from molecular biology data sources that provide access through CORBA. It also supports the creation of customised CORBA views of such data sources. This is achieved by resolving the schematic heterogeneity involved.

The remainder of the introduction is organised as follows. First, the data integration problem in the domain of molecular biology is analysed, and the most influential and currently available systems are surveyed. Then, an overview of the most important, and relevant to this thesis, elements of CORBA is provided, with a focus on data access in the framework of CORBA. Mediator architectures and their components are then introduced, together with the use of views within a mediation system. The different initiatives of utilising CORBA in the domain of Molecular Biology are outlined. The approach of this thesis is, then, introduced and compared to related work. The chapter concludes with an outline of the rest of the thesis.

1.1 Data Integration in Molecular Biology

Integrating interrelated biological data of interest opens up new possibilities such as discovering new relationships between data, and analysing the integrated data. In order to achieve data integration, the heterogeneity present at the different levels needs to be resolved.

The data integration problem in the domain of molecular biology is analysed here by examining the characteristics of the domain data, the different options for data storage and management, and the different methods for data access used. Some of the most important molecular biology information systems currently used for data integration as well as providers of the integrated data are also surveyed.

1.1.1 Data characteristics

Genome projects and various research groups have produced a large amount of data, which has been growing rapidly. Genome-related data is highly complex for a number of reasons such as, the high degree of interrelationships, the changes in the type/form of data based on different conditions, and the uncertainty inherent in science. It is also very diverse including data such as, nucleotide sequences, genomic objects, protein sequences, metabolic pathways, protein structures, genome maps, taxonomies, and bibliographic references. Adding to the complexity is the fact that biology as a science is continuously being augmented and altered. As a result, the structure of the data changes frequently in order to consistently represent the state of biological knowledge over time.

1.1.2 Data storage/management

Molecular biology data is stored in different formats and managed by a variety of systems. Formatted text files, called flat files, have been widely used for the storage and exchange of molecular biology data because no database management system is required and the data is in a human-readable form. Though it appears to be simple, this approach has certain disadvantages. First, a parser has to be written for each flat file format, which may, in addition, not be fully specified and change frequently. Second, the redundancy in data storage can be quite high. For example, multiple copies of the same data may need to be stored one for each required format, since different programs use different flat file formats. The same piece of information may also be stored multiple times one for each flat file entry, resulting in redundant flat files. Besides flat files, database management systems have been used for the storage and management of molecular biology data. These include relational and object-oriented commercial systems, as well as others specifically developed for a given project (for instance, the Acedb [Durbin 94]).

As a consequence of using different formats and systems for the storage and management of molecular biology data, a number of different data models are being used, such as the relational, the object-oriented, and the data model of a flat file format when this is defined. This creates data model heterogeneity (also called *metamodel heterogeneity* [OMG 01c], [Frankel 99]) between the different systems. Furthermore, and because different research groups have different ideas on how to best represent biological objects, many different conceptual models of the domain data (also called schemas) are used, resulting in *schematic heterogeneity*. Examples of schematic heterogeneity are:

- synonyms (different terms referring to the same entity),
- homonyms (the same term referring to different entities), and
- the same entity being modelled at different levels of detail within different schemas.

Schematic heterogeneity for the relational model is analysed in Kim et al. [Kim 91].

1.1.3 Data access

Molecular biology data is accessed through a variety of methods. The World Wide Web has become the most popular access method (offered by systems such as, the SRS [Etzold 96], OPM [Kosky 98], Entrez [Schuler 96], and ACEDB [Durbin 94]) due to its ease-of-use, its support by a wide range of hardware and software systems, as well as the development of powerful Web-based graphical user interfaces using JAVA and related tools. It offers mainly two ways for data exploration. The first is navigation using hypertext links. Although this method is very well suited for browsing molecular biology data due to the high degree of interrelationships present in the data, it is limited to exploring only a small number of data. The second method is using Web queries, which usually take the form of keyword combinations. Keyword combinations can express quite complex queries. However, this method is limited to searching according to the criteria provided by the designer of the Web interface only. In many cases, it is required to search according to different criteria, or to apply one's own visualisation and/or analysis tools to the data. In those cases, utilising Web data becomes cumbersome since it involves retrieval and parsing.

Besides Web interfaces, many systems provide access to their data through Application Programming Interfaces (APIs) for one or more programming languages (for example, the SRS [Etzold 96], and ACEDB [Durbin 94]). The advantage of this method is that once data is retrieved in data structures of the programming language, it can be manipulated with great flexibility. However, in comparison to Web interfaces, this requires a lot more involved work. Application developers have to know and use the programming language of the API to retrieve data, even if they may need to use different programming languages for visualisation and/or analysis of the retrieved data.

In contrast to APIs, which are tied to a specific programming language, a CORBA interface is programming language independent, and still offers the advantage of flexible data manipulation. CORBA, which is designed for development in distributed heterogeneous environments, also offers platform independence and network transparency. Due to the useful features of CORBA, which are examined in more detail in Section 1.2, many systems provide access to their data through CORBA (for example, the SRS [Coupaye 99], and ACEDB [Durbin 94]).

1.1.4 Information Systems

In molecular biology, a number of information systems have been developed to integrate data from heterogeneous data sources. In some cases, the developed systems act as providers of the integrated data, as well. A few of the most important of these systems are examined here by looking at their data management features and integration approach, whether they support user views, and the data access methods offered:

- SRS

The Sequence Retrieval System (SRS) [Etzold 96] integrates data from a large number of molecular biology data sources ranging from major databases, such as the EMBL [Stoesser 02] and SwissProt [Bairoch 00], to small and specialised ones. It takes the formatted text files distributed by data sources as input and, using its powerful parser, it builds indices on data fields and on links between data sources for efficient retrieval and navigation purposes. It provides its own query language and a number of different interfaces, i.e. a Web interface, a C language API, a UNIX command line interface, as well as a CORBA interface. The CORBA interface [Coupaye 99] provides SRS Object Servers, which are CORBA wrappers for the data sources managed by SRS. SRS Object Servers can be generated based on view definitions, thus providing flexible access to the underlying data sources. Queries on the created CORBA objects are also supported, though they can be expressed in the proprietary SRS Query Language only. Support for creation of user views has also been added to the SRS system including features such as the selection of the data fields to be displayed, and the definition of virtual entries that may contain information from many data sources [Etzold 97].

- BioKleisli

BioKleisli [Davidson 96] is a system that integrates heterogeneous data sources and application programs using the *Collection Programming Language* (CPL) for querying and transforming the data. BioKleisli does not support schemas. It provides drivers to Sybase, ASN.1, OPM, and ACEDB databases, as well as to the Blast and FASTA sequence analysis packages. It supports complex data types such as, *records*, *variants*, and arbitrarily nested *sets*, *bags* and *lists*. It also supports the specification of database transformations and constraints using a declarative language, called the *Well-founded Object Logic* (WOL), a prototype implementation of which is provided within the Morphase system [Davidson 97].

- OPM

The OPM data management tools have been developed for the exploration of heterogeneous databases [Kosky 98]. They use the Object Protocol Model (OPM) as the common data model [Chen 95]. This is an object-oriented data model with added support for modelling scientific experiments (i.e. using the protocol construct). The resulting system supports the assembly of heterogeneous databases into a multidatabase system using tools for translating OPM schemas into relational schemas, and, in the opposite direction, for creating OPM views for existing relational databases and flat files. OPM views generated automatically on top of existing data sources can further be refined using a number of schema restructuring operations [Chen 97]. The system also includes facilities for browsing metadata, and support for multidatabase queries using either a Web query interface or expressed textually in the OPM multidatabase query language. A CORBA wrapper for OPM systems was planned [Kosky 96], but has not been developed. The OPM tools have been employed for the construction of prototype multidatabase systems that involve heterogeneous molecular biology databases such as the Genome Database (GDB), the Genome Sequence Database (GSDB) and GenBank [Kosky 98].

- Entrez

Entrez [Schuler 96] is a retrieval system that integrates information from a number of molecular biology and biomedical databases at the National Center for Biotechnology Information (NCBI). These databases include nucleotide sequences (e.g. GenBank), protein sequences (e.g. SwissProt, translated GenBank sequences), 3-D macromolecular structures (e.g. MMDB), complete genome assemblies, population study data sets, taxonomic data, and biomedical literature (i.e. PubMed, OMIM). The system maintains links between these databases, as well as calculating and storing 'neighbours' (i.e. links within a database) for biological sequences, 3-D structures and scientific articles. The Abstract Syntax Notation (ASN.1) is used as the common data model for the integration of the databases, as well as the language in which data is expressed. Compared to flat files, the ASN.1 is less suitable to be read by humans. The system provides a Web interface, and a C language API with network modules that allows accessing Entrez via the Internet.

- ACEDB

Acedb [Durbin 94] is a genome database system that was originally developed for the C.elegans genome project (A C. elegans DataBase). It provides its own object-oriented database management system with a non-standard data model and many graphical displays and tools specifically developed for genomic data. Besides graphical browsing, it also provides its own query language. It uses a custom, human readable text file format to express/exchange data. It supports a number of schema refinement operations. It provides a Web interface, and access through Perl and Java APIs. It also provides a CORBA interface that supports accessing Acedb data through generic CORBA interfaces and submitting queries in the Acedb query language. However, the CORBA interface provides very limited support for accessing metadata (i.e. only class names can be retrieved). Thus, metadata, which are necessary for the utilisation of generic CORBA interfaces, have to be obtained separately, in a way that is not specified. Acedb has been used as the data management system of many different genomic databases from bacteria to human (IGD [Ritter 94]).

- EcoCyc

EcoCyc [Karp 96] is a knowledge base system that integrates metabolic and genomic data. It is based on the frame data model, which has certain similarities to an object-oriented data model, and, in addition, has features that facilitate schema evolution. The system provides a number of graphical displays specifically designed for metabolic and genomic data. It supports the KIF declarative language, as well as providing a number of built-in queries via menus.

The above information systems follow different approaches to solving the data integration problem in molecular biology. Each one of them has developed its own solution for accessing the data from data sources and integrating it. This involved resolving data sources heterogeneity at all levels, i.e. programming language, hardware platform, operating system, and data representation. In addition, some of these systems have been employed in molecular biology data integration projects and provide regularly updated versions of the integrated data (for example, SRS [Etzold 96], Entrez [Schuler 96], and EcoCyc [Karp 96]). They usually offer different ways of accessing and utilising the data served, including a range of visualisation and analysis tools, as well as supporting access over the Internet. A result of this big task being undertaken by different groups each developing its own system is duplication of development effort in trying to resolve data sources heterogeneity. Another drawback is that using the produced integrated data outside the provided system usually requires a lot of effort.

Alternatively, well-established technologies that promote standardisation can be utilised. That would minimise duplication of required development, and provide integrated data in a form that is easier and more flexible to use. One such technology is the Common Object Request Broker Architecture (CORBA) described in the following Section.

1.2 The Common Object Request Broker Architecture (CORBA)

CORBA is the product of the Object Management Group (OMG) [OMG]. The OMG is a software consortium that was founded in 1989 with the purpose of developing and promoting standards for software development in distributed heterogeneous environments. It adopts specifications based on contributions of its members, which include a large number of software and hardware vendors as well as end-users. Software that is developed conforming to these specifications is guaranteed to work in a heterogeneous computing environment across all major hardware platforms and operating systems.

An overview of the OMG's Object Management Architecture (OMA), which describes the main elements of CORBA, is provided in Section 1.2.1. The main component of OMA is the CORBA specification ([OMG 99b], [Vinoski 97], [Siegel 96]) which includes:

- the Interface Definition Language (IDL), which is the language CORBA objects are described in (see Section 1.2.2), and
- the Object Request Broker (ORB), which is responsible for the communication between CORBA objects (see Section 1.2.3).

The use of CORBA in a 3-tier architecture setting for data access is discussed in Section 1.2.4.

1.2.1 The Object Management Architecture (OMA)

The OMA [OMG 97] is composed of an *Object Model* and a *Reference Model*. The Object Model defines how objects distributed across a heterogeneous environment can be described. The Reference Model characterises interactions between those objects.

In the OMA Object Model, an object is an identifiable, encapsulated entity whose services can be accessed through well-defined interfaces. Clients issue requests to objects to perform services on their behalf. The implementation and location of objects are hidden from clients, thus supporting programming language, platform and network transparency.

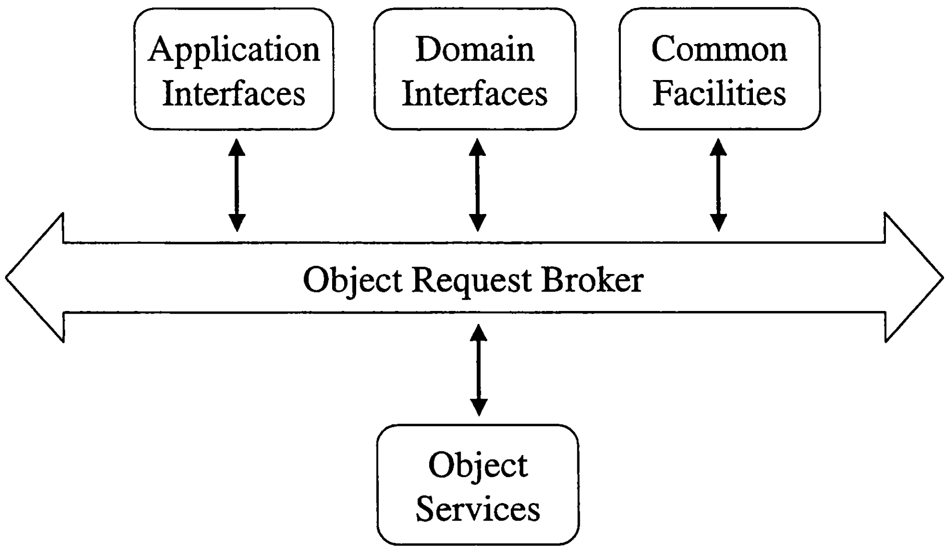

Figure 1-1: The OMA Reference Model

The OMA Reference Model identifies and characterises the components, interfaces and protocols that compose the OMA (Figure 1-1). This includes the *Object Request Broker* (ORB) that enables clients and objects to communicate in a distributed environment and four categories of object interfaces:

- *Object Services*: These are interfaces for general, domain-independent services that are likely to be used by many distributed object programs [OMG 98b]. Examples are the *Naming Service* and the *Trader Service* that provide for the discovery of objects, and the *Query Service* that provides support for queries on collections of objects.

- *Common Facilities*: These are interfaces oriented towards end-user applications, such as the document management application.
- *Domain Interfaces*: These are interfaces oriented towards specific application domains, such as the domain of *Life Sciences Research.*
- *Application Interfaces*: These are non-standardised interfaces developed specifically for a given application.

1.2.2 The Interface Definition Language (IDL)

Services provided by CORBA objects can be accessed through their public interfaces, which describe all the operations and types they support. Object interfaces are defined in the Interface Definition Language (IDL) [OMG 98a], the main elements of which are:

- *interfaces* that define the attributes (i.e. a pair of get and set methods) and operations of CORBA objects,
- *data types* used to specify the parameter types and return types for operations. The IDL supports a number of basic types such as *float*, *short*, *char*, and *boolean*, constructed types such as *struct* and *union*, and template types such as *sequence* and *string*, and
- *modules* that support scoping of definition names into a hierarchical name space to avoid name clashes.

An important feature of the IDL is that it is language independent. The IDL is used to define only the public interface of a CORBA object. The supported operations are implemented in a programming language of the developer's choice. This allows CORBA objects to be implemented using different programming languages and still communicate with each other. In order to achieve this, CORBA defines standard mappings that translate IDL definitions into constructs of all the supported programming languages, such as JAVA, C++, C and Smalltalk.

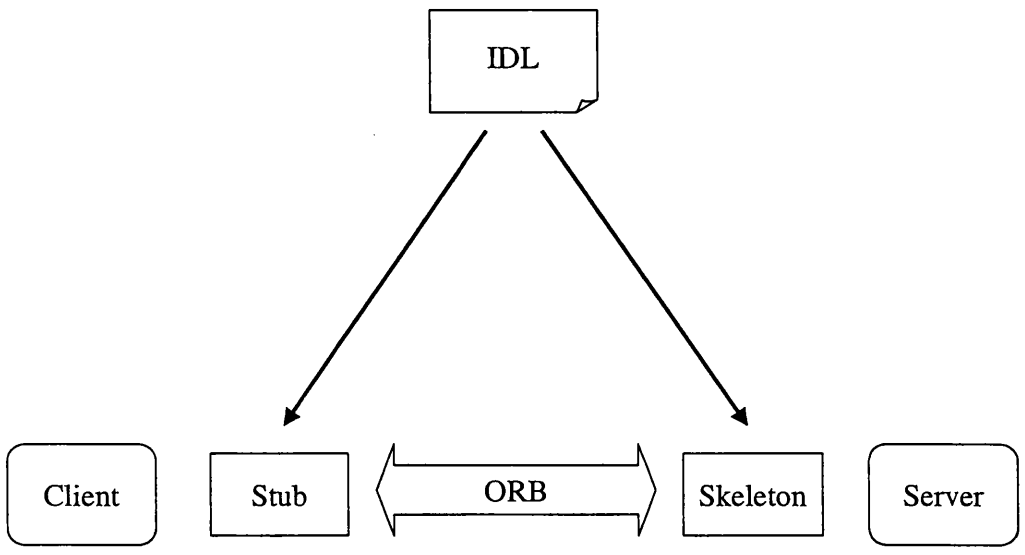

Figure 1-2: Generation of stubs and skeletons

The way object definitions in IDL are used in a CORBA setting is described below. An IDL compiler takes IDL definitions as input and generates client-side stubs and server-side skeletons (Figure 1-2). Stubs effectively issue requests on behalf of clients, while skeletons deliver requests to object implementations. Client stubs represent CORBA objects in the local programming language acting like local proxies for remote server objects. Server skeletons include declarations of CORBA objects with their methods that have to be implemented by the CORBA server developer. Stubs and skeletons are responsible for marshalling and unmarshalling, that is, converting a request between its programming language form and the form used for transmission.

1.2.3 Important ORB features

The ORB delivers client requests to objects and returns any responses. The main feature of the ORB is that it performs the communication between clients and objects transparently. That is, the client knows nothing about the object implementation (i.e. programming language, operating system and hardware), the object location, or the communication mechanism used. These features allow application developers to focus on their own application domain issues and not have to worry about low-level distributed system programming issues. They also allow CORBA-based data integration approaches to focus on resolving heterogeneity at the level of data representation.

In order to issue a request to a CORBA object, a client needs to have a "handle" on that object, called an *object reference* in CORBA terminology. An object reference is created when an object is created, and it always refers to the same object. An object reference can have a proprietary (i.e. ORB specific) format, or a standard format, called the *Interoperable Object Reference* (IOR), which is understood by different ORBs, thus allowing ORB interoperation.

1.2.4 CORBA 3-tier architecture for data access

As an extension to the traditional 2-tier client / server architecture, CORBA supports a 3-tier architecture for data access (Figure 1-3). Databases reside in the 3rd tier. In the 2nd tier, CORBA server objects are defined using the IDL. Those objects represent the specific domain and provide access to data stored in databases. A CORBA server in this middle-tier providing IDL interfaces over an underlying data source is often called CORBA *wrapper* of the data source (see Section 1.3 for more detailed information on wrappers). The 1st tier is formed by CORBA client applications that access server objects over the ORB. As a result, CORBA clients gain access to data stored in databases through the IDL definitions of CORBA server objects.

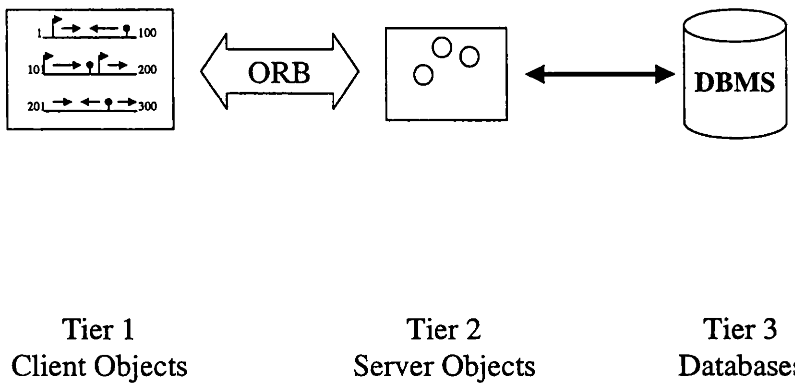

Figure 1-3: CORBA 3-tier Architecture

There are different approaches on how to define the CORBA server objects of the 2nd tier in order to best support CORBA-based data access. Some approaches define CORBA server objects that offer generic interfaces to clients for accessing the data residing in the 3rd tier ([Dogac 96], (Kemp 00]). Data is encoded in strings or bit-streams that clients need to parse. In order to guide the parsing and decoding of data, some kind of metadata is usually provided, as well. Essentially, those approaches use CORBA only for the infrastructure of the system. An advantage is that CORBA server objects are independent of data definitions. Other approaches define CORBA server objects that model the data itself in IDL, essentially creating IDL schemas in the 2nd tier. The main advantage of the latter is that clients are offered domain-specific data as opposed to, for instance, strings that they would have to parse in order to retrieve the encoded data.

If data was to be accessed from more than one data sources and integrated before being offered to clients, the 3-tier architecture of Figure 1-3 would have to be extended. Appropriate components would need to be introduced to resolve the heterogeneities involved and to integrate the data. Mediator-based architectures have been used to address those issues, and are examined in the following Section. In Section 1.5, a system is proposed that extends the 3-tier architecture of Figure 1-3 into a mediator-based CORBA architecture in order to support data integration.

1.3 Mediators

The concept of mediator was first proposed by Wiederhold [Wiederhold 92] as a way of dealing with integration of information from autonomous and heterogeneous data sources. A number of projects have adopted a mediator-based architecture including TSIMMIS [Chawathe 94], Garlic [Carey 95], DISCO [Tomasic 96] and COIN [Goh 94].

A mediator-based architecture (Figure 1-4) includes two main components: *mediators* and *wrappers.*

A *mediator* defines a common model for the representation of the information it serves to applications. It typically transforms data coming from wrappers to that common model and often to a common domain model or schema, integrates the transformed data, and provides queries over the integrated data. It translates queries received from applications into sub-queries to wrappers, and translates answers from wrappers into a form appropriate to applications.

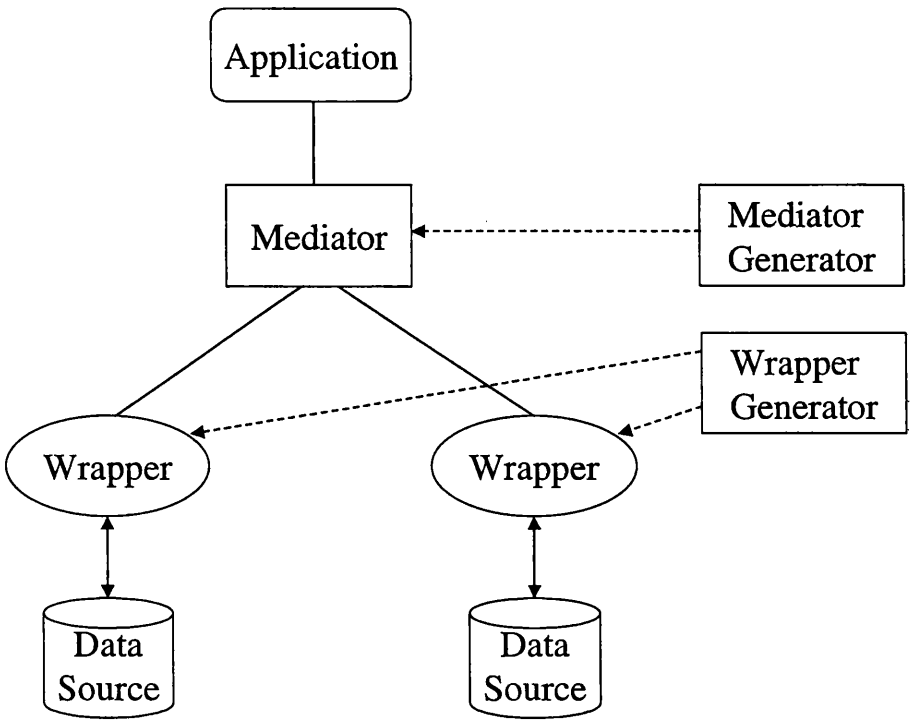

Figure 1-4: Mediator-based Architecture

A *wrapper* provides an interface to a data source. It translates sub-queries received from mediators into queries to its underlying data source, and translates answers received from its data source into a form suitable to each mediator.

Tools which facilitate the creation of mediators and wrappers (called *mediator generators* and *wrapper generators*, respectively) are also often present in a mediator-based architecture. They generate mediators or wrappers either automatically or semi-automatically from high-level descriptions of the required functionality.

Mediators often need to restructure data representation, that is, the data model and/or the schema. One way to implement the restructuring of data representation is using views. Views have been used in relational and object-oriented databases ([Abiteboul 91], [Guerrini 97], [Kuno 96], [Kim 95], [Scholl 91]) for data integration, to provide users/applications with customised representations of data, to support database schema evolution, and as shorthand for queries. More recently, views have also been used to restructure information available on the Web ([Atzeni 97], [Arocena 98], [Fernandez 97]). A mediator generator that implements mediators using views, would typically use a *view mapping language* to define the common schema as well as the mapping between existing wrapper schemas and the common schema.

1.4 CORBA in Molecular Biology

Several groups in the bioinformatics community have recognised the benefits of CORBA and have developed applications using CORBA, or have made their tools, services, and/or data available through CORBA interfaces. Examples include the EMBL database [Wang 00], SRS [Coupaye 99], ACEDB [ACEDB], RHDB [Rodriguez-Tomé 97], the HuGeMap database [Barillot 99b], SPiD [Hoebeke 01], ArkDB [Hu 98], the Virgil database [Achard 98], and JESAM [Parsons 00]. In many cases it would be advantageous to integrate data from different CORBA wrapped data sources in order to, for example, analyse or visualise the integrated data using specific software tools. Providing CORBA interfaces facilitates data integration by resolving heterogeneity at the programming language and platform levels, and providing network transparency, which drastically reduces required development work. However, if each data source provides a different CORBA interface definition, data integration still remains a complex task having to resolve schematic heterogeneity between the different CORBA wrapper schemas.

The OMG's Life Sciences Research (LSR) group [LSR] was formed in 1997. One of its main objectives is to improve interoperability among computational resources in life sciences research. This is achieved by using the OMG technology adoption process to standardise interfaces for relevant software tools, services, frameworks and components. Among others, it covers the fields of bioinformatics, genomics, genetics, structural biology, and computational molecular biology. Supporting LSR's activities and providing standardised interfaces for accessing CORBA wrapped molecular biology data sources, automatically removes schematic heterogeneity, thus making data integration a much simpler task. As an example of promoting the approach of standardised interfaces in order to overcome schematic heterogeneity between different CORBA servers, [Barillot 99a] proposes a consensus/standard IDL definition for genome maps. This approach utilises CORBA in its full potential for achieving interoperability. However, in those cases that standardisation cannot be achieved, or for definitions outside the scope of standardisation, schematic heterogeneity remains an issue.

1.5 A CORBA-based Mediation System

In this thesis, an approach is examined that attempts to solve some of the problems involved in data integration using CORBA within a mediator-based architecture. CORBA handles heterogeneity at the programming language and platform levels, and provides network transparency. This allows CORBA-based data integration approaches to focus on resolving metamodel and schematic heterogeneity. Mediator-based architectures have been used to integrate information from autonomous and heterogeneous data sources. They resolve heterogeneity at the level of data representation, i.e. metamodel and schematic heterogeneity. In the approach of this thesis, the useful features of CORBA have been utilised within a mediator-based architecture in order to support data integration.

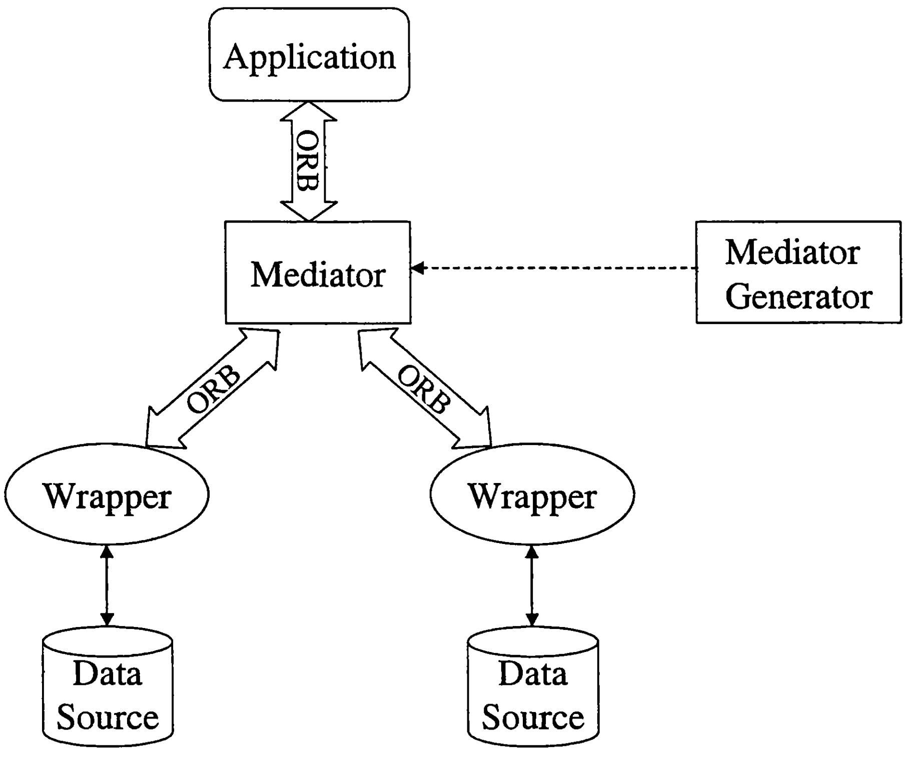

Figure 1-5: Mediator-based architecture utilising CORBA

A CORBA-based mediation system that supports the integration of data from molecular biology data sources and the creation of customised CORBA views of such data sources has been developed [Spiridou 00] and is described in this thesis. Its architecture extends the CORBA 3-tier architecture of Figure 1-3 into a mediator-based CORBA-utilising architecture (Figure 1-5). It is assumed that the data sources to be integrated provide access through CORBA. The system handles metamodel heterogeneity by defining a common model based on the CORBA Object Model, and a schema definition language based on the CORBA IDL. That leaves only the schematic heterogeneity between the different IDL-based schemas to be resolved.

Resolving schematic heterogeneity at the IDL level involves mapping from IDL schemas of available CORBA servers (source CORBA servers or wrappers) to a preferred IDL schema of a new CORBA server (target CORBA server or mediator). It is, of course, possible to develop target servers that implement the mapping manually. However, that requires in-depth CORBA experience, and developing/maintaining the code becomes cumbersome when many source servers are involved and in the light of frequently evolving source and/or target IDL schemas. The developed system supports semi-automatic generation of target servers based on high-level descriptions of the mapping from the source IDL(s) to the target IDL. In other words, it facilitates the creation of mediators by including a mediator generator. A view mapping language has been developed to define the common schema of a mediator and express the mapping between wrapper schemas and the common schema.

Compared to the information systems surveyed in Section 1.1.4, the approach examined in this thesis is utilising to a greater extent the useful features of CORBA. None of the systems of Section 1.1.4 uses CORBA to build its data integration solution. Instead each one has developed its own solution for accessing the data from data sources and integrating it. As a result, there is a lot of duplication of development effort in trying to resolve data sources heterogeneity. Another drawback is that the components of the resulting systems are less open to utilisation by independently developed software compared to the components of the mediation system examined in this thesis, which can be accessed through CORBA.

Having realised the benefits of CORBA, some of those information systems (i.e. the SRS [Etzold 96], and ACEDB [Durbin 94]) provide access to their data through CORBA, as does the mediation system examined in this thesis. The advantage is that integrated data can be manipulated easily and flexibly using any programming language supported by CORBA. Similar to the approach of this thesis, CORBA data access supported by SRS can include modelling of the data itself in IDL. On the contrary, ACEDB favours generic interfaces, as does the proposed system for biological database federations by Kemp et al. [Kemp 00]. A drawback of ACEDB CORBA access is the limited support for accessing metadata, which are necessary for the utilisation of generic CORBA interfaces. Other systems provide data access through Web interfaces, or specific APIs only (for example, OPM [Kosky 98], Entrez [Schuler 96]), making the utilisation of the integrated data more restrictive and cumbersome.

The approach examined in this thesis also differs from approaches of providing consensus/standard IDL definitions as a way to overcome schematic heterogeneity between different CORBA servers ([LSR], [Barillot 99a]). Although providing standard IDL definitions is advantageous in achieving interoperability and should be promoted whenever possible, there will probably be cases in which standardisation cannot be achieved, or definitions that are outside the scope of standardisation. In such cases, schematic heterogeneity could be resolved using the approach of this thesis. Furthermore, since the mediation system described in this thesis supports the creation of customised CORBA views, it could also, in principle, be used to provide views that conform to standard IDL definitions.

1.6 Outline of Thesis

The remainder of the thesis is organised as follows. Mediators in the framework of CORBA are introduced in Chapter 2 and issues that are important in the design of CORBA mediators are examined. The design of the mapping language based on the design dimensions of Chapter 2 is discussed in Chapter 3. In the same chapter, the mapping language is described in detail. The architecture of the system including design and implementation issues of the main components is described in Chapter 4. An example from the domain of molecular biology is used in Chapter 5 to demonstrate the usability of the system. Finally, conclusions are presented in Chapter 6.

Chapter 2: CORBA MEDIATORS

The approach proposed in this thesis uses CORBA within a mediator-based architecture. CORBA as well as mediators were introduced in Chapter 1. In this Chapter, mediators in the framework of CORBA are examined in detail. First, the architecture of CORBA mediators is considered and any relevant terms defined. A number of issues that are important in the design of CORBA mediators are then discussed, including a survey of alternative solutions for each design issue. The mapping language and the proposed system architecture examined in the following chapters of the thesis are developed based on the design dimensions of this Chapter.

2.1 Architecture

Integration of data from autonomous and heterogeneous data sources and restructuring of data representation can be achieved, in CORBA, with a mediator-based architecture like the one depicted in Figure 2-1.

CORBA *wrappers* provide IDL interfaces over their underlying data sources, which can, for example, be databases, or flat files. They may be generated automatically using available tools, like Persistence™ [Persistence], generated semi-automatically from high-level descriptions [Jungfer 99], or developed manually with/without the use of lower-level tools, libraries. A number of CORBA wrappers for molecular biology data sources are available ([Wang 00], [Coupaye 99], [ACEDB], [Rodriguez-Tomé 97], [Barillot 99b], [Hoebeke 01], [Hu 98], [Achard 98], [Parsons 00]).

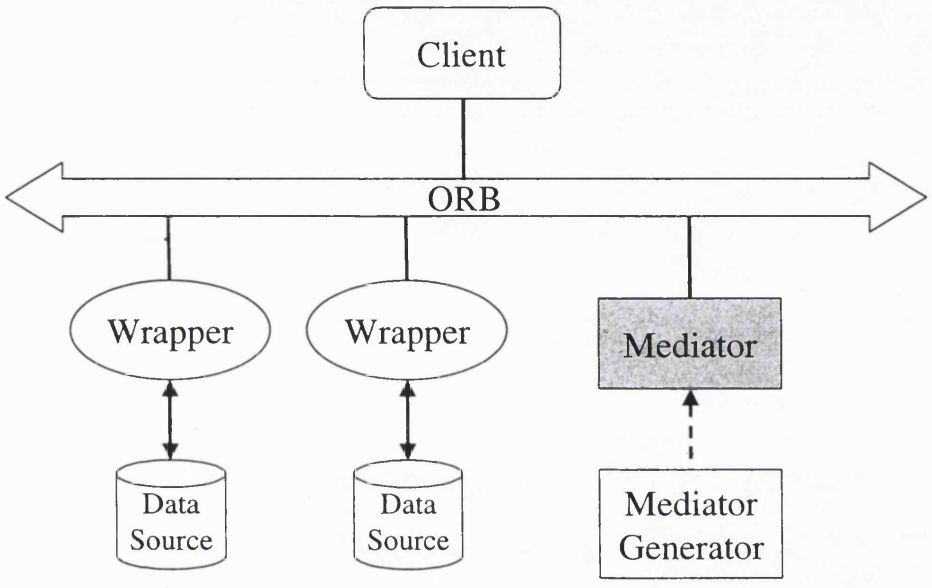

Figure 2-1: CORBA Mediator Architecture

The focus of this thesis is on the development of CORBA *mediators* that provide restructuring of representation and/or integration of data served by available CORBA wrappers. This involves defining new IDL interfaces and mapping those to the IDL interfaces of existing wrappers. Conflicts arising due to the dissimilar representation of data in different wrapper IDL interfaces (i.e. schematic heterogeneity) need to be resolved through the mapping. Manually implementing and maintaining the code of a mediator can be complicated and time consuming.

There is scope for automating much of the coding involved. In the database field, views have successfully been used for similar tasks, namely data integration and customisation of data representation ([Abiteboul 91], [Guerrini 97], [Kuno 96], [Kim 95], [Scholl 91]). By developing a view language that allows domain experts to give high-level descriptions of the required functionality of a mediator, the actual mediator code can be generated semi-automatically. Thus, a *mediator generator* is an important and frequently present component in a mediator-based architecture. Moreover, a *view mapping language* is often used for the implementation of mediator generators.

2.2 Common model and schema definition language

In a mediator-based architecture, a mediator typically defines a common model and often a common schema to represent the information served. A mediator generator typically uses a view mapping language to define the common schema as well as the mapping between existing wrapper schemas and the common schema. Choosing which common model and which definition language for the common schema to use is, obviously, an important decision in the design of any mediator.

A mediator can define its own common model and definition language. The advantage is that they can be designed in a way that suits the particular purpose of the mediator. A disadvantage is that mediator specifiers would need to learn that new model and language.

Alternatively, the common model and definition language of a mediator can be based on a model and language that already exist, are integrated and widely used in the context of the chosen mediator-based architecture, and are standardised. Advantages of this approach include relieving mediator specifiers from having to learn yet another model / language, and taking advantage of work done by others, and tools already developed. Of course, any existing models and languages chosen need to be suitable for the particular purpose of the mediator, and, in any case, extensions may be necessary.

Additionally, in the context of a CORBA mediator architecture like the one depicted in Figure 2-1, a common model and definition language may need to be transformed from/to the CORBA object model and IDL definitions. This would require the development of additional tools, and would probably reduce the runtime performance of the resulting system.

In the context of CORBA mediators, existing models and definition languages that could serve as a mediator common model and a definition language for common schemas are:

- Unified Modelling Language (UML)

"The UML is a language for specifying, visualising, constructing and documenting the artifacts of software systems, as well as for business modelling and other non-software systems" (OMG). It is widely used as a language for describing object models [Fowler 97]. It was developed by Rational Software (Rational) and its partners, and is the successor to the modelling languages found in the Booch [Booch 93], OMT [Rumbaugh 91], OOSE [Jacobson 92] and other methods. It has been adopted by the OMG as the standard modelling language [OMG 01A]. As an object modelling language, the UML includes a rich set of modelling constructs. For example, it includes classes, attributes, operations, associations, link attributes, aggregation, and generalisation.

Resolving schematic heterogeneity with CORBA mediators and at the UML level seems appealing. It means that the mediator common model and common schema definitions would be based on the UML, and the mapping between wrapper schemas and the common schema would be expressed in a language supporting the rich UML set of modelling constructs. Mapping specifiers would work with a well-defined, widely used and standardised common model, and a powerful, high-level mapping language.

Table 1: The MOF Meta-data Architecture

Layer	MOF terms	Examples
M3	meta-meta-model	MOF Model
M2	meta-models	UML Meta-model, IDL Meta-model
M1	Models	UML Models, IDL Interfaces
M0	user objects	

On the other hand, it means that translation between the UML world and CORBA/IDL world would be necessary, with translation from wrapper IDL definitions to UML object models, and from the UML common schema to the mediator IDL definitions. The Meta-Object Facility (MOF) Specification is an OMG standard which defines a set of object-oriented constructs that can be used to describe meta-models, such as the UML meta-model ([OMG 01c], [Frankel 99]). Meta-models belong in the M2 layer of the MOF meta-data architecture, which is shown in Table 1 with examples of meta-models and models for UML and IDL. The MOF also standardises the MOF-to-IDL mapping, though the provision of software tools implementing the mapping is not set as a conformance requirement by the OMG. The MOF-to-IDL taken together with a standardised UML-to-MOF mapping [OMG 01A], effectively produces a standard UML-to-IDL mapping. Software tools that support UML-to-IDL mapping have become available, such as Rose (Rational), though it is not clear whether they actually implement the standard UML-to-MOF and MOF-to-IDL mappings. In theory, the MOF can also be used to specify the OMG object model, treating it as a meta-model, and treating the IDL as an object modelling language. However, no IDL-to-MOF mapping is provided nor standardised by the MOF, making the translation in the opposite direction (i.e. IDL-to-UML) more difficult and prone to proprietary solutions.

- IDL

Resolving schematic heterogeneity with CORBA mediators at the IDL level means that the OMG object model is treated as a meta-model, which the mediator common model is based on, and the IDL is treated as an object modelling language, which the common schema definitions are expressed in. An obvious advantage of this approach is that no translation between different models and definition languages is necessary.

IDL includes the basic constructs that are essential in object-oriented modelling, i.e. *interfaces* (corresponding to object classes), *attributes* and *operations* as interface elements, *multiple inheritance* at the level of interfaces, object *collections* and a number of basic *types*. Although it lacks other more advanced modelling constructs, such as the ones provided by UML, the constructs it supports are sufficient to express object models of considerable complexity.

However, IDL is designed to describe interfaces between interacting objects within CORBA, not object models. Data can be represented in different ways using IDL. A representation should consider various parameters, such as simplicity (i.e. the best possible fit to the object model), required functionality (perhaps including extensibility), and efficiency.

For example, an object could be represented as an IDL interface or as an IDL struct, each approach having advantages and disadvantages ([Sellentin 98], [Jungfer 99], [Leser 98]). A representation using interfaces may be closer to the object model. Interfaces also have the advantage of supporting operations and multiple inheritance. However, interfaces can be slow. When clients request an entity represented as an interface, they receive only an object reference. Each subsequent attribute or operation request needs to go over the network, thus reducing performance. On the other hand, if a struct is used to represent an object, clients will receive a copy of the struct. All subsequent attribute requests are local, avoiding calls over the network and, as a result, improving performance. The disadvantage is that structs neither support operations nor inheritance, thus reducing representation flexibility and potentially making it necessary to define an IDL representation quite different to the object model.

An alternative representation (i.e. besides interfaces and structs) is supported in IDL as part of the CORBA/IIOP 2.3.1 Specification [OMG 99b]. It provides the possibility of representing an object using value types, which, in some sense, bridge the gap between interfaces and structs. More specifically, the implementation of value types is always local, and they support inheritance and operations. Thus, value types can be particularly useful when the main purpose of an object is to encapsulate data.

- eXtensible Markup Language (XML)

The XML is a format for structured documents and data on the Web. It is an open technology standard of the World Wide Web Consortium (W3C) for information exchange on the Internet [W3C 00]. It is a subset of the Standard Generalized Markup Language (SGML) that maintains the important architectural aspects of contextual separation. It is designed to enable the use of SGML on the Web. That is, to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with the HyperText Markup Language (HTML).

XML documents are tree-based structures of matched tag pairs containing nested tags and data. An XML element (i.e. a balanced tag pair) has a content (i.e. the material between the opening and closing tags) and, optionally, attributes (i.e.

name-value pairs). A Document Type Definition (DTD) defines the syntax of an XML document. That is, it defines the different kinds of elements that can appear in a valid document, the patterns of element nesting that are allowed, and the attributes that can be included in an element.

More recent XML developments that are relevant to our analysis on common model and schema definition languages include:

- *namespaces*: XML namespaces [W3C 99] provide a method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by Uniform Resource Identifier (URI) references. The goal is to provide support for multiple DTDs in the same document,
- *linking*: There are two technologies providing advanced linking facilities integrated with Web technology. XLink [W3C 01b] is for cross document links and XPointer [W3C 01a] is for links within a document,
- *schema*: The XML Schema definition language [W3C 01c] offers facilities for describing the structure and constraining the contents of XML documents. The schema language provides a superset of the capabilities found in XML DTDs. The XML Schema specification also defines data types.

The XML Metadata Interchange (XMI) Specification is an OMG standard [OMG 02] that enables easy interchange of metadata between modelling tools (based on the OMG UML) and metadata repositories (based on the OMG MOF) in distributed heterogeneous environments. It integrates three standards, i.e. the XML, the UML and the MOF. It uses the XML as a metadata interchange format. It also standardises the MOF-to-XML DTD mapping, and applies that mapping to the UML meta-model deriving a complete XML DTD for the UML meta-model.

Resolving schematic heterogeneity with CORBA mediators at the XML level has the advantage of using a standardised and widely supported language. It can also integrate with UML object models through the use of MOF. However, there are certain limitations. Compared to the UML, the XML DTD language lacks some advanced constructs for object modelling. For example, data types support is restricted, nesting is emphasised over linkage, and inheritance is not supported explicitly. Thus, when mapping a UML object model to an XML DTD certain UML constructs are mapped to lower level XML ones. Moreover, the XML DTD language can only express a subset of the structure and consistency rules contained in a MOF metamodel. For example, multiplicities on MOF attributes and associations are not fully supported, MOF constraints are not supported, and data types support is poor. As a result, a consumer of an XMI document may need certain extra knowledge not included in the XML DTD in order to check the semantic correctness of a document, or to reconstruct a model (belonging in the M1 layer of the MOF meta-data architecture, Table 1) in its original form e.g. with the correct CORBA data types. Of course, new XML technologies recently developed or currently under development are improving on some of the above areas. For example, XML Schema provides support for data types, and XLink and XPointer provide linkage support. However, it will still take some time before those technologies and their support in XMI are standardised, and associated tools that fully support those standards become available.

Similar to the case of UML, resolving schematic heterogeneity with CORBA mediators at the XML level also means that translation between the XML world and CORBA/IDL world would be necessary. That is, translation from wrapper IDL definitions to XML DTDs, and from the common schema expressed as an XML DTD to the mediator IDL definitions. The MOF standardises the MOF-to-IDL mapping and the XMI standardises the MOF-to-XML DTD mapping. However, the mappings in the opposite directions are not as simple, and not standardised. The XML DTD-to-MOF mapping has the problems discussed above due to the limited expressive power of the XML DTD language. Even the more recent OMG Specification on XMI production of XML Schema [OMG 01b] defines the provision of XML-to-MOF, XML DTD-to-MOF, and XML Schema-to-MOF mappings as optional only requirements for compliance. The IDL-to-MOF mapping is not standardised by OMG either. Thus, translation in the reverse engineering direction becomes more difficult and prone to proprietary solutions.

2.3 Target object derivation

In view design, the potential for restructuring data representation depends, among other things, on the flexibility for target object derivation. More specifically, it depends on how many and of what type source objects a target object can be derived from. The different possibilities in the CORBA environment are:

- Same interface source objects

One possibility is to support target object derivation from one or more source objects of the same interface only. This would restrict restructuring to relatively simple operations, such as adding, removing, or renaming an attribute or an operation.

- Object assembly

Another possibility is to allow a target object to be derived from one or more source objects of the same or different interfaces, while restricting composition at the object level. This is known as object assembly (see Figure 2-2).

More specifically, in object assembly, a target object can contain a number of source objects, each of which conforms to a particular interface. The object references of the source objects and their interfaces are accessible to clients. In other words, a target object provides multiple interfaces; each interface provides a view of the capabilities of the target object.

OMG is in the process of defining a CORBA component model. The idea is to be able to describe object-oriented software entities and assemble them into applications. The proposed CORBA component model [OMG 99a] supports object assembly, calling the element interfaces *facets* and the resulting overall interface *component.* Although the component model is not designed for data integration, it could potentially be used to achieve a restricted form of data integration, that is, data integration at the coarse object level.

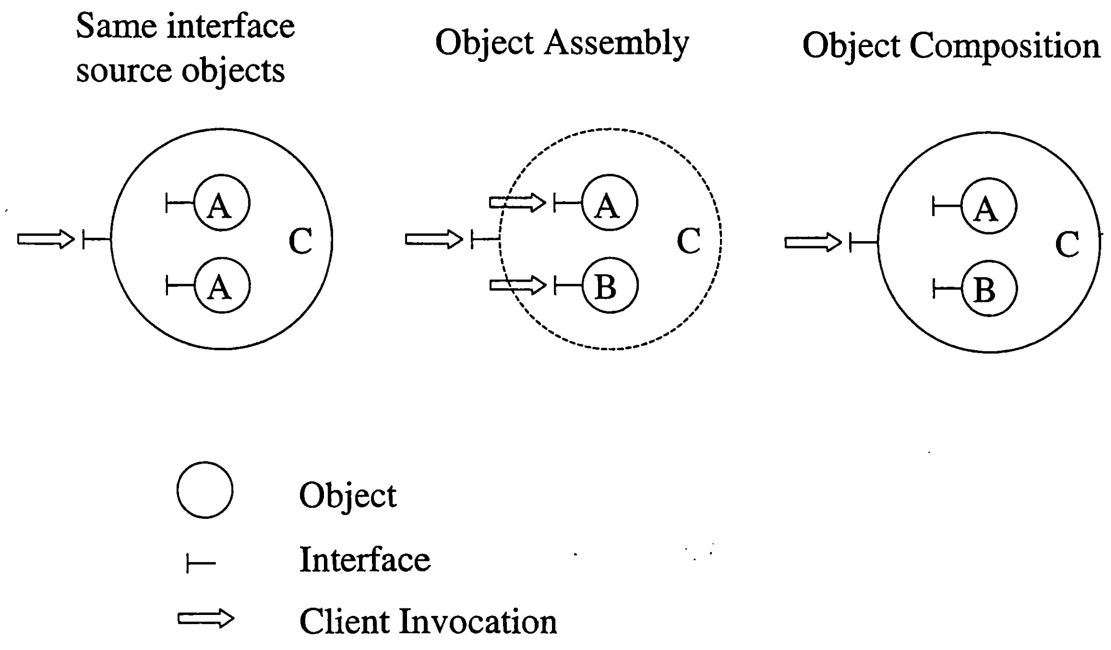

Figure 2-2: Target Object Derivation

- Object composition

The third alternative is to allow a target object to be derived from one or more source objects of the same or different interfaces, and to support composition at the object element (i.e. attribute/operation) level. This is known as object composition (see Figure 2-2).

More specifically, object composition enables the inclusion of a number of objects with different interfaces within a composite (target) object that conforms to a single interface. Only the interface of the composite object is accessible to clients. Attributes, operations of the composite interface are mapped internally to attributes, operations of included interfaces.

One can easily see that, compared to the previous two approaches, object composition allows for the most complex restructuring possibilities.

2.4 Publishing initial CORBA objects

A CORBA server typically publishes (i.e. makes available to clients) the references of a small number of objects that constitute server entry points. That is, starting from those initial objects, a client can then reach other objects contained in the server by navigation. A client can obtain such initial object references in one of three standard ways:

- *Explicit References*: A server can convert the Interoperable Object References (IORs) of CORBA objects to be published into strings (i.e. using the *object-to-string* CORBA operation), and then write those strings in files, the URLs of which the server publishes. A client can then use the *string-to-object* CORBA operation to obtain the object references from their stringified form. This approach is simple and easy for clients to use. It works well for a small number of objects that do not necessarily belong in a naming hierarchy. However, it requires maintaining one or more files that do not belong in the CORBA environment.
- *Naming Service*: A server can use the CORBA Naming Service [OMG 98b] to attach names to objects and store them in a hierarchical structure. Then, a client browses through the hierarchy and retrieves references to objects by their names. This approach is suitable for CORBA objects that conceptually

belong in a naming hierarchy. It requires access to the Naming Service, and compared to the first approach, it is not as simple for clients to use.

- *Trader Service*: A server can use the CORBA Trader Service [OMG 98b] to advertise CORBA objects, which can later be retrieved by clients given a number of object property values. This approach is suitable when clients do not know the names of objects they are looking for. It requires access to the Trader Service.

A server can, of course, publish object references in more than one way, thus providing alternative possibilities to clients for obtaining its initial object references.

2.5 Wrapper query support

In CORBA, generic wrappers need to support queries in a way that conforms to standardised IDL definitions. The CORBA Query Service Specification [OMG 98b] defines a framework for the support of queries on collections of objects. It specifies a number of generic interfaces (see Figure 2-3), the most important of which are:

- *Query evaluators* which accept queries as strings, evaluate them and return the results the type of which is unrestricted (i.e. IDL type *any*).
- Collections of objects which are supported by the *collection* interface. It defines methods for adding or removing members as well as for creating an associated iterator.
- *Iterators* which support manipulation of collections, including traversal over and retrieval of the objects within the collections.
- A *queryable collection* which is both a query evaluator and a collection, so it can serve as both the result of a query and the scope of another query.

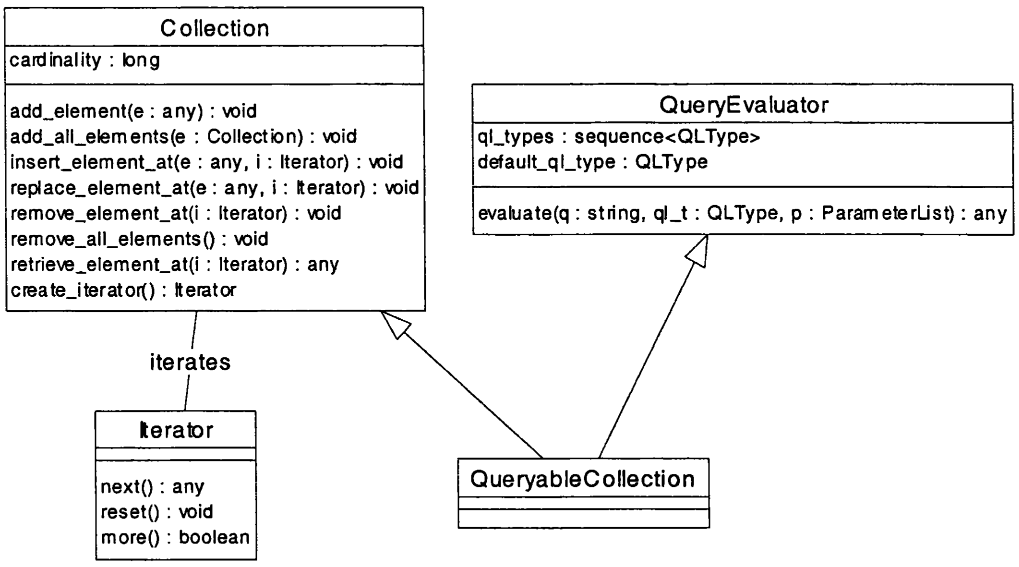
Collection
cardinality : long
add_element(e : any) : void
add_all_elements(e : Collection) : void
insert_element_at(e : any, i : Iterator) : void
replace_element_at(e : any, i : Iterator) : void
remove_element_at(i : Iterator) : void
remove_all_elements() : void
retrieve_element_at(i : Iterator) : any
create_iterator() : Iterator
QueryEvaluator
ql_types : sequence<QLType>
default_ql_type : QLType
evaluate(q : string, ql_t : QLType, p : ParameterList) : any
iterates
Iterator
next() : any
reset() : void
more() : boolean
QueryableCollection

Figure 2-3: UML Model of main CORBA Query Service Interfaces

Queries are specified as strings. The Query Service is designed to be independent of any specific query language used. At the same time, and in order to provide query interoperability, a compliant Query Service implementation must support either SQL Query or OQL.

Weak points of the Query Service Specification include not addressing query optimisation and indexing issues, and not providing meta-data support ([Leser 98], [Wells 94]). It has also been argued that an efficient implementation of the CORBA Query Service is not straightforward [Rohm 99]. In an implementation of a CORBA Query Service, called Harmony [Rohm 99], ORB-specific features, such as Orbix's smart proxies and loaders [IONA 97], had to be used in order to achieve good performance.

In its effort to be generic, the CORBA Query Service accepts queries passed as strings, and leaves the representation of query results undetermined. This means that any implementation of the service has to decide on the definition of an appropriate query language and the description of the query result data structures.

One approach that seems to fit well to the model of the Query Service is to accept arbitrary ad hoc queries expressed in SQL or OQL, and return strings as a result. Advantages of this approach include the support of powerful queries that, in addition, can be optimised by the underlying database, as well as the support of database schema evolution. A disadvantage is that ad hoc queries require that the clients have meta-data knowledge, something that is not supported by the Query Service. Another related problem is that clients would need to parse the query results.

Another approach could be to accept queries that give predefined CORBA types as query results. Typically, such types would be (homogeneous) collections of CORBA objects (i.e. modelled by IDL interfaces, or structs). Advantages of this approach include the convenience of clients, since they retrieve fully typed query results, and the support of queries that are sufficiently powerful for many applications. However, utilising full query power is not possible, since result types are predefined and cannot be joined to create new ones. Although this approach would well suit cases in which domain objects are represented at the IDL level, the CORBA Query Service Specification does not support it in an easy and clear way, since it allows for arbitrary queries passed as strings and leaves the representation of query results undetermined.

Perhaps due to such shortcomings of the CORBA Query Service Specification, available wrapper systems that support CORBA queries over underlying data sources have implemented their own 'flavour' of CORBA queries. They do not necessarily conform to the standard Query Service Specification but adapt it to their own needs. Some of those CORBA wrapper systems are briefly examined here.

The Persistence™ object-relational mapping software and its Distributed Object Connection Kit (DOCK) [Persistence] support automatic generation of CORBA wrappers over relational databases. Each relational table, named *<ObjectName>*, is mapped to an IDL interface with the same name. Table columns are mapped to readonly attributes. In addition, for each table an extra IDL interface is created, named *<ObjectName>Factory*, that supports querying the corresponding table/interface. A number of query operations are provided, the most important of which are: *queryKey*, which takes the key of the relational table (i.e. the unique identifier of the CORBA object) as input, and returns a single CORBA object of the same interface; *querySQLWhere*, which takes an SQL where-clause string as input, and returns a sequence of CORBA objects satisfying the conditions; *query*, which is similar to querySQLWhere, but has a special syntax to support joins for the specification of conditions.

A semi-automatic approach for generating CORBA wrappers over relational databases is followed by Jungfer et al. [Jungfer 99]. They provide a declarative language to describe the mapping between relations and IDL constructs. Using the mapping information, they generate a CORBA server, which also supports queries. Queries are expressed in their own query language that is similar to an SQL where-clause. Query results are represented by a predefined type that can be either a struct or an interface.

Sellentin et al. [Sellentin 99] deal with data intensive applications. They developed a CORBA Query Service to access EXPRESS-based data. Their prototype implementation accepts a number of selected rudimentary queries as input. They define generic IDL data structures that are used as query results. They also provide meta-data support to decompose query results, which, initially, have to be of a specific type.

Harmony [Rohm 99] accepts execution plans as query input, and returns either a single value or a CORBA collection as a result. The examples provided show that CORBA collections returned are always of a specific predefined CORBA type.

2.6 Mediator query support

Mediators typically provide queries over the integrated data they serve. They translate queries received from applications into sub-queries to wrappers, and translate answers from wrappers into a form appropriate to applications. In the context of CORBA mediators there are three different approaches to querying:

- *Ad hoc queries*: These are query strings expressed in SQL or OQL. Query results are returned as strings by mediators. The main advantage of this approach is the support of powerful queries. Disadvantages include that clients would need to parse the query results, and the requirement that the clients have meta-data knowledge, something that is not supported by the CORBA Query Service.
- *Type restricted queries*: The results of these queries are restricted to be of a pre-defined CORBA type. The advantages and disadvantages of this approach are similar to the ones discussed in Section 2.5 for type restricted queries of CORBA wrappers.
- *Fixed parameterised queries*: These are fixed queries encapsulated in parameterised operations in the target schema. Clients do not use any query language, only the available operations. Query results are under the control of the corresponding operations. Advantages of this approach include the convenience of clients, since they retrieve fully typed query results, which, in addition, are not restricted in any way. It also suits cases in which domain objects are represented at the IDL level, and it does not require the CORBA

Query Service since it only uses plain CORBA operations. An obvious disadvantage is the loss of query flexibility since the queries are fixed.

2.7 Conclusions

A number of issues that are important in the design of CORBA mediators have been discussed in this Chapter. Some of these issues apply to the design of any mediator (e.g. the choice of the common model and schema definition language), while others are specific to mediators utilising CORBA (e.g. the publication of initial CORBA objects).

In any case, when designing mediators that utilise CORBA, the particular strengths and possible weaknesses of CORBA need to be carefully considered. It is to the advantage of any CORBA mediator system to utilise as much as possible the large amount of development work on CORBA. At the same time, alternative solutions may need to be considered in cases where the specific CORBA standard does not serve the purposes of the CORBA mediator system. For example, CORBA mediator systems have to carefully consider the use of the CORBA Query Service or alternatives for wrapper query support.

For each one of the design issues examined in this Chapter, alternative solutions have been discussed. The chosen solution, in each case, will shape the resulting CORBA mediator system.

In particular, the main design decisions for the system described in this thesis have been the chosen solutions for each of the design dimensions examined in this Chapter. Both the mapping language and the architecture of the system are developed based on those design decisions. The design and implementation of the mapping language are examined in Chapter 3, while the system architecture including the design and implementation of its main components are examined in Chapter 4.

Chapter 3: THE MAPPING LANGUAGE

A number of design issues that are important for the development of CORBA mediators were discussed in Chapter 2. Those influence the design of the view mapping language used for the implementation of mediator generators. A view mapping language typically defines the common schema and the mapping between wrapper schemas and the common schema. The design of the mapping language, which has been developed as part of the CORBA mediation system described in this thesis, is based on the design dimensions of Chapter 2.

Chapter 3 begins with the identification of the general requirements for the mapping language. The design of the mapping language is, then, discussed in Section 3.2. That is followed by a detailed description of the mapping language in Section 3.3, and concluding remarks.

3.1 General requirements

The general requirements for the mapping language are:

- To support a high-level notation

The language developed as part of the CORBA mediation system described in this thesis describes mappings between schemas. A high-level notation is required in order to express such mappings easily and concisely. This would also facilitate modification of the mapping definitions when necessary, e.g. in case the source or target schemas evolve.

- To support constructs for object models

The mapping language is designed for mapping one or more source schemas into a target schema, all defined in IDL. It needs to support a subset of the modelling constructs included in the OMG Object Model [OMG 98a]. The supported constructs need to be able to express object models. That is, constructs that model objects, inheritance between object definitions, the definition of object characteristics, and collections of objects are required.

- To support read-only access

As far as access rights to data are concerned, the focus of the proposed system is to access data, not to create new or change existing data. Both objectives of creating customised representations of source data and integrating source data into a target CORBA server can be satisfied by assuming read-only access rights to source data, and supporting read-only access in the mapping language.

- To provide a way to publish initial object references

The target CORBA server accessed by clients would need to publish a number of initial object references. At least one way of publishing such references, as examined in Section 2.4, should be provided.

- To support a suitable mediator and wrapper query model

Mediator queries are mapped to queries supported by available wrappers. When designing the mapping language, the query model to be supported should be chosen. That includes the type of queries supported by mediators (alternatives discussed in Section 2.6) and at least one mapping to available wrappers (alternatives discussed in Section 2.5).

3.2 Design

In order to support a high-level notation for the mapping definitions, the language combines declarative and procedural ways to express mappings. In the following sections, the design of the mapping language is examined in terms of the design dimensions of Chapter 2. For each design dimension, the preferred solution and the rationale behind the choice made are presented.

3.2.1 Common model and schema definition language

In the system described in this thesis, schematic heterogeneity is resolved at the IDL level. This means that the OMG object model is treated as a meta-model, which the mediator common model is based on, and the IDL is treated as an object modelling language, which the common schema definitions are expressed in.

This approach has certain advantages. First, a model and language are chosen that already exist, are integrated and widely used in the context of the chosen mediator-based architecture (i.e. within CORBA), and are standardised. Thus, mediator specifiers do not have to learn a new model / language. Second, no translation between different models and definition languages is necessary, as would be the case for UML- or XML-based solutions.

IDL constructs, though not as rich as UML ones, suffice to express object models of considerable complexity. In order to support the essential constructs for object-oriented modelling, the mapping language includes:

- IDL *interfaces* for modelling objects.
- *Multiple inheritance* at the IDL interface level.
- *Attributes* for describing object characteristics. Read-only attributes are supported as opposed to read-write ones, since read-only data access is required.
- *Operations* for describing object functionality. IDL operations can raise *exceptions*. Exceptions are not an essential object-oriented modelling construct. However, if an exception is thrown by any source IDL operation that is used in the mapping of a target IDL operation, it must either be

caught or thrown by that target IDL operation, in order to ensure correct compilation of the generated target IDL and target server implementation. This is the reason for allowing operations declared in the mapping language to raise exceptions defined in source IDL(s). In order to keep the implementation simple, declaration of new user exceptions in the target IDL is not supported.

- A number of *basic types*.
- *Sequences* for modelling collections of objects.

Other IDL constructs such as *structs* and *value types* are not supported in the current implementation.

The mapping language is designed in a way that its rules about valid names and scope are at least as restrictive as the corresponding rules followed in the IDL. In some cases, and for reasons of simplicity, the rules of the mapping language are more restrictive compared to those of the IDL. This approach guarantees that any generated target IDLs will be valid.

In the system described in this thesis, target servers are semi-automatically generated based on view definitions. Developers can, of course, customise the resulting server code to suit their specific requirements. In addition to that, inclusion of user implementation within the view specification itself can be used in order to create prototype target servers easily and rapidly, and to isolate mapping specifiers from the implementation details of generated CORBA servers. In particular, inclusion of user methods within the view specification greatly enhances mapping possibilities, yet it does not overload the view specification with many implementation details. For instance, user methods can implement new (i.e. non-derived from existing ones) attributes / operations / interfaces of target servers. They also provide an alternative and powerful procedural way to implement mappings that are too complex/impossible to express in other ways.

For these reasons, the mapping language supports user methods inclusion within the view specification.

Alternatively, schematic heterogeneity could be resolved at the object modelling level, i.e. at the UML level with/without the use of XML. While bidirectional UML-MOF-IDL and XML-MOF-IDL mappings are not fully standardised, and associated tools fully supporting the standard mappings are not yet available, the system described in this thesis provides a way for resolving schematic heterogeneity for a selected set of object modelling constructs at the IDL level.

3.2.2 Target object derivation

The model followed in the approach described in this thesis is based on *object composition* (described in Section 2.3). Compared to the *same interface source objects* model and to the *object assembly* model (described in Section 2.3), *object composition* enables more powerful data integration and customisation of data representation, since composite object interfaces can define new attributes, operations and internally map them to attributes, operations of the included interfaces.

Source references are introduced in the mapping language in order to support object composition. Source references, as explained in Section 3.3.7, are references to those source objects, which the target object is derived from. Source references provide the means to access source objects and combine their elements (i.e. attributes, operations) in whatever way is appropriate in order to derive the elements of the composite target object.

3.2.3 Publishing initial CORBA objects

In the system described in this thesis, target servers do not implicitly publish any CORBA objects. The mapping specifier has the flexibility to explicitly specify the objects to be published. As described in Section 3.3.5, the mapping language provides a convenient way to publish initial object references of the generated server, i.e. using the *location* field.

Clients can then access those initial object references using the *explicit references* option, as described in Section 2.4. Each target server is expected to publish only a few factory-type objects, so the *explicit references* approach would typically be sufficient and easy for clients to use. Using the *Naming Service* would also be a suitable alternative but it is not supported in the current implementation.

3.2.4 Wrapper query support

The approach described in this thesis is based on the idea that domain objects are represented in IDL. In this case, suitable query support would be achieved if queries are restricted to IDL interfaces and query results are collections of CORBA objects of predefined interfaces that satisfy the specified query conditions. In other words, wrappers would need to support *restriction* queries (also called *selection*) on collections of CORBA objects, returning subsets of those collections that include only the objects satisfying the query conditions. Advantages of this approach, as discussed in Section 2.5, include the convenience of clients, since they retrieve fully typed query results, and the support of queries that are sufficiently powerful for many applications.

In the mapping language, the notion of special CORBA objects (called factories) that may support restriction queries on collections of CORBA objects is included. When factories provide query support, they act essentially as query evaluators that accept an SQL where-clause as input and return a collection of the objects satisfying the given conditions. In the implementation, any specified queries are translated to *querySQLWhere* operations like those supported by the Persistence™ generated factory objects. That is, a mapping to Persistence™ generated CORBA wrappers is provided, since the functionality those wrappers provide is very close to the required query support. In principle, it would not be difficult to provide mappings to alternative wrappers, provided that their query model is close to the query model outlined above.

It should be noted that, although wrapper query support enhances mapping possibilities (by allowing the mapping specifier to map a source reference to a *query* within an object specification mapping) it is not a compulsory requirement. Mapping can be specified using the other options provided by the language.

3.2.5 Mediator query support

The approach described in this thesis is based on the idea that domain objects are represented in IDL. It was argued in Section 2.5 that the way ad hoc queries are specified in the CORBA Query Service is not well suited to CORBA servers that represent domain objects in IDL.

Instead, fixed parameterised queries are supported, which (as discussed in Section 2.6) are encapsulated in CORBA parameterised operations of the target schema. The mapping language supports the specification and mapping of such operations. The main advantage of this approach is that query results are fully typed and the type is controlled by the mapping specifier (i.e. the operation return type defined by the mapping specifier).

Another possibility could be to support type restricted queries, as described in Section 2.6. Similar to fixed queries, type restricted queries are also suitable for domain objects represented in IDL and provide fully typed query results. This approach is not supported in the current implementation.

3.3 Description

An overview of the mapping language is provided below (words in bold are language keywords; the star symbol denotes zero or more occurrences; a pair of brackets denotes optional inclusion). Any necessary IDL files can be included using *#include*. The *server* declaration part contains the declaration of any source factory objects. Views are defined within a *module*. A *view* can be forward declared; it has a name, and optionally a list of views from which it inherits. Within each view, the following parts are defined:

- an optional *location* where the view instance may be published,
- the *definition* of the target IDL interface corresponding to the view,
- the *source* part defining references to source CORBA objects,
- the *mapping* part defining the mapping for attributes / operations of the view, and
- the *user_method* part containing the implementation of any user defined methods used within *mapping*.

```
// Comments
(#include "<file_name>") *

server <server_name> {
   ( factory <factory_name> {
      ftype <type_name> ;
      ior "<url>" ;
      }
   ) *
}

module <module_name> {
   ( view <view_name> ; ) *
   ( view <view_name> [: <superview_list>] {
      location "<url>"
      definition {
            ( <idl_dcl> ; ) *
          }
          source {
            ( <source_ref_type> <source_ref_name_list> ; ) *
          }
          mapping {
            ( <target> = <source> ; ) *
          }
          user_method {
            <user_method_list>
          }
        }
      ) *
    }
```

In the following, the mapping language constructs are examined in more detail. The full specification of the mapping language in BNF form is included in Appendix A.

3.3.1 Include directive

Definitions from other IDL files can be included using the *#include* directive. Only those files containing definitions used within the *definition* part of the views need to be included. All *#include* directives appearing in the mapping specification will also appear in the generated target IDL file.

3.3.2 Server declaration

The server declaration part includes the definition of the generated server class name (given after the keyword *server*) and the declaration of any factory objects.

In the context of the mapping language, a *factory* is a CORBA object of a source server with a published IOR. Typically, factory objects will support queries, allowing the retrieval of other CORBA objects. Queries can either be in the form of plain CORBA parameterised operations, or simple SQL where-clause queries the syntax of which is discussed later in the *mapping* part.

In the latter case, queries will be translated to *querySQLWhere* operations like those supported by the Persistence™ generated factory objects, described in Section 2.5. So, the corresponding source server factory objects should have either been generated using the Persistence™ software, or should support a *querySQLWhere* operation similar to the one supported by Persistence™ generated factories.

A factory declaration must include:

- the name of the factory, given after the keyword *factory*; in the rest of the specification, the factory object will be referred to by its fully scoped name, that is, the server class name, followed by the '.' character, followed by the factory name,
- the type of the factory, given after the keyword *ftype,* and
- the URL where the Interoperable Object Reference (IOR) of the factory can be found; it is given after the keyword *ior.*

For example:

```
server MyServer {
  factory factA {
    ftype Server1.factoryA;
    ior "file:/server1/ior/factoryA.ior";
  }
}
```

3.3.3 Module

A module is a kind of container for view definitions. It is used for name scoping. For details on names and scoping see Section 3.3.10.

3.3.4 View

View is the main construct in the mapping language including, among others, the definition of target IDL attributes/operations and their mapping to source IDL(s) elements. Each view defined in the mapping language maps to an interface in the target IDL.

- View header

A view has a name, and optionally a list of views from which it inherits.

- Forward declaration

A view can be forward declared to allow for cross-referencing. It is not allowed to inherit from a forward-declared view whose definition has not yet been given. For example:

```
module M {
   view V1;             // Forward declaration
   view V2 : V1 {       // Error: V1 has not yet been defined
   }
   view V1 {
   }
   view V2 : V1 {       // Ok
   }
}
```

- View inheritance

A view can inherit from another view, which is then called a *base* view of the derived view. A derived view can declare new elements (i.e. types, attributes, and operations). It can also refer to the elements of a base view as if they were elements of its own. Inheritance in the mapping language, generally, follows the rules for interface inheritance defined by OMG [OMG 98a]. Similar to IDL rules, a derived view cannot redefine attributes, or operations. Contrary to IDL, and for reasons of simplicity, a derived view cannot redefine types either. For example:

```
view V1 {
  typedef sequence <string> StringList;
  string op1();
}
view V2 : V1 {
  short op1();                // Error: V2 redefines an inherited
                              // operation name
  string StringList();        // Error: V2 redefines an inherited
                              // type name
}
```

The mapping language supports multiple inheritance for view definitions. That is, a view may be derived from any number of base views inheriting their definitions. Inheriting from the same view through more than one inheritance path is allowed (this is called "diamond" shape inheritance). For example, view *D* in Figure 3-1a inherits from view *A* through two paths, one path via view *B* and a second one via view *C*. It is also allowed to inherit from two views, one being derived from the other, as is the case of view *E* in Figure 3-1b, which inherits from views *B* and *A*. It is not allowed to inherit from two different views with the same attribute, operation, or type name, a rule which guarantees that references to base view elements are unambiguous.

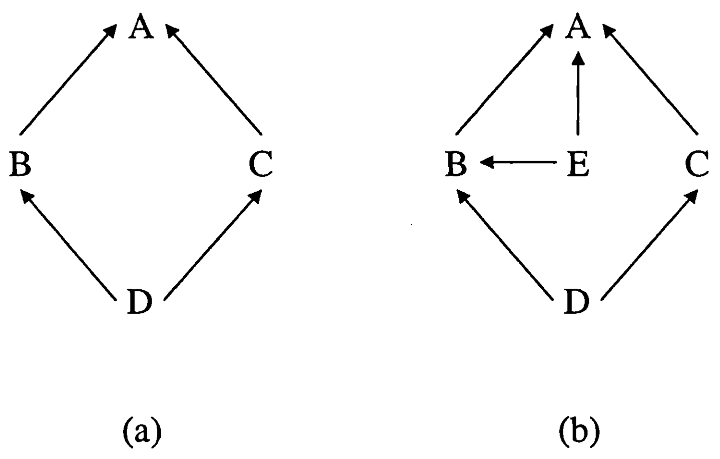

Figure 3-1: Legal Multiple Inheritance

The mapping language also supports single inheritance for view implementations. That is, a view can inherit the implementation of, at most, one other view using the *extends* keyword. For example in the following code, view *V* inherits the definitions of views *V1, V2* and *V3* and the implementation of view *V2*.

```
view V1 {...}

view V2 {...}
view V3 {...}
view V : V1, V2 (extends), V3 {...}
```

- View body

A view is comprised of the following parts: *location, definition, source, mapping* and *user_method.* Empty views (i.e. containing none of the above parts) are allowed.

3.3.5 Location

A target CORBA server needs to publish one or more factory-type object through which other objects can be obtained. Like every CORBA object, factories are defined as IDL interfaces. However, only one instance of such an interface is required, which is created and then published by the server.

The mapping language provides an easy way to create target server factory objects, i.e. using the *location* field. The presence of a location field in a view specification denotes a published factory object. The location specification starts with the keyword *location*, which is followed by the URL of the filename into which the corresponding factory object IOR will be written. This information is used to generate target CORBA server code that creates a factory object of the associated interface and writes its IOR into the specified file. For example:

```
location "file:/tmp/BactGen.GenFactory.ior"
```

3.3.6 Definition

This is the definition of the view. It includes the declaration of attributes, operations, and types. The syntax for their declaration is the same as the IDL syntax.

```
Definitions      ::="definition" "{" DefinitionBody "}"
DefinitionBody   ::=(Definition ";")*
```

```
Definition        ::= TypeDecl
                    | AttributeDecl
                    | OperationDecl
```

- Attribute declaration

The mapping language supports read-only attributes.

- Operation declaration

The mapping language supports declaration of IDL operations. Target IDL operations must either catch or throw any exceptions defined in source IDL operations used in the mapping.

If the details of source server exceptions need to be hidden from the clients of the target CORBA server, the developer of the target CORBA server has to catch and deal with those source IDL exceptions within the implementation of the relevant target IDL operations.

If, on the other hand, source server exceptions need to be forwarded to clients of the target CORBA server (informing them, for instance, of incorrect domain object identifiers, or problems with a back-end database) target IDL operations can simply raise those source IDL exceptions in the mapping language.

- Type declaration

The mapping language supports the following IDL types: base types (i.e. floats, integers, chars, boolean, octet, any), template types (i.e. strings, unbounded sequences), and type names.

Type declaration, that is, association of a name with a data type, is used for declaration of sequences.

3.3.7 Source references

A target CORBA object is typically derived from (i.e. retrieves its data from) one or more object of source CORBA server(s). References to those source objects are declared within *source*. Values (i.e. source objects) are assigned to those references during creation of the target object.

```
SourceRefs       ::="source" "{" (SourceRefDecl ";")* "}"
SourceRefDecl    ::=JavaScopedName ( "[""]" )? <id> ("," <id>)*
```

A source reference type is a scoped name that may optionally be followed by brackets in order to denote a sequence of objects.

In order to avoid naming conflicts source reference names follow the same rules as attribute / operation / type names do. That is, source reference names follow the case-sensitivity and name ambiguities in inheritance rules for identifiers defined in Section 3.3.10, as well as the naming in the presence of inheritance rules defined in Section 3.3.4.

3.3.8 Mapping

This defines the mapping for attributes / operations of the view, which are either declared within *definition* or inherited. The attribute / operation name of the target IDL to be mapped goes on the left-hand side of the mapping assignment. It can be mapped to a scoped source name, a user supplied Java method, a Java expression, or a specification for creation of new object(s).

```
Target           ::= <id>"("")" ( "[]" )?
Source           ::= MyScopedName
                   | JavaMethod
                   | JavaExpression
                   | ObjectSpec
```

- Mapping to a source name

A source name is the name of an attribute, or operation. Source references can be used in a source name. For example:

```
op() = a.source.op();
```

- Mapping to a user method

A user method is a call to a Java method that is implemented within *user_method.*

```
JavaMethod      ::= "jmethod" "{" <id> Arguments "}"
Arguments       ::= "("")"
                  | "(" Argument ("," Argument)* ")"
```

For example:

```
op() = jmethod { myOp() };
```

- Mapping to a Java expression

The mapping specifier can use any Java expression that can be placed on the right-hand side of a plain assignment. Source references as well as standard Java methods can be used in the implementation. When using abbreviated names to refer to classes in a Java expression, it may be necessary to manually include the corresponding Java *import* statements in the generated code. Java expressions are included *as-is* in the generated target server code.

```
JavaExpression  ::= "jexpression" "{" stringToMatchingBrace "}"
```

For example:

```
op() = jexpression { ( ref.posA() + ref.posB() ) / 2 };
```

- Mapping to an object specification

The keyword *object* starts the mapping specification for the creation of one or more new target objects. When new target objects are created, their source references must also be assigned. A source reference can be mapped to a query, a user method, or a scoped source name.

ObjectSpec ::= "object" "{" ComponentSpec ("," ComponentSpec)* "}"

ComponentSpec ::= <id> ("[""]")? "=" (JavaMethod | QuerySpec | MyScopedName)

In the case of a query, the name of the source factory object (to which the query will be sent) is given as the first argument with JavaScopedName. An SQL where-clause specifying the query conditions is given as the second argument with Expression. The predicates of the SQL where-clause are formulated using attribute names, operation names, parameter names of the current operation, source reference names of the current object specification, or source reference names of the current view specification. Conditions are specified using the predicates '<', '<=', '>', '>=', '=', and '<>'. Predicates can be negated using the logical operator 'not', or combined using the logical operators 'and', 'or'.

QuerySpec ::= "query" "(" JavaScopedName "," Expression ")" ("[" (<int>)? "]")?

For example:

```
op() = object {
  ref = query( Server.factA, id()='an_id' and type()='a_type' )
};
```

- Iterator

In order to facilitate mapping an *iterator* operator is defined, which provides a convenient way to iterate over sequences of objects. It is, essentially, a shortcut for a simple *for* loop. It is denoted with a pair of brackets "[]" and can be used within target mapping or source reference mapping. It is always used in pairs, appearing in the left- and the right-hand side of a mapping. Its semantics are that for each object encountered in the right-hand side of a mapping a new object should be created in the left-hand side. The possible pairs are:

a) Target – Source name; for example:

```
op()[] = an.op()[];
op()[] = another()[].op();
```

b) Target – Object specification source_name; for example:

```
op()[] = object {
  ref = a.source.op()[]
};
```

c) Target – Object specification query; for example:

```
op()[] = object {
  ref = query ( Server.factory, type()='a_type' )[]
};
```

d) Object specification source reference name - Object_specification source name; for example:

```
op() = object {
  ref[] = an.op()[]
};
op() = object {
  ref[] = another()[].op()
};
```

e) Object specification source reference name – Object specification query; for example:

```
op() = object {
  ref[] = query ( Server.factory, type()='a_type' )[]
};
```

- Accessing a single sequence element

If an integer literal is written between the brackets following a query or a source name, a single object of the sequence is assigned to the left-hand side. That would be the object indexed by the specified number. For example:

```
getGene() = object {
  g = query( Server.geneFactory, id_gene()='an_id' )[0]
};
```

- Writing a parameter name

Parameter names of an operation are written within mapping using a special syntax, i.e. the operation name, followed by ':', followed by the parameter name.

This guarantees that there is no name conflict with other attribute or source reference names. In the following example, *id* is a parameter of the operation *getById()*:

```
Gene getById(in string id);
getById() = object {
  g = query( Server.geneFactory, id_gene() = getById:id )[0]
};
```

- Referencing a source reference name of the current object specification

Source reference names of an object specification can be used within the mapping specification of other source references within the same object specification. This feature allows the result of one source reference mapping to be used within another. In the following example, *go* is used within the mapping specification of *f*:

```
getPromoter() = object {
  go = query ( Server.genomicObjectFactory,
    (type() = 'promoter') and (id_gene() = id()) ),
  f = query ( Server.dnaFragmentFactory,
    id_dna_fragment() = go.id_dna_fragment() )
};
```

In case of a naming conflict, source reference names of the current object specification take precedence over other names, e.g. source reference names of the current view specification, or attribute names. If the intention is, instead, to use a source reference name of the current view specification or an attribute name, that name should be preceded by the prefix '*this.*'.

- Union

If a target attribute / operation returns a sequence of elements, and its mapping is specified more than once, the *union* of all the elements returned from each mapping is calculated and returned, instead. For example:

```
op()[] = an.op()[];
op()[] = another.op()[];
```

3.3.9 User methods

The mapping language supports user methods inclusion within the view specification. The implementation of any user method called within *mapping* is provided in the *user_method* part of the view definition. User methods are implemented in the same language as the generated target servers (i.e. in Java). Source references as well as standard Java methods can be used in the implementation. When using abbreviated names to refer to classes in the implementation of a user method, it may be necessary to manually include the corresponding Java *import* statements in the generated code.

User method implementation is included *as-is* in the generated target server code. That is, no parsing is performed.

3.3.10 Names and Scope

The rules of the mapping language about valid names and scope are:

- Case-sensitivity

Identifiers are case insensitive; that is, two identifiers that differ only in the case of their characters are considered re-definitions of one another. However, all references to the definition must use the same case as the defining occurrence. For example:

```
view V {
  typedef sequence<string> StringList;

  short op1();
  long op1();         // Error: multiple declaration of op1
  stringList op2();   // Error: inconsistent capitalisation
}
```

- Qualified names

A qualified name in IDL is a name of the form <scoped_name>::<identifier>. Qualified names can be used in the mapping language to declare any source IDL exceptions raised by target IDL operations. If an exception is declared within the scope of an interface (i.e. one of the elements of the <scoped_name> is an interface), that fact has to be explicitly specified in the mapping language in order to allow for correct mapping to Java. This is achieved by inserting the keyword *interface* in parenthesis after the interface name of the <scoped_name>. For example, the following IDL definition

```
module M {
  interface I {
    exception E {};
  };
};
```

corresponds to the following exception declaration in the mapping language

```
M:I(interface):E
```

while the generated Java class for the exception is accessed by

```
M.IPackage.E
```

- Name ambiguities in inheritance

Name ambiguity does not arise in the case of multiple inheritance, because multiple inheritance from views that share one or more identifiers is disallowed. For example:

```
view V1 {
  string op();
}
view V2 {
  short op();
}
view A : V1, V2 {     // Error: V1 and V2 share an operation
name
...
}
```

- Scoping and name resolution

A *module*, a *view*, or an *operation* form scopes. An identifier can only be defined once in a scope. Identifiers can be redefined in nested scopes, but not within the immediate scope of a module or a view. That is, the name of a module, or a view cannot be redefined within the immediate scope of the module, or the view. For example:

```
module M {
  view M {              // Error: M clashes with the module name
    short M();          // Error: M clashes with the view name
  }
}
```

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes, while taking into consideration inheritance relationships among views. In the following example, B will be the first scope to search for the declaration of Vlist. If not found in B, its base view A will then be searched. Similarly, A will be the first scope to search for the declaration of V. If not found in A, the module M will then be searched.

```
module M {
  view V {
  }
  view A {
    typedef sequence<V> VList;
  }
  view B : A {
    VList op();
  }
}
```

3.4 Conclusions

The mediator system described in this thesis includes a mapping language that is used for the implementation of mediator generators. The language is designed for mapping one or more source schemas into a target schema. The design of the mapping language has been driven by the main requirements identified early in this Chapter, and has been based on the design dimensions discussed in Chapter 2. In the following, there is a summary of the main features that identify the mapping language and suggestions for future extensions / improvements, especially with focus on molecular biology data integration.

The language combines declarative and procedural features designed to support a high-level notation, in order to provide easy and concise specification of mappings and to facilitate modification of mapping definitions. The latter is especially important in the field of molecular biology for the support of source and/or target schema evolution due to frequent changes of data structures.

Resolving schematic heterogeneity at the IDL level means that mediator specifiers do not have to learn a new model/language, and no translation between different models and definition languages is necessary. In the future, and as mappings between IDL and UML/XML become fully standardised and associated tools become available, it would be of interest investigating the resolution of schematic heterogeneity at the object modelling level, i.e. at the UML level with/without the use of XML.

The IDL constructs supported by the mapping language suffice to express object models of considerable complexity. However, not all IDL constructs are supported (e.g. IDL *structs* and *value types*). As a result, if an existing CORBA wrapper that contains such constructs needs to be utilised and its unsupported constructs mapped, the mapping would have to be implemented manually. It would be of interest to investigate the inclusion of additional IDL constructs in the mapping language, such as *structs* and *value types*, for improved support of more wrappers, and with regard to efficiency issues.

The object composition model is chosen to support target object derivation. Compared to alternative models, this enables more powerful data integration and customisation of data representation. This feature is especially important in the field of molecular biology due to the different level of detail at which biological objects are modelled in different data sources, and the high degree of interrelationships.

The mapping language supports a convenient way (i.e. using explicit references) of publishing initial CORBA objects. Using the CORBA *Naming Service* would also be a suitable option, which takes advantage of other development work on CORBA, and it would be worth supporting in the future.

The mapping language supports the specification and mapping of fixed parameterised queries. The main advantage is that query results are fully typed with a type controlled by the mapping specifier. An alternative to investigate would be to support type restricted queries, which are also suitable for domain objects represented in IDL and provide fully typed query results.

A mapping to queries supported by CORBA wrappers generated by Persistence™ is provided. Query results are fully typed facilitating client development, and supported queries are sufficiently powerful for the application domain. It would be of interest to investigate providing query mappings to alternative wrappers / wrapper query models, especially the ones used for molecular biology data.

Chapter 4: THE MEDIATOR SYSTEM

The approach examined in this thesis is a mediation system that utilises CORBA in order to support the integration of data from molecular biology data sources that provide access through CORBA and the creation of customised CORBA views of such data sources. The architecture of the developed system combines elements of a mediator-based architecture (as examined in Section 1.3 and depicted in Figure 1-4) with a CORBA-based 3-tier architecture (as examined in Section 1.2.4 and depicted in Figure 1-3). The result is a mediator-based CORBA utilising architecture such as the ones examined in Sections 1.5 and 2.1.

In this Chapter, the architecture of the system is examined in detail including discussion of design and implementation issues of its main components, which are its mediator generator (called IDL View Generator) and the generated mediators.

4.1 Architecture

The architecture of the CORBA mediation system described in this thesis is depicted in Figure 4-1. It is based on the general CORBA mediator architectures of Figure 1-5 and Figure 2-1.

It is assumed that one or more CORBA wrappers are available each one providing access to data stored in a data source. Data sources can, for example, be databases, flat files, or information systems. The goal is to create CORBA servers that customise data representation and/or integrate data (i.e. mediators) providing preferred IDL interfaces.

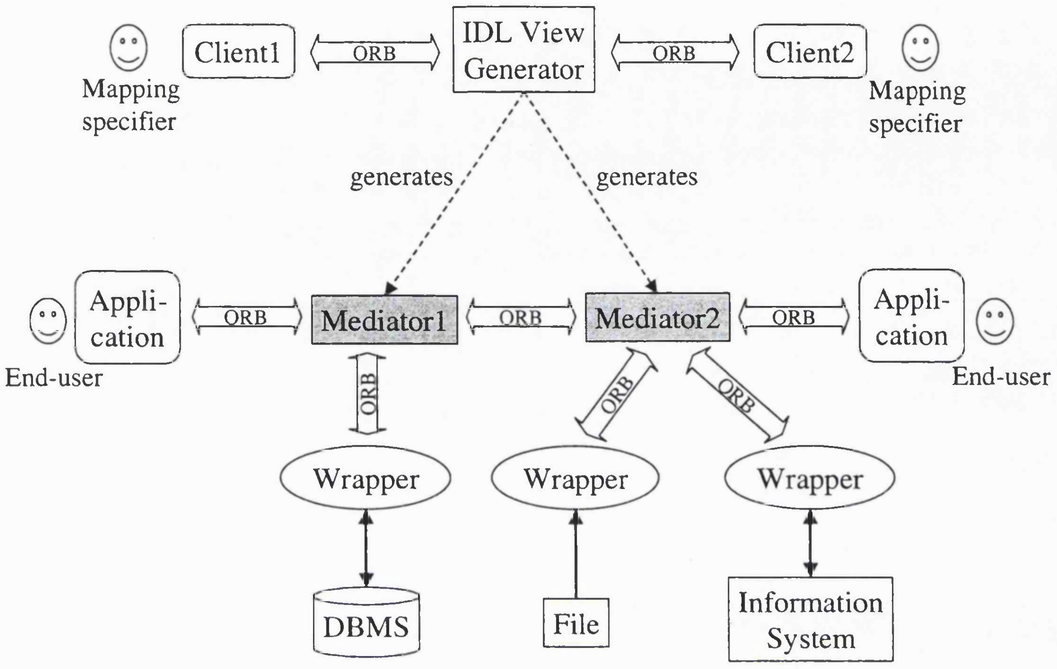

Figure 4-1: System Architecture

Mediators are generated in a semi-automatic way. First, view definitions written in the mapping language need to be provided by the mapping specifiers. These are high-level descriptions of the mapping of mediator IDL definitions to wrapper IDL definitions. In Figure 4-1, ClientX provides view definitions for MediatorX. Based on the view definitions, the IDL View Generator (that is, the system's mediator generator) generates the mediator IDL as well as the mediator implementation.

Applications for end-users can then be developed as clients to generated mediators. Those applications will use the generated IDLs to send requests to mediators. Mediators will convert received IDL requests to wrapper IDL requests.

In the opposite direction, mediators will map results from wrapper IDL representation to their own IDL representation and return them back to applications.

One important point to note is that mediators can, in principle, interact not only with wrappers but also with other mediators, sending requests and receiving results. Mediator1 in Figure 4-1 is an example of a client of another mediator, i.e. Mediator2. That is, there are no special requirements or restrictions on the kind of a CORBA server used as a source by a mediator.

4.2 The IDL View Generator

The IDL View Generator accepts view definitions written in the mapping language. It parses them and generates the mediator IDL and the mediator implementation.

4.2.1 Use of the IDL View Generator

The IDL View Generator can be used either as a stand-alone program from the command line, or as a CORBA server.

In the first case, the program takes as input a filename with the view definitions that describe a mapping, and generates the mediator files. The command line syntax is:

```
java IdlViewG <ViewDefsFile>
```

In the latter case, the IDL View Generator becomes fully integrated within CORBA. This has the advantage that any mapping specifier can easily use it over the Internet. The IDL definition provided by the IDL View Generator is quite simple and shown below:

```
module ivgServer {
  typedef sequence <octet> FileFlow;
  interface ivgManager {
    FileFlow generateZipFile(in string viewDefs);
  }
}
```

There is only one interface defined (i.e. *ivgManager*), which provides one operation (i.e. *generateZipFile*). This operation takes as input the view definitions that describe a mapping and returns a zip-compressed file. The returned file is a tar-collection of the mediator files.

4.2.2 Mediator generated files

The generated mediator, which is implemented in the Java programming language, is a collection of files that include:

- One file with the IDL definition of the generated CORBA server. Each view in the view definitions is mapped to an interface with the same name in the mediator IDL definition. The file is named *<ModuleName>.idl*, where *<ModuleName>* is the name of the module given within the view definitions, as described in Section 3.3.3.
- One file that defines the Java class of the server including the *main()* method. The file is named *<ServerName>.java*, where *<ServerName>* is the name of the server given in the server declaration part of the view definitions, as described in Section 3.3.2.
- A number of files implementing the interfaces defined within the generated IDL. According to the IDL to Java mapping specification [OMG 98a] each IDL interface is mapped to one Java class. One file is generated for each interface, and is named after that interface as *<InterfaceName>Impl.java*.

Each of those files defines the Java class that corresponds to the interface and includes the implementation of all required methods.

In order to get a complete CORBA server, the server class file and the interface implementation files will need to be compiled together with the skeletons, which are generated automatically by the ORB after compiling the provided IDL.

4.2.3 Implementation

The IDL View Generator is implemented in the Java programming language.

The JavaCC (Java Compiler Compiler) parser generator program was used for the implementation. JavaCC is a freely available program that was originally developed by SUN [Sun] and is now distributed and supported by Metamata [Metamata]. A parser generator is a tool that reads a high-level grammar specification and converts it into a program that can recognise matches to the grammar. This allows language developers to concentrate on the grammar and not worry about the implementation details of parsing, thus making the development of a new language easier and faster, as well as providing better documentation and facilitating maintenance of the code. JavaCC generates pure Java code.

4.3 Mediators

Mediators are CORBA servers that customise data representation and/or integrate data coming from available CORBA wrapped data sources. They provide preferred IDL interfaces that can then be used by prospective applications.

4.3.1 Object composition and the adapter design pattern

Target objects are derived from source objects following the *object composition* model (see Section 2.3 and Section 3.2.2). That is, composite objects need to provide an interface to the outside world, while internally mapping the elements of the supported interface into the elements of one or more interfaces of included source objects.

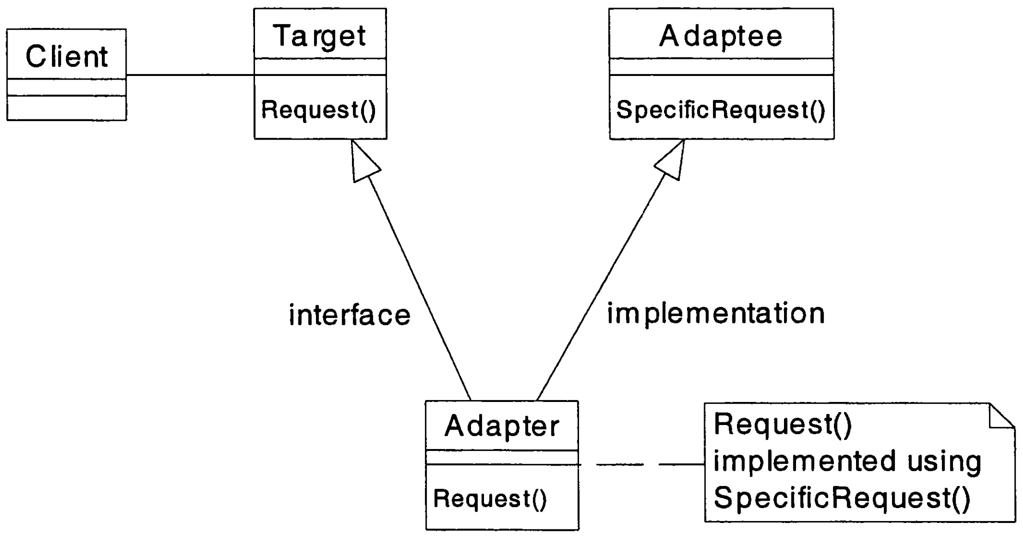

Figure 4-2: UML Model of Class Adapter (taken from [Gamma 95])

This is similar to the object-oriented design problem of converting the interface of a class into another interface that clients expect. Frequently occurring design problems and their solutions in object-oriented software engineering are called design patterns [Gamma 95]. The *adapter* design pattern focuses on the problem mentioned above. It allows conversion of an existing interface (defined by *adaptee*) into another one (defined by *target*) that *clients* expect [Gamma 95]. An *adapter* is used to adapt the interface of adaptee to the target interface. It receives requests from clients, which it translates into calls to adaptee operations that carry out the requests.

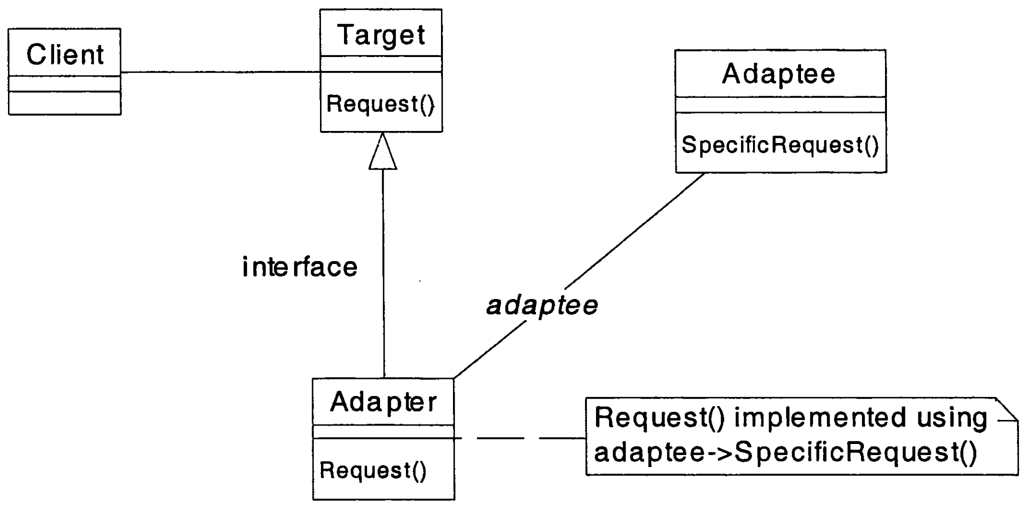

Figure 4-3: UML Model of Object Adapter (taken from [Gamma 95])

An adapter can be implemented using multiple inheritance (called class adapter, see Figure 4-2), or object composition (called object adapter, see Figure 4-3). In the first case, the adapter inherits the interface definition from target and the implementation from adaptee. When it receives a *Request()*, it calls the corresponding *SpecificRequest()* operation of adaptee. In the latter case, the adapter inherits the interface definition from target and holds a reference to the adaptee. When it receives a *Request()*, it uses the *adaptee* reference to call the corresponding *SpecificRequest()* operation of adaptee.

The generated mediators are implemented using object adapters as opposed to class adapters, since the first ones provide greater flexibility in composing a target object out of many source objects (i.e. adaptee objects) of different interfaces and allow the source and target interface hierarchies to be kept separate. Object adapters need to keep references to the corresponding source objects. These are the source references of the mapping language described in Section 3.3.7. They are assigned in the constructor method of an adapter.

4.3.2 Interface implementation

CORBA provides two approaches to interface implementation. The *ImplBase* approach, which is based on class adapters (explained in Section 4.3.1), and the *TIE* approach, which is based on object adapters (explained in Section 4.3.1).

In the *ImplBase* approach, for each interface, the ORB generates an abstract Java class named after the interface as *_<InterfaceName>ImplBase* (see Figure 4-4). The corresponding interface implementation class should inherit from that *ImplBase* class. The *Server* class instantiates an object of an interface implementation class within its *main()* method. This approach requires the user-defined implementation class to extend an ORB generated base class. That limits the flexibility of implementation classes and eliminates the possibility of reusing existing implementations for languages that do not support multiple inheritance, e.g. Java.

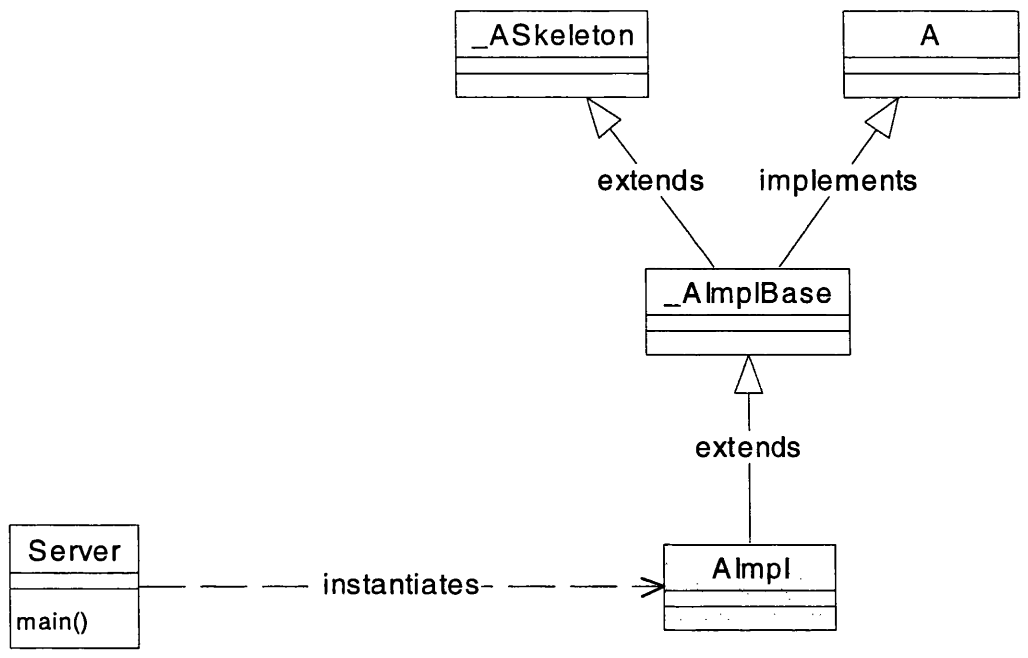

Figure 4-4: The ImplBase Approach for Interface A

In the *TIE* approach, for each interface, the ORB generates a Java class named after the interface as *_tie_<InterfaceName>* (see Figure 4-5). It also generates an interface, named *_<InterfaceName>Operations*, which defines the operations to be implemented by the interface implementation class. The corresponding interface implementation class should inherit the definition of that *_<InterfaceName>Operations* interface. The *Server* class instantiates a TIE object passing an object of the corresponding interface implementation class to the constructor. As a result, a TIE object is created that delegates incoming operations to the methods of the corresponding interface implementation object. This approach requires the creation of an additional object for each implementation object instantiated in a server. That increases memory requirements especially when a large number of implementation objects are created in the server. In addition, client invocations are delegated by TIE objects to implementation objects. That involves an additional Java method invocation for each incoming request.

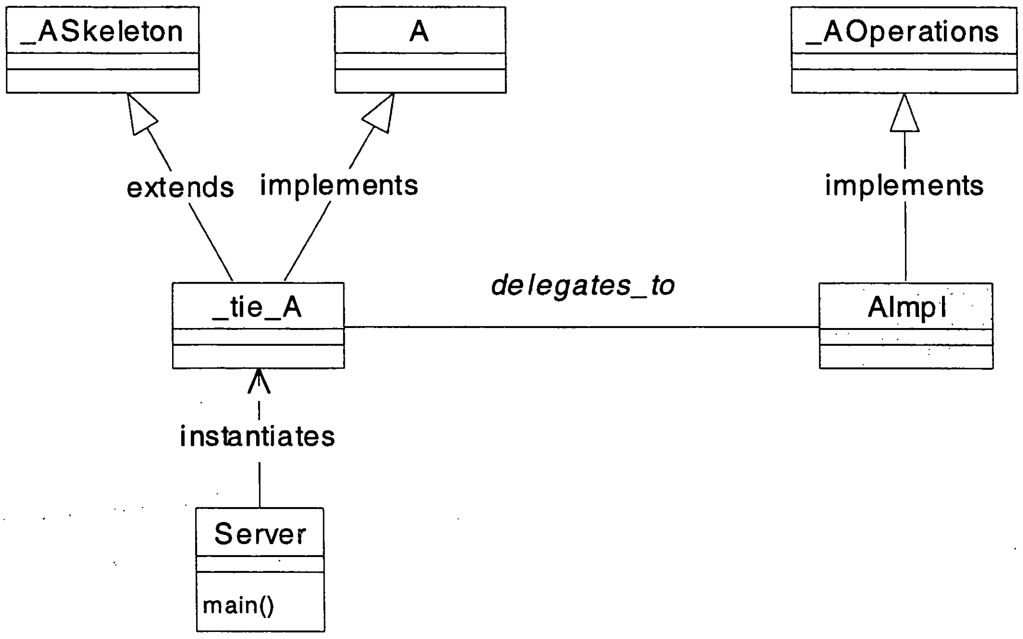

Figure 4-5: The TIE Approach for Interface A

The generated mediators are implemented in the Java programming language that does not support multiple inheritance. Therefore, the TIE approach is used to implement interfaces. This allows implementation to be inherited from other interfaces.

4.3.3 Inheritance implementation

An example is used here to discuss the implementation of inheritance within mediators. Three views are defined, namely *GenomicObject*, *Promoter* and *Terminator*. The *Promoter* and *Terminator* views inherit both the definition and the implementation of the *GenomicObject* view, as shown in the following code (details not necessary for the purposes of this example are omitted).

```
view GenomicObject {
      definition {
         long posFirst();
         long posLast();
         string dnaSeq();
      }
...}
view Promoter : GenomicObject (extends) {...}
view Terminator : GenomicObject (extends) {...}
```

As discussed in Section 4.2.2 each view is mapped to an interface in the generated mediator IDL definition, i.e.

```
interface GenomicObject {...}
interface Promoter : GenomicObject {...}
interface Terminator : GenomicObject {...}
```

Each interface is mapped to a Java class in the generated interface implementations. The use of the TIE approach to interface implementation (see Section 4.3.2) means that interface implementation classes inherit the definitions of (i.e. Java *implements* inheritance) their corresponding CORBA generated

_<*InterfaceName*>*Operations* interfaces, implementing the methods included in those definitions. This is depicted (for the current example) in Figure 4-6, where the interface implementation classes *GenomicObjectImpl*, *PromoterImpl* and *TerminatorImpl* inherit respectively the definitions of *_GenomicObjectOperations*, *_PromoterOperations* and *_TerminatoOperations*.

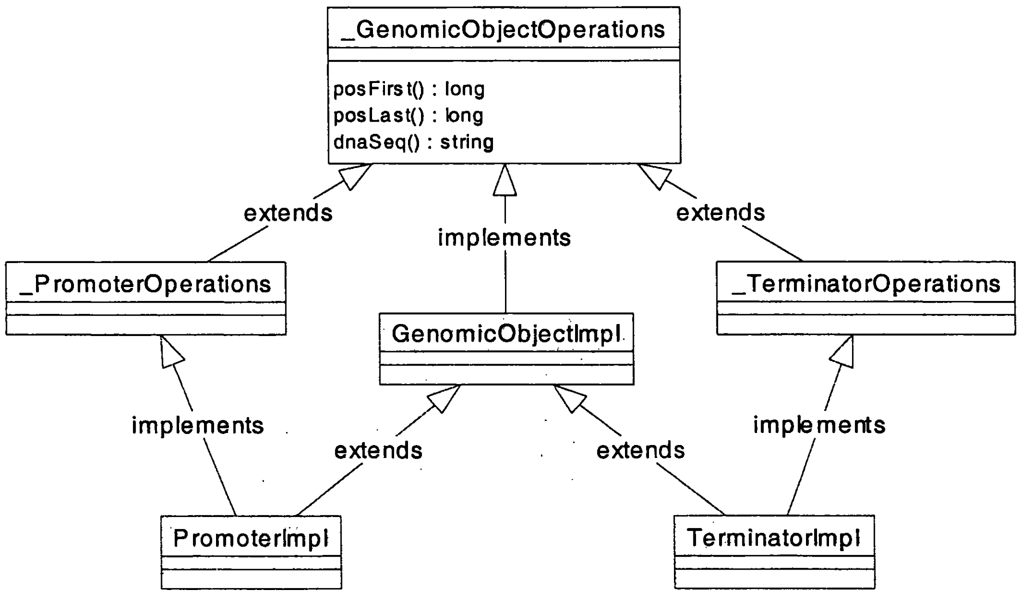

Figure 4-6: UML Model of Inheritance Implementation

Additionally, if a view inherits the implementation of another view (i.e. using the mapping language *extends* inheritance keyword), its respective interface implementation class will also inherit the implementation of (i.e. Java *extends* inheritance) the superview's interface implementation class. This is depicted in

Figure 4-6, where the interface implementation classes *PromoterImpl* and *TerminatorImpl* inherit the implementation of *GenomicObjectImpl.* The corresponding generated Java code is:

```
class GenomicObjectImpl implements _GenomicObjectOperations
{...}
class PromoterImpl extends GenomicObjectImpl implements
_PromoterOperations {...}
class TerminatorImpl extends GenomicObjectImpl implements
_TerminatorOperations {...}
```

As discussed in Section 3.3.4, in the case of multiple view inheritance, a view is allowed to inherit the definitions of many views but the implementation of at most one view (i.e. the one for which the *extends* keyword is used). This is because in Java, multiple implementation inheritance is not supported. As an example, take a view *A* that inherits the definitions of views *B*, *C*, and *D*, as well as the implementation of view *C*, i.e.

```
view A : B, C (extends), D {...}
```

The Java implementation class of *A* will inherit the definitions of all the *_<InheritedInterfaceName>Operations* interfaces. It will also inherit the implementation of class *C*, i.e.

```
class AImpl extends CImpl implements _AOperations {...}
```

where,

```
public interface _AOperations extends _BOperations,
_COperations, _DOperations {...}
```

4.4 Conclusions

The architecture of the system described in this thesis is a mediator-based one that utilises CORBA for communication between mediators and wrappers, and between mediators and end-user applications. It is flexible, modular and extensible in that mediators can, in principle, communicate not only with wrappers but also with other mediators. That is, there are no special requirements or restrictions on the kind of CORBA server used as a source by a mediator. As a consequence, mediators could be utilised as modules using which other more complex or more specialised mediators would be constructed.

One of the main components of the system is the IDL View Generator that generates mediators based on view definitions written in the mapping language. The IDL View Generator is provided as a CORBA server, thus becoming fully integrated within CORBA. The advantage is that mapping specifiers can easily access and use it over the Internet.

The other main component is the generated mediator in each case. Generated mediators are implemented using object adapters. This approach enables flexible composition of a target object out of many source objects of different interfaces, and allows the source and target interface hierarchies to be kept separate. In addition, and consistent to the design of the mapping language, mediators implement inheritance in a way that supports multiple inheritance for view definitions, and single inheritance for view implementation.

Chapter 5: APPLICATION IN BACTERIAL GENOMES

An example from the domain of molecular biology and, in particular, bacterial genome data is used in this chapter to demonstrate the use of the system. It also proposes a software development process in achieving the goal of data integration.

Two data sources have been used for obtaining genome data of the *Bacillus subtilis* bacterium. One is the SubtiList database ([Moszer 95], [Moszer 98]), which is a relational database for the Bacillus subtilis genome. A CORBA wrapper for this database has been generated using the Persistence™ software. The other is the EMBL nucleotide sequence database [Stoesser 02], a database for nucleotide sequence data and related biological information for many organisms, including Bacillus subtilis. The available CORBA server for this database [Wang 00] has been used to access the data. The data relevant to the application served by those two CORBA wrappers and the provided IDL interfaces are described in Section 5.1.

Having the two CORBA wrappers, the aim has been to develop a mediator that integrates Bacillus subtilis data from the wrappers and provides the integrated data through an alternative and preferred IDL definition. In order to achieve that, an object model of bacterial genome data has been developed (see Section 5.2), based on which, suitable IDL interfaces for the mediator CORBA server have been defined (see Section 5.3). The IDL View Generator is then used to generate the required mediator. This involves the definition of the mapping of mediator to wrapper IDL interfaces (see Section 5.4), which is given as input to the IDL View Generator. Based on the view definitions, the IDL View Generator generates a CORBA server with the specified IDL interfaces. In order to test the generated mediator, a simple textual client application has been developed that uses the IDL interfaces of the mediator CORBA server and retrieves the integrated data (see Section 5.5).

5.1 Source CORBA Servers

Two independently developed CORBA servers have been used for the purposes of this application.

5.1.1 SubtiList

The SubtiList database is a relational database for the genome of Bacillus subtilis (reference strain 168), the model organism of sporulating Gram-positive bacteria ([Moszer 98], [Moszer 95]). It provides the complete DNA sequence with location information within the single chromosome of the organism. It also links the DNA sequence to relevant annotation and analysis data. This includes characterisation of protein coding genes and the derived protein sequences, characterisation of RNA genes, prediction of possible operons, annotation of protein similarities, functional classification of protein gene products, nucleotide base composition and oligonucleotide bias data, repeated sequences and codon usage data. The data originated mainly from the B. subtilis genome sequencing project [Kunst 97], supplemented with information from the B. subtilis entries present in the EMBL/GenBank/DDBJ databanks [Stoesser 02], as well as observations either published in international journals or communicated directly by individual researchers. The current data available from the SubtiList Web server [Moszer 98] were released on April 26, 2001 (data release R16.1).

A subset of the complete SubtiList database was made available, installed locally and used for the purposes of the application. It includes the complete DNA sequence with location information, characterisation of protein coding and RNA genes, functional classification of protein gene products, as well as codon usage data. The latest data stored in that database and used for the application were released on November 20, 1997 (data release 14.2). This is the same release for B. subtilis data as the one used from the EMBL/GenBank/DDBJ databanks (see Section 5.1.2).

A CORBA wrapper for the locally installed SubtiList database has been generated using the Persistence™ software. Persistence™ generates a C++ class library to access an underlying relational database providing an object model that closely reflects the underlying database schema. In addition, it generates a CORBA server for the IONA Orbix ORB that uses the class library to access the data. The kind of IDL interfaces generated by Persistence™ and their mapping to the database schema are outlined in Section 2.5.

The complete CORBA server IDL definitions generated by Persistence™ for the SubtiList database are provided in Appendix B. For each project, Persistence™ generates three IDL definition files: *PS.idl*, *PSAdmin.idl* and *<project_name>.idl.* The PS module contains declarations for basic capabilities that are commonly used by CORBA server and client programs. It defines a transaction interface, enumerations, specialised attribute types, and exception types. The PSAdmin module defines advanced capabilities that are used to configure a CORBA server. The classes of the PSAdmin module are used to manage the server, database tables, database connections, the cache, and the event log. Most client programs would not need these operations, and it is possible to "hide" the interface to the PSAdmin module such that a client cannot easily invoke the methods. The <project_name> module defines the project specific interfaces. For each class *<class_name>* specified in the object model during the generation process, it defines an interface *<class_name>* containing the relevant attributes and relationships, an associated factory-type interface *<class_name>_Factory* that supports queries, and an associated collection type *<class_name>_Cltn* as an IDL sequence of the interface.

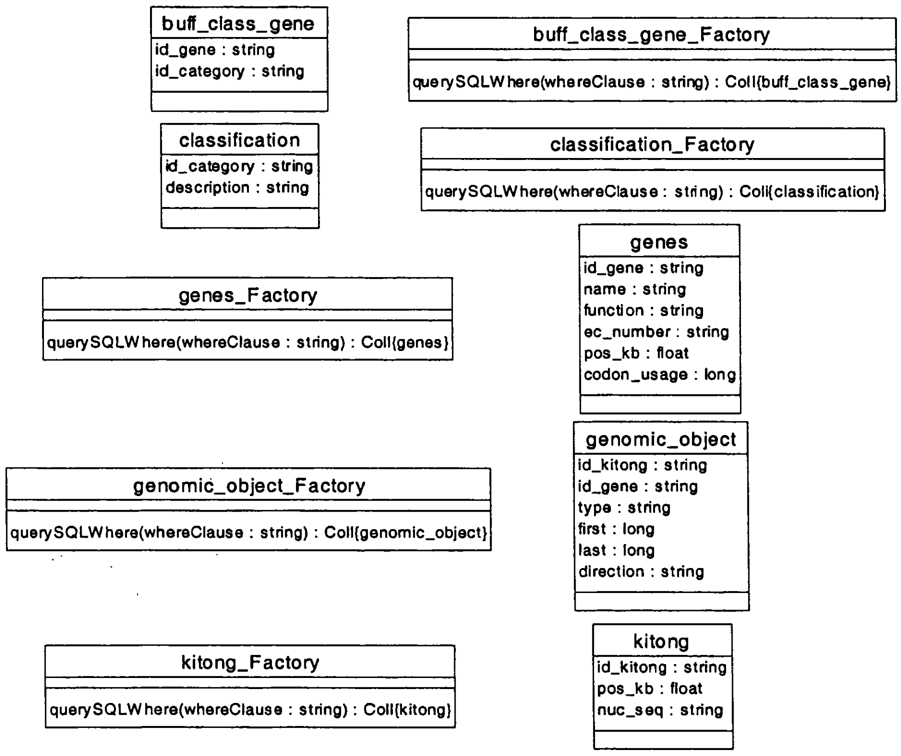

Figure 5-1: Part of SubtiList CORBA Server IDL in UML notation

A part of the SubtiList IDL definitions used for the purposes of the application is depicted in Figure 5-1. They represent DNA sequence and location information for protein coding genes and other functionally significant regions of DNA sequences (e.g. promoters, terminators), as well as information on functional classification of protein gene products and codon usage. Five main interfaces are defined: *genes*, *genomic_object*, *kitong* (described later on), *buff_class_gene* (described later on) and *classification*, as well as five factory-type interfaces for querying purposes: *genes_Factory*, *genomic_object_Factory*, *kitong_Factory*, *buff_class_gene_Factory* and *classification_Factory*.

- The *genes* interface represents biological genes. It contains naming/identification information (i.e. *id_gene*, *name*, *ec_number*), functional description (i.e. *function*), position within the chromosome (i.e. *pos_kb*), and codon usage (i.e. *codon_usage*) information.
- The *genomic_object* interface represents various functionally significant regions of DNA sequences (also known as features), such as promoters, terminators, -35 signals and -10 signals. Their type is distinguished by the value of the attribute *type*, while their start and end nucleotide base positions within a DNA fragment are given by *first* and *last,* respectively. Their nucleotide sequence direction is specified by the attribute *direction*: a "+" value means left-to-right, and a "-" value means right-to-left.
- The *kitong* interface represents fragments of DNA sequences. It contains the relevant DNA sequence string (i.e. *nuc_seq*), and the starting position of the kitong within the chromosome (i.e. *pos_kb*). The name *kitong* is the exact name used in the database schema, and it is reflected unchanged in the generated IDL. Perhaps, it was devised as a close anagram of the word contig, which means a DNA sequence fragment!
- The *buff_class_gene* and *classification* interfaces represent information on functional classification of protein gene products. The *buff_class_gene* classifies genes into a number (i.e. total 52) of categories, that is, it associates each gene with one functional class. The name *buff_class_gene* is the exact name used in the database schema, and it is reflected unchanged in the generated IDL. The *classification* provides a description of each functional category [Moszer 98].

- The *genes_Factory*, *genomic_object_Factory*, *kitong_Factory*, *buff_class_gene_Factory* and *classification_Factory* interfaces represent factory-type objects as described in Section 2.5. They all provide the query operation *querySQLWhere,* which takes a string as an input parameter. The string corresponds to the where-clause of an SQL query. They evaluate the query and return a collection (i.e. an IDL sequence) of the corresponding objects.

The CORBA wrapper generated by Persistence™ has proved useful and manageable for a small database such as SubtiList. A few adaptations had to be made to the generated IDL and server program in order to make the existence of the underlying SubtiList database transparent to client programs and to support read-only access to the data. They included removing operations that controlled database connections, and transactions, removing operations of factory-type interfaces that removed/cleared their corresponding interface objects and modifying the server program. Apart from these modifications made solely for the purposes of restricted database access, the generated IDL definitions did not change in any other way that would facilitate integration.

5.1.2 EMBL

The EMBL nucleotide sequence database [Stoesser 02] maintained at the European Bioinformatics Institute (EBI) is a database for nucleotide sequence data and related biological information, such as description, taxonomic classification, citations, biological features with location information and feature qualifiers. It is produced in collaboration with the GenBank (NCBI, Bethesda, USA) [Benson 02] and DDBJ (CIB, Mishima, Japan) [Tateno 02] databanks. The three databanks exchange data that they receive from genome sequencing centres, individual scientists and patent offices. Sequence data are assigned accession numbers that uniquely identify them. The complete nucleotide sequence of the genome of Bacillus subtilis and associated annotation are stored in the database and assigned the accession number *AL009126*. The complete genome data used for the purposes of this application were created and last updated in November 1997, i.e. they are of the same data release as the ones stored in the locally installed SubtiList database (see Section 5.1.1). More recent updates of specific parts of the genome are also available. However, at the time of the development of the application those were not available through the complete genome data.

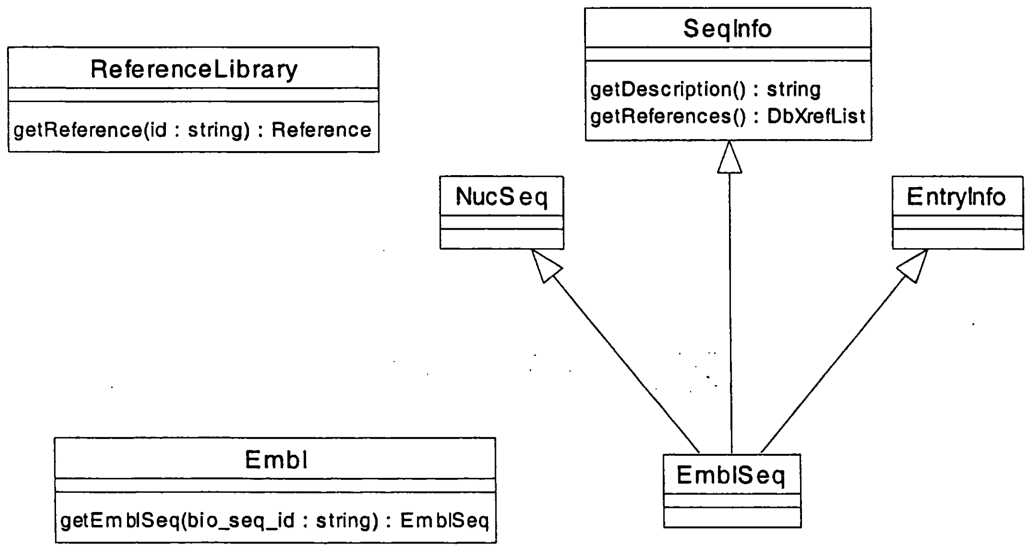
ReferenceLibrary
getReference(id : string) : Reference
SeqInfo
getDescription() : string
getReferences() : DbXrefList
NucSeq
EntryInfo
Embl
getEmblSeq(bio_seq_id : string) : EmblSeq
EmblSeq

Figure 5-2: Part of EMBL CORBA Server IDL in UML notation

The available EMBL CORBA server [Wang 00] has been used to access the B. subtilis data that is stored in the EMBL database. A part of the EMBL CORBA server IDL definitions used for the purposes of the application is depicted in

Figure 5-2. They represent information on biological sequences including description and bibliographic references.

- The *Embl* interface is the entry point for obtaining nucleotide sequence data. It defines the operation *getEmblSeq()* that retrieves a nucleotide sequence given its accession number.
- The *EmblSeq* interface inherits from the *NucSeq*, the *SeqInfo* and the *EntryInfo* interfaces.
- The *NucSeq* interface contains the nucleotide sequence and provides access to feature (i.e. sequence annotation), location and originating organism information.
- The *SeqInfo* interface has information associated with a sequence, such as description, comments, cross references to other databases and cross references to bibliographic references provided by the publication part of the EMBL CORBA server.
- The *EntryInfo* interface provides information specific to the EMBL database entry.
- The *ReferenceLibrary* interface contains bibliographic references associated with sequences. It defines the operation *getReference()* that retrieves a reference given its identifier, and *getReferences()* that retrieves bibliographic references of a sequence given its accession number.

5.2 Object Model

An object model of bacterial genomes that is the basis for the definition of the mediator CORBA server IDL (to be discussed in Section 5.3) has been developed. The mediator integrates Bacillus subtilis data from the two independently developed wrappers described in Section 5.1. The part of the object model used for the purposes of the current application is presented and discussed in detail in this Section.

An object model of bacterial genomes has been developed using the Unified Modelling Language (UML). It is provided together with an accompanying glossary of terms in Appendix C. Its purpose is to form the basis for the definition of IDL interfaces for the mediator CORBA server (see Section 5.3). It focuses on the representation of structural and functional information primarily at the DNA and RNA sequence levels, linking that to some high level representation of structural information at the protein level, as well. More specifically, it tries to capture, from a biological point of view, the different kinds of genomic objects coded by nucleotide sequences, their location, their nucleotide sequence, as well as how they relate to each other and to the resulting proteins.

A simplified part of the overall bacterial genomes object model is depicted in Figure 5-3. This shows the part of the model used for the purposes of the current application. It includes classes that represent location and sequence information for a number of genomic objects such as promoters, terminators, -35 and -10 signals. Genomic objects are associated to genes. Genes include identification and naming information, links to bibliographic references, and codon usage analysis results. Suitable classes for the functional classification of protein coding genes are also identified and linked to genes. A more detailed description of the classes included in the model is given below:

- The *GenomicObject* class represents functionally significant regions of nucleotide sequences. These can be either regulatory signals, or protein coding sequences. Genomic objects have a location on a nucleotide sequence, for example, a chromosome.
- The *NucSeq* class represents a nucleotide sequence associated with a genomic object. It includes the actual nucleotide sequence string (i.e. *seq_string*) and a boolean type attribute that encodes whether the nucleotide sequence direction

is left-to-right, or right-to-left (i.e. *direction_is_left_to_right*). Any nucleotide sequence can be represented by this class, an example being a chromosome.

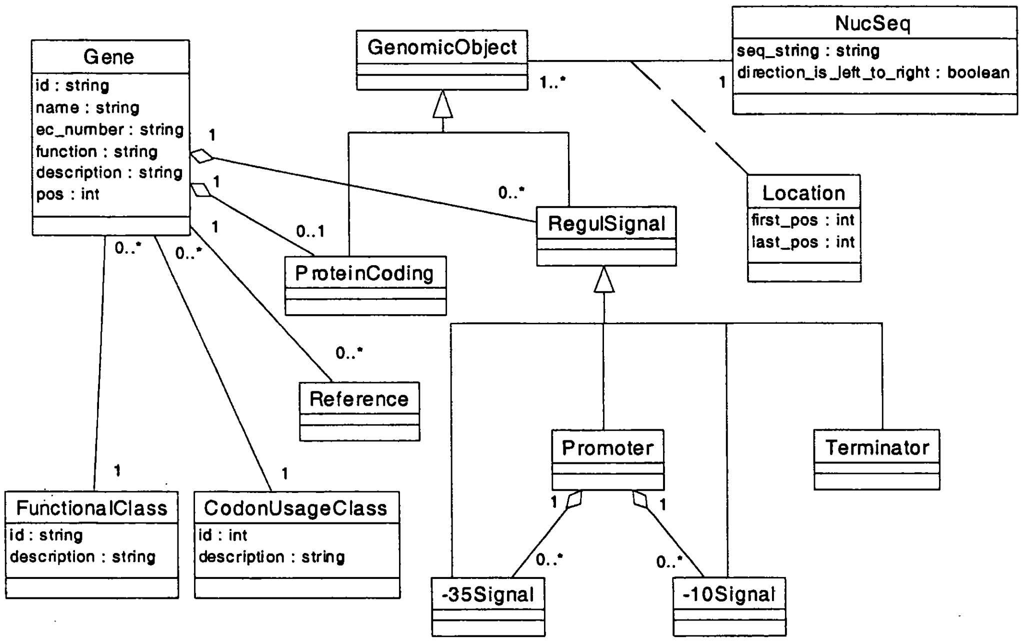

Figure 5-3: Part of Bacterial Genomes Model in UML notation

- The *Location* class represents the positions of the first (i.e. *first_pos*) and last (i.e. *last_pos*) nucleotides of a genomic object within a nucleotide sequence.
- The *RegulSignal* class represents DNA sequences that act as signals, such as promoters, terminators, -35 and -10 signals.
- The *Promoter* class represents promoters. A promoter is the region of DNA where RNA polymerase initiates transcription. It contains the startpoint where

transcription begins, the -35 and -10 signals, and is usually positioned before the protein coding sequence(s).

- The *-35Signal* class represents -35 signals. A -35 signal is the recognition domain of a promoter.
- The *-10Signal* class represents -10 signals. A -10 signal is the unwinding domain of a promoter.
- The *Terminator* class represents terminators. A terminator is the region of DNA where transcription ends and is usually positioned after the protein coding sequence(s).
- The *Gene* class represents genes, which are regions of DNA sequence having some specific functionality, and containing DNA sequences that encode for proteins/RNAs, as well as any associated regulatory signals. A gene includes identification/naming information (i.e. *id*, *name*, *ec_number*), the approximate location of the gene within a nucleotide sequence (i.e. *pos*), and general information related to the gene and its function (i.e. *function*, *description*). It may also include links to one or more relevant bibliographic references, and, if it is a protein coding gene, it can be classified into a functional class, and/or a codon usage class.
- The *FunctionalClass* class represents suitable classes into which protein coding genes are classified based on their function. Each functional class is identified by a structured number (i.e. *id*) and briefly described (i.e. *description*).
- The *CodonUsageClass* class represents classes identified and used for the classification of genes based on codon usage analysis results. Each codon usage class is identified by a number (i.e. *id*) and briefly described (i.e. *description*).

- The *Reference* class represents bibliographic reference information that is related to genes.
- The *ProteinCoding* class represents DNA sequences that encode proteins.

5.3 Mediator CORBA Server IDL

Based on the object model discussed in Section 5.2, an IDL definition that will be used for the mediator CORBA server has been developed and is depicted in Figure 5-4. The differences between the object model and the resulting IDL definition are discussed in this Section.

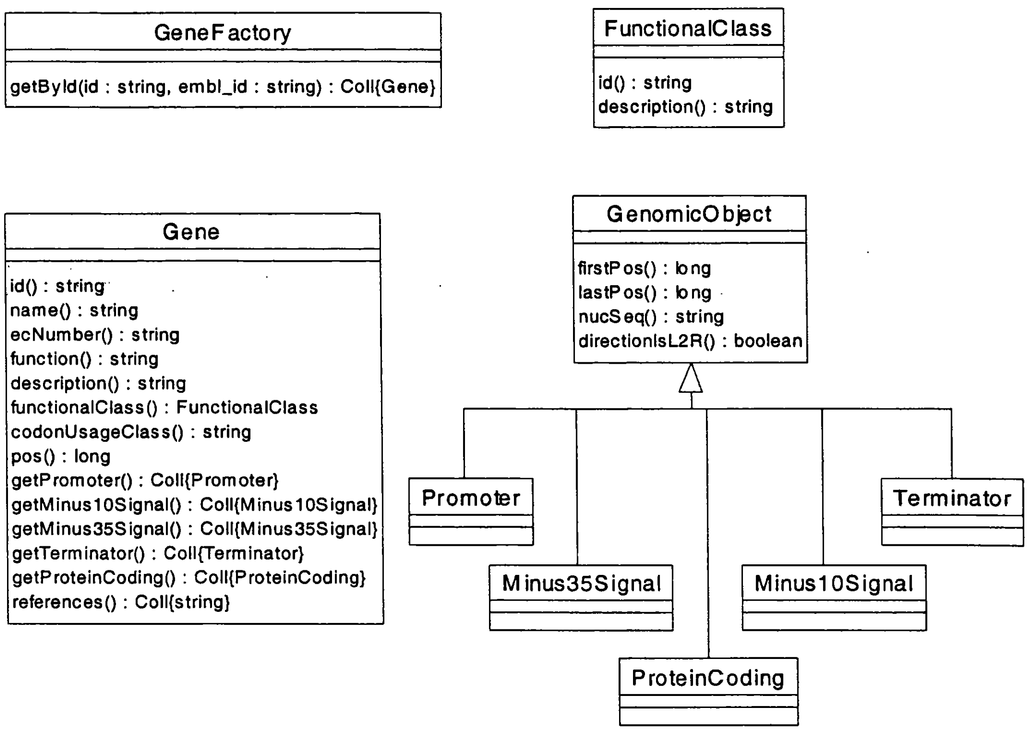
GeneFactory
getById(id : string, embl_id : string) : Coll{Gene}
FunctionalClass
id() : string
description() : string
Gene
id() : string
name() : string
ecNumber() : string
function() : string
description() : string
functionalClass() : FunctionalClass
codonUsageClass() : string
pos() : long
getPromoter() : Coll{Promoter}
getMinus10Signal() : Coll{Minus10Signal}
getMinus35Signal() : Coll{Minus35Signal}
getTerminator() : Coll{Terminator}
getProteinCoding() : Coll{ProteinCoding}
references() : Coll{string}
GenomicObject
firstPos() : long
lastPos() : long
nucSeq() : string
directionIsL2R() : boolean
Promoter
Terminator
Minus35Signal
Minus10Signal
ProteinCoding

Figure 5-4: Mediator IDL in UML notation

The *GeneFactory* interface has been introduced in the mediator IDL definition. Its role being to act as a starting point of the CORBA server, from which *Gene* objects can subsequently be accessed.

Location and nucleotide sequence information for genomic objects have been included within the *GenomicObject* class for simplicity. As a result the *NucSeq* and *Location* classes are not represented within the mediator IDL definition.

The *RegulSignal* class is not included within the mediator IDL definition for simplicity. As a result, operations to access any regulatory signal genomic objects from their associated gene objects are directly included in the *Gene* class (i.e. *getPromoter(), getMinus10Signal(), getMinus35Signal(), getTerminator()*).

Aggregation is a special form of association. It implies a part-whole relationship. This means that usually a part exists when and as long as the whole exists [Rumbaugh 91]. The aggregation relationship between the *Gene* class and the *RegulSignal* and *ProteinCoding* classes of the bacterial genomes model of Figure 5-3 is supported in the mediator IDL definition, since *ProteinCoding*, *Promoter*, *Terminator*, *Minus10Signal*, and *Minus35Signal* objects can only be accessed through their aggregate *Gene* object. In the case of the aggregation relationship between the *Promoter* class and the *Minus10Signal* and *Minus35Signal* classes, the mediator IDL definition does not support any such "constraint", the reason being to keep the IDL definition leaner and avoid extra network calls when accessing information on -10 and -35 signals.

Genes are classified into codon usage classes according to the results of codon usage analysis. Only three such classes are identified in the SubtiList database [Moszer 98]:

- *Class 1* contains most of the genes except genes for classes 2 and 3,
- *Class 2* contains genes that are highly expressed during exponential growth, and

- *Class 3* contains genes of unknown function corresponding to A+T-rich islands. Their function is often associated with bacteriophages or the cell envelope.

Due to the small number of classes and in order to keep the IDL definition leaner and simpler, the *CodonUsageClass* class is not included within the mediator IDL definition. Instead, the *codonUsage()* operation in the *Gene* class provides the codon usage class associated with the specific gene.

The *Reference* class is not included within the mediator IDL definition for simplicity. Instead, the *references()* operation in the *Gene* class provides the list of bibliographic references that are relevant to the specific gene as a list of strings.

5.4 View Definition

Having the two CORBA wrappers and the mediator CORBA server IDL definition, the mapping of mediator to wrapper IDL interfaces needs to be defined. This is given as input to the IDL View Generator in order to generate the required mediator, which is a CORBA server with IDL interfaces as specified within the view definitions. The complete generated mediator code is provided in Appendix D. The complete view definitions, which include the mediator CORBA server IDL definition and the mapping, are presented and discussed in detail in this Section. The purpose of this is to give the reader a complete picture of the view definition code required by the mediator system. In order to aid the reader further, explanations/comments are included throughout the view definitions code.

```
/*
The necessary IDL definitions for the EMBL (i.e. types.idl,
nsdb.idl) and the SubtiList (i.e. PS.idl) CORBA servers to be
included within the mediator IDL.
*/
```

```
#include "/embl/idl/types.idl"

#include "/embl/idl/nsdb.idl"

#include "/sldb/idl/PS.idl"


/*

Declaration of the generated CORBA server class name (i.e.
Server), and definition of available wrapper factories (i.e.
classgeneF, classF, geneF, geneobjF, and dnafragF for the
SubtiList, and emblF, ctrlF, and libF for the EMBL CORBA
wrappers).

*/

server Server {

  // Source factories.

  factory classgeneF {

    ftype SLdb.buff_class_gene_Factory;

    ior "file:/sldb/ior/cg_Fact.ior";

  }

  factory classF {

    ftype SLdb.classification_Factory;

    ior "file:/sldb/ior/c_Fact.ref";

  }

  factory geneF {

    ftype SLdb.genes_Factory;

    ior "file:/sldb/ior/g_Fact.ior"

  }

  factory genobjF {

    ftype SLdb.genomic_object_Factory;

    ior "file:/sldb/ior/go_Fact.ior"

  }
  factory dnafragF {
    ftype SLdb.kitong_Factory;
    ior "file:/sldb/ior/df_Fact.ior"
  }
  factory emblF {
    ftype nsdb.Embl;
    ior "http://corba.ebi.ac.uk/EMBL/IOR/Embl.ior"
  }
  factory ctrlF {
    ftype meta.Controled;
    ior "http://corba.ebi.ac.uk/EMBL/IOR/Meta.ior"
  }
  factory libF {
    ftype bibliography.ReferenceLibrary;
    ior "http://corba.ebi.ac.uk/EMBL/IOR/Reference.ior"
  }
}

module BacterialGenome {
  // Forward declaration of views.
  view GeneFactory;
  view GenomicObject;
  view FunctionalClass;
  view Gene;
  view Promoter;
  view Minus10Signal;
  view Minus35Signal;
  view Terminator;
view ProteinCoding;


/*
The GeneFactory view acts as a starting point, from which
Gene objects can subsequently be accessed.
*/
view GeneFactory {
   /*
   A GeneFactory factory-type object will be published by the
   mediator CORBA server. Its IOR will be written into the
   file, the URL of which is specified after the location
   keyword.
   */
   location "file:/tmp/BactGen.GeneFactory.ior"
   definition {
      /*
      The operation getById() raises wrapper IDL exceptions
      (i.e. type::NoResult and
      nsdb::Embl(interface)::Superceded for the EMBL, and
      PS::ServerError for the SubtiList CORBA wrappers), in
      order to forward those exceptions to clients of the
      mediator CORBA server. Note the insertion of the keyword
      interface in nsdb::Embl(interface)::Superceded, to point
      out the fact that the exception Superceded is declared
      within the scope of the interface Embl, and allow for
      correct mapping to Java (as explained in Section
      3.3.10).
      */
      Gene getById(in string id, in string embl_id)
         raises (type::NoResult,
                 nsdb::Embl(interface)::Superceded,
            PS::ServerError);
}
mapping {
   /*
```

The operation getById() returns a new Gene object, assigning its source references g, go, and es:

- *variable g is assigned the SLdb.genes object the id of which (i.e. id_gene()) is the same as the id given as an input parameter in the current operation (special syntax for the parameter is: getById:id). The factory queried (i.e. geneF) is a SubtiList factory, and the query will be sent to the Persistence™ generated CORBA wrapper for SubtiList. In the supported mapping to Persistence™ generated CORBA wrappers, queries always return sequences of objects (as explained in Sections 3.2.4 and 2.5). In this case, only one object must be included in the returned sequence (since the id_gene() uniquely identifies an SLdb.genes object), hence the use of [0].*
- *in a similar way to g, variable go is assigned the corresponding SLdb.genomic_object object(s).*
- *variable es is assigned the nsdb.EmblSeq object the id of which is the same as the id given as an input parameter in the current operation (special syntax for the parameter is: getById:embl_id).*

```
   */
   getById() = object {
      g = query(Server.geneF,
            id_gene()=getById:id)[0],
      go = query(Server.genobjF,
            id_gene()=getById:id),
        es = Server.emblF.getEmblSeq(getById:embl_id)
      };
    }
  }

/*
The GenomicObject view represents functionally significant
regions of nucleotide sequences.
*/
view GenomicObject {
  definition {
    /*
    The firstPos() and lastPos()correspond to the beginning
    and end of the genomic object after taking into account
    its direction in the chromosome.
    */
    long firstPos();
    long lastPos();
    string nucSeq();
    boolean directionIsL2R();
  }
  source {
    // Source reference to the genomic object in SubtiList.
    SLdb.genomic_object  go;
    // Source reference to the relevant kitong in SubtiList.
    SLdb.kitong          k;
  }
  mapping {
    firstPos() = jexpression { (int)(k.pos_kb()*1000) +
      go.direction().startsWith("+") ? go.first()
                                     : go.last() };
   lastPos() = jexpression { (int)(k.pos_kb()*1000) +
      go.direction().startsWith("+") ? go.last()
                                     : go.first() };
   nucSeq() = jmethod { myNucSeq() };
   directionIsL2R() = jexpression {
      go.direction().startsWith("+")
   };
}
user_method {
   /*
   It reverses the order of nucleotides of a given
   nucleotide sequence string and returns the resulting
   string.
   */
   String myNucReverse(String nseq) {
      String rev="";
      for (int i=0; i<nseq.length(); i++)
         rev = rev + nseq.charAt(nseq.length()-1-i);
      return rev;
   }


   /*
   It complements the nucleotides of a given nucleotide
   sequence string and returns the resulting string.
   */
   String myNucComplement(String nseq) {
      String compl="";
```

```
  for (int i=0; i<nseq.length(); i++)
    switch (nseq.charAt(i)) {
      case 'a': compl = compl + "t"; break;
      case 'c': compl = compl + "g"; break;
      case 'g': compl = compl + "c"; break;
      case 't': compl = compl + "a"; break;
      default : compl = compl + "?"; break;
  }
  return compl;
}


/*
It calculates the nucleotide sequence string of a
genomic object given the source references to the
relevant genomic object and kitong information in
Subtilist.
*/
String myNucSeq() {
  /*
  The nucleotide sequence string of a kitong in
  SubtiList always has a left-to-right direction. If the
  direction of the nucleotide sequence of the genomic
  object is left-to-right (i.e. "+"), then the genomic
  object nucleotide sequence string is a straight
  substring of the kitong which it belongs to.
  Otherwise, the nucleotides of the retrieved substring
  have to be complemented and their order reversed.
  */
  if (go.direction().startsWith("+"))
    return (k.nuc_seq().substring(
        go.first(), go.last()));
      else
         return myNucReverse(myNucComplement(
             k.nuc_seq().substring(go.first(), go.last()) ));
    }
  }
}


/*
The FunctionalClass view represents classes into which
protein coding genes are classified based on their function.
*/
view FunctionalClass {
  definition {
    string id();
    string description();
  }
  source {
    SLdb.buff_class_gene        b;
    SLdb.classification         c;
  }
  mapping {
    id() = b.id_category();
    description() = c.description();
  }
}


/*
The Gene view represents genes, which are regions of DNA
sequence having some specific functionality, and containing
```

```
DNA sequences that encode for proteins/RNAs, as well as any
associated regulatory signals (e.g. promoters, -35 signals,
etc.).
*/
view Gene {
  definition {
    typedef sequence<Promoter> PromoterList;
    typedef sequence<Minus10Signal> Minus10SignalList;
    typedef sequence<Minus35Signal> Minus35SignalList;
    typedef sequence<Terminator> TerminatorList;
    typedef sequence<ProteinCoding> ProteinCodingList;
    typedef sequence<string> StringList;
    string id();
    string name();
    string ecNumber();
    string function();
    string description();
    FunctionalClass functionalClass()
      raises (PS::ServerError);
    string codonUsageClass();
    long pos();
    PromoterList getPromoter()
      raises (PS::ServerError);
    Minus10SignalList getMinus10Signal()
      raises (PS::ServerError);
    Minus35SignalList getMinus35Signal()
      raises (PS::ServerError);
    TerminatorList getTerminator()
      raises (PS::ServerError);
  ProteinCodingList getProteinCoding()
    raises (PS::ServerError);
  StringList references();
}
source {
  SLdb.genes                  g;
  SLdb.genomic_object[]       go;
  nsdb.EmblSeq                es;
}
mapping {
  id() = g.id_gene();
  name() = g.name();
  ecNumber() = g.ec_number();
  function() = g.function();
  description() = es.getDescription();
  functionalClass() = object {
    b = query(Server.classgeneF,
          id_gene() = name())[0],
    c = query(Server.classF,
          id_category() = b.id_category())[0]
  };
  codonUsageClass() = jmethod { myCodonUsageClass() };
  pos() = jexpression { (int)(g.pos_kb()*1000) };
  /*
  The operation getPromoter() of the view Gene returns one
  or more new Promoter object, assigning the source
  references go, k. Note the use of the iterator operator
  [] which means that, for each genomic object returned by
  the first query, a new promoter object will be created:
```

```
  •  variable go is assigned the SLdb.genomic_object
     object the type of which is 'promoter' and the id
     of which is the same as the id_gene of the current
     Gene object.

  •  variable k is assigned the SLdb.kitong object the
     id of which is the same as the id_kitong of the
     current go.

The operations getMinus10Signal(), getMinus35Signal(),
getTerminator(), and getProteinCoding() work in a
similar way.

*/

getPromoter()[] = object {

   go = query(Server.genobjF,

          (type() = 'promoter') and (id_gene() = id()))[],

   k = query(Server.dnafragF,

          id_kitong() = go.id_kitong())[0]

};

getMinus10Signal()[] = object {

   go = query(Server.genobjF,

          (type() = '-10_signal') and

          (id_gene() = id()))[],

   k = query(Server.dnafragF,

          id_kitong() = go.id_kitong())[0]

};

getMinus35Signal()[] = object {

   go = query(Server.genobjF,

          (type() = '-35_signal') and

          (id_gene() = id()))[],

   k = query(Server.dnafragF,

          id_kitong() = go.id_kitong())[0]
  };
  getTerminator()[] = object {
    go = query(Server.genobjF,
           (type() = 'terminator') and
           (id_gene() = id()))[],
    k = query(Server.dnafragF,
           id_kitong() = go.id_kitong())[0]
  };
  getProteinCoding()[] = object {
    go = query(Server.genobjF,
           (type() = 'CDS') and (id_gene() = id()))[],
    k = query(Server.dnafragF,
           id_kitong() = go.id_kitong())[0]
  };
  references() = jmethod { myRefs() };
}
user_method {
  /*
  It returns the list of bibliographic references that are
  relevant to the specific gene.
  */
  String[] myRefs()
    throws type.NoResult, type.InvalidArgumentValue {
    bibliography.Reference ref=null;
    type.DbXref[]          dbRefL;
    String[]               strL=null;
    biblio.BiblioFormatter bibFormatter=null;
  /*
  BiblioFormatter is a class containing methods for
  formatting EMBL bibliographic references. It is
  borrowed from the implementation of an example
  bibliography client to the EMBL CORBA server and used
  in the implementation of this server. The code is not
  included in order to keep the example short.
  */
  bibFormatter = new biblio.BiblioFormatter();
  dbRefL = es.getReferences();
  strL=new String[dbRefL.length];
  for (int i=0; i<dbRefL.length; i++) {
    ref = Server.libF.getReference(
        dbRefL[i].primary_id);
    strL[i] = bibFormatter.formatBiblioRef(ref);
  }
  return strL;
}

/*
It returns the codon usage class the specific gene
belongs to, as explained in Sections 5.2 and 5.3.
*/
String myCodonUsageClass() {
  switch (g.codon_usage()) {
    case 0: return "Not a protein coding gene";
    case 1: return "Class 1";
    case 2: return
      "Class 2: High expression in exponential growth";
    case 3: return "Class 3: Prophages";
```

```
            default: return "Unknown codon usage class";
          }
        }
      }
    }

    /*
    The views Promoter, Minus10Signal, Minus35Signal, and
    Terminator represent DNA sequences that act as signals. The
    view ProteinCoding represents DNA sequences that encode for
    proteins. They all inherit the definition as well as the
    implementation (hence, the use of extends keyword) from the
    view Genomic Object.
    */
    view Promoter : GenomicObject(extends) {}
    view Minus10Signal : GenomicObject(extends) {}
    view Minus35Signal : GenomicObject(extends) {}
    view Terminator : GenomicObject(extends) {}
    view ProteinCoding : GenomicObject(extends) {}
  }
```

5.5 A Client Application

In order to test the generated mediator, a simple textual client application has been developed that uses the IDL interfaces of the mediator CORBA server. The complete client code developed using the Java programming language is provided in Appendix E. Parts of the client application code along with results generated using example data are presented in this Section.

The client is called with a number of arguments, i.e.

i. the IOR of the *GeneFactory* object of the SubtiList CORBA server

ii. the SubtiList accession

iii. the IOR of the *Embl* object of the EMBL CORBA server

iv. the EMBL accession

```
// Check input arguments.
if (args.length < 4) {
  System.out.println("Usage: java Client <GeneFactory-ior>
                      <gene-id> <Embl-ior> <embl-id>");
  return;
}
```

The *GeneFactory* and *Embl* IORs are retrieved into suitable factory objects (the *string_to_object* CORBA operation is used to obtain the object references from their stringified form, as explained in Section 2.4).

```
// Obtain IORs of factory objects.
geneFactory = BacterialGenome.GeneFactoryHelper.narrow(
  client.retrieveIORFromFile(orb, geneFactory_ior));
emblFactory = nsdb.EmblHelper.narrow(
  client.retrieveIORFromURL(orb, embl_ior));
```

General identification, naming, approximate location and functional description information about the gene is retrieved using the factory objects, and then printed out.

```
// Get gene data.
nsdb.EmblSeq emblSeq=null;
emblSeq = emblFactory.getEmblSeq(embl_id);
BacterialGenome.Gene gene=null;
gene = geneFactory.getById(gene_id, embl_id);
System.out.print("\nGene: " + gene_id
  + "\nid: " + (gene.id()==null ? "" : gene.id())
  + "\nname: " + (gene.name()==null ? "" : gene.name())
  + "\nec number: " +
    (gene.ecNumber()==null ? "" : gene.ecNumber())
  + "\nfunction: " +
    (gene.function()==null ? "" : gene.function())
  + "\ndescription: " +
    (gene.description()==null ? "" : gene.description())
  + "\npos: " + gene.pos()
);
```

Any bibliographic references associated with the specific gene are also retrieved and printed out.

```
// Get gene references.
String[] refs = gene.references();
System.out.print("\nreferences (no. = "
  + refs.length + "):");
for (int i=0; i<refs.length; i++) {
  System.out.print("\n    " + (i+1) + ". " + refs[i]);
}
```

The genomic objects associated with the specific gene together with their location and nucleotide sequence are retrieved (the client includes similar code for genomic objects other than promoters, such as –10 signals, -35 signals and terminators).

```
// Get gene promoter(s).
BacterialGenome.Promoter[] promoters=null;
promoters = gene.getPromoter();
for (int i=0; i<promoters.length; i++) {
  System.out.print("\nPromoter " + (i+1) + " location: ("
    + promoters[i].firstPos() + ", "
    + promoters[i].lastPos() + ")    "
```

```
        + ((promoters[i].directionIsL2R()) ? "->" : "<-")
        + "\n              sequence: "
        + promoters[i].nucSeq()
        );
    }
```

If the gene is a protein coding one, the location and nucleotide sequence of its protein coding part as well as its functional and codon usage classes are retrieved.

```
    /* For protein coding genes, get associated sequence,
       codon usage class, and functional class. */
    BacterialGenome.ProteinCoding[] pCoding=null;
    pCoding = gene.getProteinCoding();
    BacterialGenome.FunctionalClass func_class=null;
    func_class = gene.functionalClass();
    for (int i=0; i<pCoding.length; i++) {
      System.out.print("\nProteinCoding " + (i+1)
        + " location: (" + pCoding[i].firstPos() + ", "
        + pCoding[i].lastPos() + ")    "
        + ((pCoding[i].directionIsL2R()) ? "->" : "<-")
        + "\n                sequence: " +
          pCoding[i].nucSeq()
        + "\n                codon usage class: " +
          gene.codonUsageClass()
        + "\n                functional class: " +
          (func_class.id()==null ? "" : func_class.id())+
          ": " + (func_class.description()==null ? "" :
          func_class.description())
      );
    }
```

The client application was tested with a number of input data, an example of which is provided here in order to illustrate the generated output. The following input parameters were used:

```
gene-id: BG12674

embl-id: AF084950
```

The resulting output was:

```
Gene: BG12674

id: BG12674

name: sipV

ec number: EC 3.4.21.89

function: involved in the processing of secretory preproteins

description: Bacillus amyloliquefaciens signal peptidase type I
(sipV) gene, complete cds.

pos: 1121500

references (no. = 2):

   1. Chu H.H., Hofemeister J.W.; "Multiple type I signal
peptidases of Bacillus amyloliquefaciens"; Unpublished.

   2. Chu H.H., Hofemeister J.W.; ; Submitted (20-AUG-1998) to
the EMBL/GenBank/DDBJ databases. Molecular Genetics, Institute
of Plant Genetics and Crop Plant Research (IPK), Corrensstrasse
3, Gatersleben, SA 06466, Germany

Terminator 1 location: (1121985, 1122015)   ->

             sequence: taaaaagacgctaattgaagaggcgtcttt

ProteinCoding 1 location: (1121480, 1121983)   ->

                sequence:
atgaaaaaacggttttggtttcttgccggtgtagtgtccgttgttctcgccattcaggttaaa
aatgctgtctttattgattacaaggtagaaggcgtcagtatgaacccgaccttccaggaagga
aacgaattgttggtcaataaattttcgcatcgatttaaaccatccatcgttttgacatcgtc
ctttttaaaggccctgatcataaagtgctgattaaacgggtaatcggcttgcccggtgaaacg
atcaaatataaagatgatcagctgtatgtgaacggaaagcaggttgctgagccatttttgaag
catttgaaatctgtttctgccggcagccatgtaacgggtgatttttctttgaaagatgtgacg
```

```
ggaacaagcaaggtgccgaaaggaaaatatttgtcgttggagataatcgcatatacagcttc
gacagccggcattttggtccgataagagaaaaaaatattgtcggtgtgatttctgatgccgaa
                    codon usage class: Class 1
                    functional class: 1.6: Protein secretion
```

5.6 Conclusions

The use of the mediator system described in this thesis has been demonstrated in this Chapter with an example from the domain of molecular biology. The example has also shown the complete software development process that was devised in order to achieve the goal of data integration.

In particular, bacterial genome data from two sources have been integrated. One source is the SubtiList database of the Bacillus subtilis genome, which contributed DNA sequence and location information for protein coding genes and other functionally significant regions of DNA sequences (e.g. promoters, terminators), as well as information on functional classification of protein gene products and codon usage. The other is the EMBL nucleotide sequence database, which contributed description and bibliographic reference information.

A CORBA wrapper for the SubtiList database has been generated using the Persistence™ software and modified minimally for the purposes of restricted database access. Apart from those modifications, the generated IDL definitions did not change in any other way that would facilitate integration. The data of the EMBL database have been accessed using the available CORBA server. No modifications were made on that wrapper. It should be emphasised that the two chosen CORBA wrappers were developed completely independently by different people and for different purposes. The data integration approach presented in this thesis does not pose any special requirements on the CORBA wrapper definitions. Thus, data from any CORBA wrapper can be integrated. This kind of flexibility is important especially in the field of molecular biology, where data providers tend to make their data available using their own customised definitions.

An object model of bacterial genome data has been developed to model the wider domain. Part of that model has been used for the current application. Based on the object model, a number of IDL interfaces for the mediator CORBA server have been defined. It has already been mentioned in Section 2.2 that there are software tools available that translate UML object models to IDL definitions automatically. However, when mapping specifiers need to work at the level of IDL, they would require control over the mediator IDL definitions, in order to achieve an efficient and not over-complicated CORBA server. So, it is expected that mediator specifiers would opt for a mediator IDL that is manually defined. This has also been the case in the specific application, where the mediator IDL has been manually defined for reasons of efficiency and simplicity.

Having the two CORBA wrappers and the mediator CORBA server IDL definition, the mapping of mediator to wrapper IDL interfaces was defined. The complete view definition code for the specific example has been provided in this Chapter. These view definitions were given as input to the IDL View Generator that generated the required mediator. The mediator integrates Bacillus subtilis data from the two wrappers and provides the integrated data through the preferred IDL definition.

In order to test the generated mediator, a simple textual client application has been developed that uses the IDL interfaces of the mediator CORBA server. Parts of the client application code along with results generated using example data have been presented in this Chapter. The client application retrieves integrated Bacillus subtilis data transparently. That is, it does not need to have any knowledge of, or access directly the SubtiList and EMBL data sources. It simply retrieves the integrated data from a single server, which is the generated mediator.

Chapter 6: CONCLUSIONS

An approach has been examined in this thesis that solves some of the problems involved in integration of molecular biology data from distributed, heterogeneous and autonomous data sources. Molecular biology data is managed by a variety of systems distributed around the world. The management and access of data provided by different sources differ at many levels, i.e. programming language, platform, data representation. Data sources operate, to a large extent, independently and autonomously. They generally want to manage, represent and provide access to 'their' data in their own ways that may or may not agree with available standards. Under these conditions, data integration becomes a challenging problem.

The approach of this thesis solves some of the problems involved in data integration using CORBA within a mediator-based architecture. CORBA handles heterogeneity at the programming language and platform levels, and provides network transparency. This allows CORBA-based data integration approaches to focus on resolving metamodel and schematic heterogeneity. Mediator-based architectures have been used to integrate information from autonomous and heterogeneous data sources. They resolve heterogeneity at the level of data representation, i.e. metamodel and schematic heterogeneity. The useful features of CORBA are utilised within a mediator-based architecture resolving schematic heterogeneity at the IDL level in a semi-automatic way. The approach supports integration of data from molecular biology data sources that provide access through CORBA. It also supports creation of customised CORBA views of such data sources.

In order to investigate the feasibility and suitability of this approach, a mediation system has been developed and described in this thesis. Similar to other mediator-based approaches ([Chawathe 94], [Carey 95], [Tomasic 96], [Goh 94]), mediators of the developed system transform data from wrappers to a common schema resolving schematic heterogeneity, and provide support for the integration of the transformed data (i.e. through object composition and union of objects). However, unlike many other mediator-based approaches and molecular biology information systems that develop their own proprietary middleware, the system described in this thesis is based on the OMG CORBA.

The decision to base this system on CORBA means that it works in an open, well-founded, widely supported and heterogeneous distributed computing environment across all major hardware platforms and operating systems. As a result, it can easily be used by other systems within CORBA, as well as take advantage of other present or future development work in CORBA (such as, the Naming, Trader, and Collection Service). At the same time, this decision has not been restrictive. The particular strengths and weaknesses of CORBA have been considered, and alternative solutions have been sought in cases where the specific CORBA standard did not suit the purposes of the specific mediation system.

A number of issues, which are important for the design of a CORBA mediator system, have been examined in the thesis and alternative solutions have been discussed. These design issues guided the development of the different components of the system.

A fundamental design decision for the architecture of the mediator system described was to utilise CORBA for communication between mediators and wrappers and between mediators and end-user applications. Since any CORBA server can be used as a source by a mediator (that is, in principle, even another mediator), mediators could be utilised as modules using which other more complex or more specialised mediators would be constructed. This results in a flexible, modular and extensible system. Moreover, in order to make sure the system is fully integrated within CORBA, its main software components have been designed with the requirement of CORBA access in mind. That includes both the IDL View Generator, a CORBA wrapper of which has been developed, as well as any generated mediators, which are CORBA servers ready to be used by prospective CORBA clients. As a result, mediator specifiers can easily use the IDL View Generator to define a mediator IDL definition and generate the mediator code. More importantly, application developers can easily and flexibly use the integrated data provided by one or more mediators, programming in their preferred CORBA supported language for the development of appropriate applications.

Another important design decision was to resolve schematic heterogeneity at the IDL level. IDL constructs suffice to express object models of considerable complexity, thus making it possible to treat the IDL as an object modelling language in which the common schema definitions are expressed. Advantages of this approach include existing knowledge, wide use and standardisation of the language, as well as its integration within CORBA, which means that no translation between different models and definition languages is necessary. In the future, and when relevant mappings are fully standardised and associated tools become available, it would be a worthwhile challenge to investigate the resolution of schematic heterogeneity at the object modelling level, i.e. at the UML level with/without the use of XML.

Resolving schematic heterogeneity at the IDL level involves mapping between one or more IDL schemas of available source CORBA servers and the preferred IDL schema of the target CORBA server. In the system described in this thesis, mapping specifiers need only provide high-level descriptions of the mapping of target IDL to source IDL definitions. High-level descriptions are expressed in the mapping language especially developed for this purpose. Using the mapping definitions, the system generates the target IDL and the implementation code of the target CORBA server. In other words, the system follows a semi-automatic approach for the production of target CORBA servers that implement mappings, as opposed to leaving developers implement mappings manually, i.e. without any supporting tools. The semi-automatic generation has several advantages over manual implementation of mappings. These include fast implementation of a working mapping, no need for in-depth CORBA experience, and easier maintenance of the resulting code especially when many source servers are involved and in the light of frequently evolving source and/or target IDL schemas. Obviously, the generated code could later be manually modified, if required. Within a mediator-based architecture, the semi-automatic approach facilitates the creation of mediators by including a mediator generator.

A view definition language has been developed to define the common schema of a mediator and express the mapping between wrapper schemas and the common schema. The language has a high-level notation, in order to provide easy and concise specification of mappings and to facilitate modification of mapping definitions. The later is especially important in the field of molecular biology for the support of source and/or target schema evolution due to frequent changes of data structures. Complex mappings are further supported with the inclusion of a number of procedural features in the language. The IDL constructs supported by the mapping language (including support of inheritance) suffice to express object models of considerable complexity. However, the mapping of unsupported IDL constructs has to be implemented manually. It would be beneficial to investigate the inclusion of additional IDL constructs in the mapping language (such as, *structs* and *value types*) for improved support of more wrappers, and with regard to efficiency issues.

The generated mediators of the system described in this thesis do not support ad hoc queries, a feature that is usually included in other mediator-based systems. Instead, the mapping language supports fixed parameterised queries, which are encapsulated in CORBA parameterised operations of the target schema, an approach that is suitable for domain objects represented in IDL. It would be useful investigating the support of type-restricted queries as an alternative. Similar to fixed parameterised queries, type-restricted queries are also suitable for domain objects represented in IDL and provide fully typed query results.

Wrapper development was beyond the scope of the system described in this thesis. Instead, the ways the system can work with available wrappers have been investigated. The functionality required by wrappers, in order to be able to use queries within mappings has been defined. The current implementation provides query mapping to wrappers generated by Persistence™. In principle, the system could be extended with a number of options for the generated code that interfaces with wrappers. For instance, support of collections and iterators to traverse those collections could be added as an alternative to sequences of objects that currently represent query results. Such customisation options would make possible the combination of the system with alternative wrappers. Exploring ways to interface this system with other systems in the biological domain, which offer CORBA interfaces and query support for accessing their underlying data ([Coupaye 99]), would also be of great interest.

Another design decision regarded the different models for supporting target object derivation. The choice of the object composition model enables powerful data integration and customisation of data representation. This feature is especially important in the field of molecular biology due to the different levels of detail at which biological objects are modelled in data sources, and the high degree of interrelationships.

The bacterial genomes application has served its purpose of demonstrating the use of the system by integrating data from the SubtiList database (using the CORBA server especially developed) and the EMBL database (using the available CORBA server). The two chosen CORBA wrappers were developed completely independently and no modifications were made that would facilitate integration. That is, the data integration approach presented in this thesis does not pose any special requirements on CORBA wrapper IDL definitions, making possible the integration of data from any CORBA wrapper. This kind of flexibility is important especially in the field of molecular biology, where data providers tend to make their data available using their own customised definitions.

The application has also demonstrated the complete software development process that was devised in order to achieve the goal of data integration. In summary, the process includes the analysis of the available data sources and their respective CORBA servers, the development of a CORBA wrapper using an automatic code generation approach, the development of an object model for the application domain, the development of the mediator IDL definition based on the object model, the definition of the mapping of mediator to wrapper IDL interfaces, the generation of the mediator code using the IDL View Generator, and finally the development of a client that retrieves the integrated data from the mediator. In particular, the mediator IDL has been manually defined based on the object model, as opposed to automatically generating it using software tools for reasons of efficiency and simplicity.

It would be of interest to use the mediation system for the integration of data from other biological data sources that offer access via CORBA interfaces, especially, to investigate the suitability of the system in tackling more complex and biologically relevant data integration problems. At the time of development of the mediation system presented in this thesis not many CORBA wrappers of biological data were available, which restricted the choice of possible applications.

As more CORBA servers wrapping biological data sources are becoming available, new possibilities arise of integrating data of interest using the mediation system.

Using the mediation system for the creation of customised CORBA views of CORBA-wrapped data sources has useful applications, too. As one example, CORBA data wrappers within the scope of OMG LSR but not conforming to the standard IDL definitions, could provide alternative and conformant IDL definitions with CORBA views developed using the system described in this thesis. Other examples may include providing specialised CORBA data wrappers for different user groups, supporting different levels of access to CORBA wrappers of data sources, etc.

Finally, the focus of the approach has been on integrating data and creating customised views of data from CORBA-wrapped molecular biology data sources. In principle, the approach can be applied to the creation of customised views of any CORBA objects (i.e. not 'data-wrapping' objects). In that case, the views used for the implementation of mediator generators would restructure the source CORBA objects providing access through target IDL definitions. The approach is particularly suited for the purposes of object composition with a target object deriving from one or more source objects and composing their source object elements (i.e. attributes, methods) into preferred target ones. The restructuring and mapping of object methods is a particularly challenging problem and would require further investigation.

Appendices

Appendix A. The mapping language specification in BNF

```
Specification        ::=
                     ( <HASH> <INCLUDE> <JAVA_STRING_LITERAL> )*
                     ServerDecl
                     <MODULE> JavaScopedName
                     <LBRACE> (FwdViewDecl | ViewDecl )* <RBRACE>
                     <EOF>
ServerDecl           ::= <SERVER> <ID> <LBRACE> ( FactoryDecl )* <RBRACE>
FactoryDecl          ::= <FACTORY> <ID>
                     <LBRACE>
                             <FTYPE> JavaScopedName <SEMICOLON>
                             <IOR> <JAVA_STRING_LITERAL> <SEMICOLON>
                     <RBRACE>
FwdViewDecl          ::= <VIEW> <ID> <SEMICOLON>
ViewDecl             ::= Header
                     <LBRACE>
                             ( PublicationInfo )? ( Definitions )? ( SourceRefs )?
                             ( Mapping )? ( UserMethods )?
                     <RBRACE>
Header               ::= <VIEW> <ID>
                     ( <COLON> <ID> ( <LPAREN> <EXTENDS> <RPAREN> )?
                     ( <COMMA> <ID> ( <LPAREN> <EXTENDS> <RPAREN> )? )* )?
PublicationInfo      ::= <PUBL_LOC> <JAVA_STRING_LITERAL>
Definitions          ::= <DEF> <LBRACE> DefinitionBody <RBRACE>
DefinitionBody       ::= ( Definition <SEMICOLON> )*
Definition           ::= TypeDecl
                     | AttributeDecl
                     | OperationDecl
TypeDecl             ::= <TYPEDEF> <SEQUENCE> <LT> TypeSpec <GT> <ID>
AttributeDecl        ::= <READONLY> <ATTRIBUTE> TypeSpec
                     <ID> ( <COMMA> <ID> )*
OperationDecl        ::= ( <ONEWAY> )? OperTypeSpec
```

```
                        <ID> ParamDecls ( <RAISES> IDLScopedNameList )?
OperTypeSpec            ::= TypeSpec
                        | <VOID>
ParamDecls              ::=
                        <LPAREN> ( ParamDecl ( <COMMA> ParamDecl )* )? <RPAREN>
ParamDecl               ::= ParamAttr TypeSpec <ID>
ParamAttr               ::= <IN>
                        | <OUT>
                        | <INOUT>
SourceRefs              ::= <SOURCE>
                        <LBRACE> ( SourceRefDecl <SEMICOLON> )* <RBRACE>
SourceRefDecl           ::= JavaScopedName ( <LBRACKET> <RBRACKET> )?
                        <ID> ( <COMMA> <ID> )*
Mapping                 ::= <MAPPING>
                        <LBRACE> ( MappingSpec <SEMICOLON> )* <RBRACE>
MappingSpec             ::= Target <EQ> Source
Target                  ::= <ID> <LPAREN> <RPAREN>
                        ( <LBRACKET> <RBRACKET> )?
Source                  ::= JavaMethod
                        | JavaExpression
                        | ObjectSpec
                        | MyScopedName
JavaMethod              ::= <JAVA_METHOD> <LBRACE> <ID> Arguments <RBRACE>
JavaExpression          ::= <JAVA_EXP> <LBRACE> stringToMatchingBrace <RBRACE>
UserMethods             ::= <METHOD_IMPL>
                        <LBRACE> stringToMatchingBrace <RBRACE>
stringToMatchingBrace   ::= java code
Arguments               ::= <LPAREN> <RPAREN>
                        | <LPAREN> Argument ( <COMMA> Argument )* <RPAREN>
Argument                ::= JavaExpression
                        | JavaMethod
                        | JavaLiteral
                        | MyScopedName
ObjectSpec              ::= <OBJECT>
                        <LBRACE>
                        ComponentSpec ( <COMMA> ComponentSpec )*
                        <RBRACE>
ComponentSpec           ::= <ID> ( <LBRACKET> <RBRACKET> )?
                        <EQ> ( JavaMethod | QuerySpec | MyScopedName )
QuerySpec               ::= <QUERY>
                        <LPAREN> JavaScopedName <COMMA> Expression <RPAREN>
                        ( <LBRACKET> ( <INTEGER_LITERAL> )? <RBRACKET> )?
Expression              ::= ORExpression
ORExpression            ::= ANDExpression ( <OR> ANDExpression )*
ANDExpression           ::= RelationalExpression ( <AND> RelationalExpression )*
RelationalExpression    ::= UnaryExpression
                        ( ( <EQ> | <NE> | <LT> | <GT> | <LE> | <GE> ) UnaryExpression ) ?
UnaryExpression         ::= <NEG> PrimaryExpression
                        | PrimaryExpression
                        | SQLLiteral
PrimaryExpression       ::= ( <LPAREN> Expression <RPAREN> )
                        | MyScopedName
JavaLiteral             ::= <JAVA_STRING_LITERAL>
                        | <CHARACTER_LITERAL>
                        | <BOOLEAN_LITERAL>
                        | <INTEGER_LITERAL>
                        | <FLOATING_POINT_LITERAL>
SQLLiteral              ::= <SQL_STRING_LITERAL>
                        | <CHARACTER_LITERAL>
                        | <INTEGER_LITERAL>
                        | <FLOATING_POINT_LITERAL>
TypeName                ::= <ID>
IDLScopedNameList       ::=
                        <LPAREN>
                                IDLScopedName ( <COMMA> IDLScopedName )*
                        <RPAREN>
IDLScopedName           ::= <ID> ( <LPAREN> <INTERFACE> <RPAREN> )?
                        ( <DCOLON> <ID> ( <LPAREN> <INTERFACE> <RPAREN> )? )*
JavaScopedName          ::= <ID> ( <DOT> <ID> )*
MyScopedName            ::= MyScopedNameElem
```

```
                        ( <DOT> MyScopedNameElem )*
MyScopedNameElem        ::= <ID> <COLON> <ID>
                        | <ID> Arguments ( <LBRACKET> <RBRACKET> )?
                        | ( <SELF> <DOT> )? <ID> ( <LBRACKET> <RBRACKET> )?
getNextElement          ::= java code
TypeSpec                ::= BaseTypeSpec
                        | StringType
                        | WideStringType
                        | TypeName
BaseTypeSpec            ::= FloatingPtType
                        | IntegerType
                        | CharType
                        | WideCharType
                        | BooleanType
                        | OctetType
                        | AnyType
FloatingPtType          ::= <FLOAT>
                        | <DOUBLE>
IntegerType             ::= SignedInt
                        | UnsignedInt
SignedInt               ::= <SHORT>
                        | <LONG>
                        | <LONG> <LONG>
UnsignedInt             ::= <UNSIGNED> <SHORT>
                        | <UNSIGNED> <LONG>
                        | <UNSIGNED> <LONG> <LONG>
CharType                ::= <CHAR>
WideCharType            ::= <WCHAR>
BooleanType             ::= <BOOLEAN>
OctetType               ::= <OCTET>
AnyType                 ::= <ANY>
StringType              ::= <STRING>
WideStringType          ::= <WSTRING>
```

Appendix B. The SubtiList CORBA Server IDL

The PS.idl

```
// =================================================================
// File:    PS.idl
// Author:  Persistence(TM) by Persistence Software, Inc.
// =================================================================
// Copyright (C) 1997 Persistence Software, Inc.
// All rights reserved.
// Use of this code without permission is prohibited.
// =================================================================
// The PS module contains declarations for basic capabilities
// which are commonly used by application servers and client
// programs. It defines a transaction interface, enumerations,
// specialized attribute types, and exception types. This chapter
// discusses the transaction interfaces and enumerations. The
// chapters that follow discuss specialized attribute types and
// exception types.
// =================================================================

module PS
{
      interface Defs
      {
            const unsigned long k_noWait = 0;
            const unsigned long k_infiniteWait = 4294967295;

            enum DatabaseOperation { k_insert, k_remove, k_read,
            k_update };

            enum DatabaseType { k_noDatabase, k_informix, k_oracle,
            k_sybase, k_odbc };

            enum ErrorCategory { k_cache, k_class, k_connection,
            k_database, k_decimal, k_dock, k_export, k_factory,
            k_import, k_informixError, k_key, k_odbcError,
            k_oracleError, k_server, k_sybaseError, k_transaction,
            k_utility, k_thread, k_attributeValidity,
            k_classValidity, k_fkeyValidity, k_license, k_project,
            k_relationshipValidity, k_validity, k_filePool,
            k_functionDisp, k_parser, k_parseTreeNode };

            enum LockType { k_shared, k_exclusive };

            enum ObjectSource { k_cacheThenDatabase, k_cacheOnly };

            enum ServerState { k_active, k_idle, k_starting, k_stopping,
            k_failed };

            enum TableStatus { k_noTables, k_someTables, k_allTables };
      };

      struct ArcId
      {
            unsigned long hiword;
            unsigned long loword;
      };
      typedef sequence<ArcId> ArcId_Cltn;

      struct ClassId
      {
            short id;
            string alias;
      };
      typedef sequence<ClassId> ClassId_Cltn;
      typedef sequence<octet> Binary;
      typedef string Decimal;
      typedef string Time;
      typedef sequence<octet, 12> Oid;

      exception ServerError
      {
            string name;
            string description;
            Defs::ErrorCategory category;
            long code;
            long databaseCode;
      };

      interface PObject_Factory
      {
            PS::ArcId makeArcId(in string arcName)
                  raises(ServerError);
            PS::ClassId makeClassId(in string alias)
                  raises(ServerError);
            readonly attribute string className;
      };

      interface PObject
      {
            void clear();
            void lock();
            void read();
            void write();

            readonly attribute boolean locked;
            readonly attribute string className;
            readonly attribute unsigned long classIdent;
      };

      typedef sequence<PObject> PObject_Cltn;

      interface TransactionMgr
      {
            void begin() raises(ServerError);
            void commit() raises(ServerError);
            void rollback() raises(ServerError);
            void rollbackToSavePoint(in string savePoint)
                  raises(ServerError);
            void setSavePoint(in string savePoint)
                  raises(ServerError);
            void write() raises(ServerError);

            readonly attribute unsigned long nestingCount;
            attribute boolean lockUponRead;
      };
};
// =================================================================
```

The PSAdmin.idl

```
// ===================================================================
// File:    PSAdmin.idl
// Author:  Persistence(TM) by Persistence Software, Inc.
// ===================================================================
// Copyright (C) 1997 Persistence Software, Inc.
// All rights reserved.
// Use of this code without permission is prohibited.
// ===================================================================
// The PSAdmin module defines advanced capabilities that are used
// to configure an application server. Most client programs would
// not need these operations, and it is possible to "hide" the
// interface to the PSAdmin module such that a client cannot
// easily invoke the methods. The classes of the PSAdmin module
// are used to manage the server, database tables, database
// connections, the cache, and the event log.
// ===================================================================

#include "PS.idl"

module PSAdmin
{
       // 14.2 PSAdmin::ServerMgr
       interface ServerMgr
       {
              void reset();
              void start();
              void stop();

              readonly attribute PS::Defs::ServerState state;
       };

       // 14.6 PSAdmin::DBConnParam
       struct DBConnParam
       {
              string applicationName;
              string characterSet;
              string databaseName;
              PS::Defs::DatabaseType databaseType;
              unsigned long maxConnections;
              unsigned long maxDatabaseCursors;
              unsigned long minConnections;
              string nationalLanguage;
              string password;
              string serverName;
              string userName;
              unsigned long waitTime;
              boolean trimTrailingBlanks;
              boolean charColumnType;
       };

       // 14.8 PSAdmin::DBConnSpec
       interface DBConnSpec
       {
              attribute DBConnParam connParam;
              long sendNonSelectSQL(in string sqlStatement)
                     raises(PS::ServerError);
              long sendNonSelectSP(in string spStatement)
                     raises(PS::ServerError);
              string print();
       };
```

```
// 14.4 PSAdmin::DBClassMgr
interface DBClassMgr
{
	void createTables(in string classSpec)
		raises(PS::ServerError);
	void dropTables(in string classSpec)
		raises(PS::ServerError);
	void exportTables(in string classSpec, in string path)
		raises(PS::ServerError);
	void importTables(in string classSpec, in string path)
		raises(PS::ServerError);
	void lockTables(in string classSpec,
		in PS::Defs::LockType lockType)
		raises(PS::ServerError);
	PS::Defs::TableStatus doTablesExist(in string classSpec)
		raises(PS::ServerError);

	void setDBConnSpec(in string classSpec,
		in DBConnSpec connSpec)
		raises(PS::ServerError);
	void setTableName(in string classSpec, in string tableName)
		raises(PS::ServerError);
	void setSPSpecs(in string classSpec,
		in PS::Defs::DatabaseOperation op, in string spName)
		raises(PS::ServerError);

	DBConnSpec getDBConnSpec(in string classSpec)
		raises(PS::ServerError);
	string getTableName(in string classSpec)
		raises(PS::ServerError);
	string getSPName(in string classSpec,
		in PS::Defs::DatabaseOperation op)
		raises(PS::ServerError);
};

// 14.10 PSAdmin::DBConnMgr
typedef sequence<DBConnSpec> DBConnSpec_Cltn;

interface DBConnMgr
{
	DBConnSpec_Cltn allDBConnSpecs()
		raises(PS::ServerError);
	DBConnSpec makeDBConnSpec(in DBConnParam connParam)
		raises(PS::ServerError);
	DBConnSpec findDBConnSpec(in DBConnParam connParam)
		raises(PS::ServerError);
	void removeDBConnSpec(in DBConnSpec connSpec)
		raises(PS::ServerError);
};

// 14.12 PSAdmin::CacheMgr
interface CacheMgr
{
	void clear() raises(PS::ServerError);

	readonly attribute PS::PObject_Cltn allInstances;
	readonly attribute unsigned long numInstances;
	attribute unsigned long numWorkingTransactionInstances;
	attribute unsigned long numWorkingSharedInstances;
	attribute boolean defaultTransLockUponRead;
};

// 14.14 PSAdmin::EventLogMgr
interface EventLogMgr
```

```
	{
		void logEvent(in string category, in string message)
			raises(PS::ServerError);
		void flush() raises(PS::ServerError);

		attribute boolean active;
		attribute string fileName;
	};
};
// ==================================================================
```

The SLdb.idl

```
// ================================================================
// File:    idl/SLdb.idl
// Author:  Persistence(TM) by Persistence Software, Inc.
// ================================================================
// Copyright (C) 1997 Persistence Software, Inc.
// All rights reserved.
// Use of this code without permission is prohibited.
// ================================================================

#ifndef SLDB_IDL
#define SLDB_IDL

#include "PS.idl"

module SLdb
{
    interface buff_class_gene;
    typedef sequence<buff_class_gene> buff_class_gene_Cltn;
    interface classification;
    typedef sequence<classification> classification_Cltn;
    interface genes;
    typedef sequence<genes> genes_Cltn;
    interface genomic_object;
    typedef sequence<genomic_object> genomic_object_Cltn;
    interface kitong;
    typedef sequence<kitong> kitong_Cltn;
    interface synogenes;
    typedef sequence<synogenes> synogenes_Cltn;

    interface buff_class_gene_Factory : PS::PObject_Factory
    {
        buff_class_gene_Cltn allInstances(
            in PS::Defs::ObjectSource onlyInCache);
        buff_class_gene_Cltn query(
            in PS::ClassId_Cltn clsList, in string from,
            in string where, in PS::ArcId_Cltn arcList)
            raises(PS::ServerError);
        buff_class_gene_Cltn queryId_gene(in string val)
            raises(PS::ServerError);
        buff_class_gene_Cltn queryId_category(in string val)
            raises(PS::ServerError);
        buff_class_gene_Cltn queryId_geneRange(
            in string minVal, in string maxVal)
            raises(PS::ServerError);
        buff_class_gene_Cltn queryId_categoryRange(
            in string minVal, in string maxVal)
            raises(PS::ServerError);
        buff_class_gene queryKey(in string id_gene,
            in PS::Defs::ObjectSource onlyInCache)
            raises(PS::ServerError);
        buff_class_gene_Cltn querySP(in string sqStatement)
            raises (PS::ServerError);
        buff_class_gene_Cltn querySQLWhere(in string whereClause)
            raises(PS::ServerError);
    };

    interface buff_class_gene : PS::PObject
    {
        // Attributes
        readonly attribute string id_gene;
        readonly attribute string id_category;
    readonly attribute boolean id_geneNULL;
    readonly attribute boolean id_categoryNULL;

    // Relationships
    readonly attribute genes rel_genes;
    attribute classification rel_classification;
};

interface classification_Factory : PS::PObject_Factory
{
    classification_Cltn allInstances(
        in PS::Defs::ObjectSource onlyInCache);
    classification_Cltn query(in PS::ClassId_Cltn clsList,
        in string from, in string where, in PS::ArcId_Cltn arcList)
        raises(PS::ServerError);
    classification_Cltn queryId_category(in string val)
        raises(PS::ServerError);
    classification_Cltn queryDescription(in string val)
        raises(PS::ServerError);
    classification_Cltn queryId_categoryRange(
        in string minVal, in string maxVal)
        raises(PS::ServerError);
    classification queryKey(in string id_category,
        in PS::Defs::ObjectSource onlyInCache)
        raises(PS::ServerError);
    classification_Cltn querySP(in string sqStatement)
        raises (PS::ServerError);
    classification_Cltn querySQLWhere(in string whereClause)
        raises(PS::ServerError);
};

interface classification : PS::PObject
{
    // Attributes
    readonly attribute string id_category;
    readonly attribute string description;

    readonly attribute boolean id_categoryNULL;
    readonly attribute boolean descriptionNULL;

    // Relationships
    attribute buff_class_gene_Cltn rel_buff_class_gene;
    void addToRel_buff_class_gene(in buff_class_gene relInst)
        raises (PS::ServerError);
    void rmvFromRel_buff_class_gene(in buff_class_gene relInst)
        raises (PS::ServerError);
};

interface genes_Factory : PS::PObject_Factory
{
    genes_Cltn allInstances(in PS::Defs::ObjectSource onlyInCache);
    genes_Cltn query(in PS::ClassId_Cltn clsList,
        in string from, in string where, in PS::ArcId_Cltn arcList)
        raises(PS::ServerError);
    genes_Cltn queryPhysical_map(in float val)
        raises(PS::ServerError);
    genes_Cltn queryEc_number2(in string val)
        raises(PS::ServerError);
    genes_Cltn queryName(in string val)
        raises(PS::ServerError);
    genes_Cltn queryLength_prot(in long val)
        raises(PS::ServerError);
    genes_Cltn queryBrief_descript(in string val)
```

```
        raises(PS::ServerError);
    genes_Cltn queryCodon_usage(in long val)
        raises(PS::ServerError);
    genes_Cltn queryBrief_descript2(in string val)
        raises(PS::ServerError);
    genes_Cltn queryIsoelec_point(in float val)
        raises(PS::ServerError);
    genes_Cltn queryFunction2(in string val)
        raises(PS::ServerError);
    genes_Cltn queryLength(in long val)
        raises(PS::ServerError);
    genes_Cltn queryGenetic_map(in long val)
        raises(PS::ServerError);
    genes_Cltn queryEvidence(in string val)
        raises(PS::ServerError);
    genes_Cltn querySignal_peptide(in long val)
        raises(PS::ServerError);
    genes_Cltn queryMnemonic(in string val)
        raises(PS::ServerError);
    genes_Cltn queryType(in string val)
        raises(PS::ServerError);
    genes_Cltn queryComments2(in string val)
        raises(PS::ServerError);
    genes_Cltn querySp_xref(in string val)
        raises(PS::ServerError);
    genes_Cltn queryRank(in long val)
        raises(PS::ServerError);
    genes_Cltn queryProduct(in string val)
        raises(PS::ServerError);
    genes_Cltn queryComments(in string val)
        raises(PS::ServerError);
    genes_Cltn queryEc_number(in string val)
        raises(PS::ServerError);
    genes_Cltn queryId_gene(in string val)
        raises(PS::ServerError);
    genes_Cltn queryFunction(in string val)
        raises(PS::ServerError);
    genes_Cltn queryText_id(in long val)
        raises(PS::ServerError);
    genes_Cltn queryMol_weight(in float val)
        raises(PS::ServerError);
    genes_Cltn queryPos_kb(in float val)
        raises(PS::ServerError);
    genes_Cltn queryPos_kbRange(in float minVal, in float maxVal)
        raises(PS::ServerError);
    genes_Cltn queryNameRange(in string minVal, in string maxVal)
        raises(PS::ServerError);
    genes_Cltn queryEc_number2Range(in string minVal,
        in string maxVal)
        raises(PS::ServerError);
    genes_Cltn queryLength_protRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    genes_Cltn queryPhysical_mapRange(in float minVal,
        in float maxVal)
        raises(PS::ServerError);
    genes_Cltn queryMol_weightRange(in float minVal, in float maxVal)
        raises(PS::ServerError);
    genes_Cltn queryId_geneRange(in string minVal, in string maxVal)
        raises(PS::ServerError);
    genes_Cltn queryGenetic_mapRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    genes_Cltn queryText_idRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    genes_Cltn queryRankRange(in long minVal, in long maxVal)
```

```
            raises(PS::ServerError);
        genes_Cltn querySignal_peptideRange(in long minVal,
            in long maxVal)
            raises(PS::ServerError);
        genes_Cltn queryEc_numberRange(in string minVal,
            in string maxVal)
            raises(PS::ServerError);
        genes_Cltn queryCodon_usageRange(in long minVal, in long maxVal)
            raises(PS::ServerError);
        genes_Cltn queryLengthRange(in long minVal, in long maxVal)
            raises(PS::ServerError);
        genes_Cltn querySp_xrefRange(in string minVal, in string maxVal)
            raises(PS::ServerError);
        genes_Cltn queryIsoelec_pointRange(in float minVal,
            in float maxVal)
            raises(PS::ServerError);
        genes queryKey(in string id_gene,
            in PS::Defs::ObjectSource onlyInCache)
            raises(PS::ServerError);
        genes_Cltn querySP(in string sqStatement)
            raises (PS::ServerError);
        genes_Cltn querySQLWhere(in string whereClause)
            raises(PS::ServerError);
    };

    interface genes : PS::PObject
    {
        // Attributes
        readonly attribute string id_gene;
        readonly attribute string name;
        readonly attribute string type;
        readonly attribute long length;
        readonly attribute string function;
        readonly attribute string function2;
        readonly attribute string evidence;
        readonly attribute string ec_number;
        readonly attribute string ec_number2;
        readonly attribute string product;
        readonly attribute float pos_kb;
        readonly attribute long genetic_map;
        readonly attribute float physical_map;
        readonly attribute string sp_xref;
        readonly attribute long length_prot;
        readonly attribute long rank;
        readonly attribute long text_id;
        readonly attribute string seq_prot;
        readonly attribute long signal_peptide;
        readonly attribute float mol_weight;
        readonly attribute float isoelec_point;
        readonly attribute string brief_descript;
        readonly attribute string brief_descript2;
        readonly attribute string mnemonic;
        readonly attribute long codon_usage;
        readonly attribute string comments;
        readonly attribute string comments2;

        readonly attribute boolean id_geneNULL;
        readonly attribute boolean nameNULL;
        readonly attribute boolean typeNULL;
        readonly attribute boolean lengthNULL;
        readonly attribute boolean functionNULL;
        readonly attribute boolean function2NULL;
        readonly attribute boolean evidenceNULL;
        readonly attribute boolean ec_numberNULL;
    readonly attribute boolean ec_number2NULL;
    readonly attribute boolean productNULL;
    readonly attribute boolean pos_kbNULL;
    readonly attribute boolean genetic_mapNULL;
    readonly attribute boolean physical_mapNULL;
    readonly attribute boolean sp_xrefNULL;
    readonly attribute boolean length_protNULL;
    readonly attribute boolean rankNULL;
    readonly attribute boolean text_idNULL;
    readonly attribute boolean seq_protNULL;
    readonly attribute boolean signal_peptideNULL;
    readonly attribute boolean mol_weightNULL;
    readonly attribute boolean isoelec_pointNULL;
    readonly attribute boolean brief_descriptNULL;
    readonly attribute boolean brief_descript2NULL;
    readonly attribute boolean mnemonicNULL;
    readonly attribute boolean codon_usageNULL;
    readonly attribute boolean commentsNULL;
    readonly attribute boolean comments2NULL;

    // Relationships
    readonly attribute buff_class_gene rel_buff_class_gene;
    attribute genomic_object_Cltn rel_genomic_object;
    void addToRel_genomic_object(in genomic_object relInst)
        raises (PS::ServerError);
    void rmvFromRel_genomic_object(in genomic_object relInst)
        raises (PS::ServerError);
    readonly attribute synogenes_Cltn rel_synogenes;
};

interface genomic_object_Factory : PS::PObject_Factory
{
    genomic_object_Cltn allInstances(
        in PS::Defs::ObjectSource onlyInCache);
    genomic_object_Cltn query(in PS::ClassId_Cltn clsList,
        in string from, in string where, in PS::ArcId_Cltn arcList)
        raises(PS::ServerError);
    genomic_object_Cltn queryId_gene(in string val)
        raises(PS::ServerError);
    genomic_object_Cltn queryFirst(in long val)
        raises(PS::ServerError);
    genomic_object_Cltn queryDirection(in string val)
        raises(PS::ServerError);
    genomic_object_Cltn queryType(in string val)
        raises(PS::ServerError);
    genomic_object_Cltn queryLast(in long val)
        raises(PS::ServerError);
    genomic_object_Cltn queryFrame(in long val)
        raises(PS::ServerError);
    genomic_object_Cltn queryId_kitong(in string val)
        raises(PS::ServerError);
    genomic_object_Cltn queryPartial(in long val)
        raises(PS::ServerError);
    genomic_object_Cltn queryId_geneRange(in string minVal,
        in string maxVal)
        raises(PS::ServerError);
    genomic_object_Cltn queryLastRange(in long minVal,
        in long maxVal)
        raises(PS::ServerError);
    genomic_object_Cltn queryFrameRange(in long minVal,
        in long maxVal)
        raises(PS::ServerError);
    genomic_object_Cltn queryId_kitongRange(in string minVal,
        in string maxVal)
        raises(PS::ServerError);
    genomic_object_Cltn queryPartialRange(in long minVal,
        in long maxVal)
        raises(PS::ServerError);
    genomic_object_Cltn queryFirstRange(in long minVal,
        in long maxVal)
        raises(PS::ServerError);
    genomic_object queryKey(in string id_kitong,
        in long first, in long last, in string type,
        in PS::Defs::ObjectSource onlyInCache)
        raises(PS::ServerError);
    genomic_object_Cltn querySP(in string sqStatement)
        raises (PS::ServerError);
    genomic_object_Cltn querySQLWhere(in string whereClause)
        raises(PS::ServerError);
};

interface genomic_object : PS::PObject
{
    // Attributes
    readonly attribute string id_kitong;
    readonly attribute long first;
    readonly attribute long last;
    readonly attribute string type;
    readonly attribute long partial;
    readonly attribute string direction;
    readonly attribute long frame;
    readonly attribute string id_gene;

    readonly attribute boolean id_kitongNULL;
    readonly attribute boolean firstNULL;
    readonly attribute boolean lastNULL;
    readonly attribute boolean typeNULL;
    readonly attribute boolean partialNULL;
    readonly attribute boolean directionNULL;
    readonly attribute boolean frameNULL;
    readonly attribute boolean id_geneNULL;

    // Relationships
    readonly attribute kitong rel_kitong;
    attribute genes rel_genes;
};

interface kitong_Factory : PS::PObject_Factory
{
    kitong_Cltn allInstances(in PS::Defs::ObjectSource onlyInCache);
    kitong_Cltn query(in PS::ClassId_Cltn clsList,
        in string from, in string where, in PS::ArcId_Cltn arcList)
        raises(PS::ServerError);
    kitong_Cltn queryPos_kb(in float val)
        raises(PS::ServerError);
    kitong_Cltn queryCreation_date(in PS::Time val)
        raises(PS::ServerError);
    kitong_Cltn queryStatus(in long val)
        raises(PS::ServerError);
    kitong_Cltn queryId_replicon(in string val)
        raises(PS::ServerError);
    kitong_Cltn queryId_kitong(in string val)
        raises(PS::ServerError);
    kitong_Cltn queryPos_bp(in long val)
        raises(PS::ServerError);
    kitong_Cltn queryLength(in long val)
        raises(PS::ServerError);
    kitong_Cltn queryPos_genetic(in long val)
```

```
        raises(PS::ServerError);
    kitong_Cltn queryUpdate_date(in PS::Time val)
        raises(PS::ServerError);
    kitong_Cltn queryText_id(in long val)
        raises(PS::ServerError);
    kitong_Cltn queryText_idRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryUpdate_dateRange(in PS::Time minVal,
        in PS::Time maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryPos_kbRange(in float minVal, in float maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryStatusRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryLengthRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryCreation_dateRange(in PS::Time minVal,
        in PS::Time maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryId_kitongRange(in string minVal,
        in string maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryPos_geneticRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    kitong_Cltn queryPos_bpRange(in long minVal, in long maxVal)
        raises(PS::ServerError);
    kitong queryKey(in string id_kitong,
        in PS::Defs::ObjectSource onlyInCache)
        raises(PS::ServerError);
    kitong_Cltn querySP(in string sqStatement)
        raises (PS::ServerError);
    kitong_Cltn querySQLWhere(in string whereClause)
        raises(PS::ServerError);
};

interface kitong : PS::PObject
{
    // Attributes
    readonly attribute string id_kitong;
    readonly attribute long length;
    readonly attribute string id_replicon;
    readonly attribute long pos_bp;
    readonly attribute float pos_kb;
    readonly attribute long pos_genetic;
    readonly attribute PS::Time creation_date;
    readonly attribute PS::Time update_date;
    readonly attribute long status;
    readonly attribute long text_id;
    readonly attribute string nuc_seq;

    readonly attribute boolean id_kitongNULL;
    readonly attribute boolean lengthNULL;
    readonly attribute boolean id_repliconNULL;
    readonly attribute boolean pos_bpNULL;
    readonly attribute boolean pos_kbNULL;
    readonly attribute boolean pos_geneticNULL;
    readonly attribute boolean creation_dateNULL;
    readonly attribute boolean update_dateNULL;
    readonly attribute boolean statusNULL;
    readonly attribute boolean text_idNULL;
    readonly attribute boolean nuc_seqNULL;

    // Relationships
    readonly attribute genomic_object_Cltn rel_genomic_object;
    };

    interface synogenes_Factory : PS::PObject_Factory
    {
        synogenes_Cltn allInstances(
            in PS::Defs::ObjectSource onlyInCache);
        synogenes_Cltn query(in PS::ClassId_Cltn clsList,
            in string from, in string where, in PS::ArcId_Cltn arcList)
            raises(PS::ServerError);
        synogenes_Cltn queryId_gene(in string val)
            raises(PS::ServerError);
        synogenes_Cltn queryAlias(in string val)
            raises(PS::ServerError);
        synogenes_Cltn queryAliasRange(in string minVal,
            in string maxVal)
            raises(PS::ServerError);
        synogenes_Cltn queryId_geneRange(in string minVal,
            in string maxVal)
            raises(PS::ServerError);
        synogenes queryKey(in string id_gene, in string alias,
            in PS::Defs::ObjectSource onlyInCache)
            raises(PS::ServerError);
        synogenes_Cltn querySP(in string sqStatement)
            raises (PS::ServerError);
        synogenes_Cltn querySQLWhere(in string whereClause)
            raises(PS::ServerError);
    };

    interface synogenes : PS::PObject
    {
        // Attributes
        readonly attribute string id_gene;
        readonly attribute string alias;

        readonly attribute boolean id_geneNULL;
        readonly attribute boolean aliasNULL;

        // Relationships
        readonly attribute genes rel_genes;
    };
};
#endif
```

Appendix C. The bacterial genomes object model

An object model of bacterial genomes described in Section 5.2 is presented here in the UML notation. An overall view of the main and interconnecting objects of feature, sequence and location information is depicted in Figure C-1. The complete model is organised into packages, which are depicted in Figures C-2 and C-4. The overall model of feature, sequence and location information is shown in Figure C-3. Figures C-5, C-6 and C-7 focus on the representation of feature information at the DNA, RNA and protein levels respectively, while Figure C-8 shows some extra nucleotide sequence information.

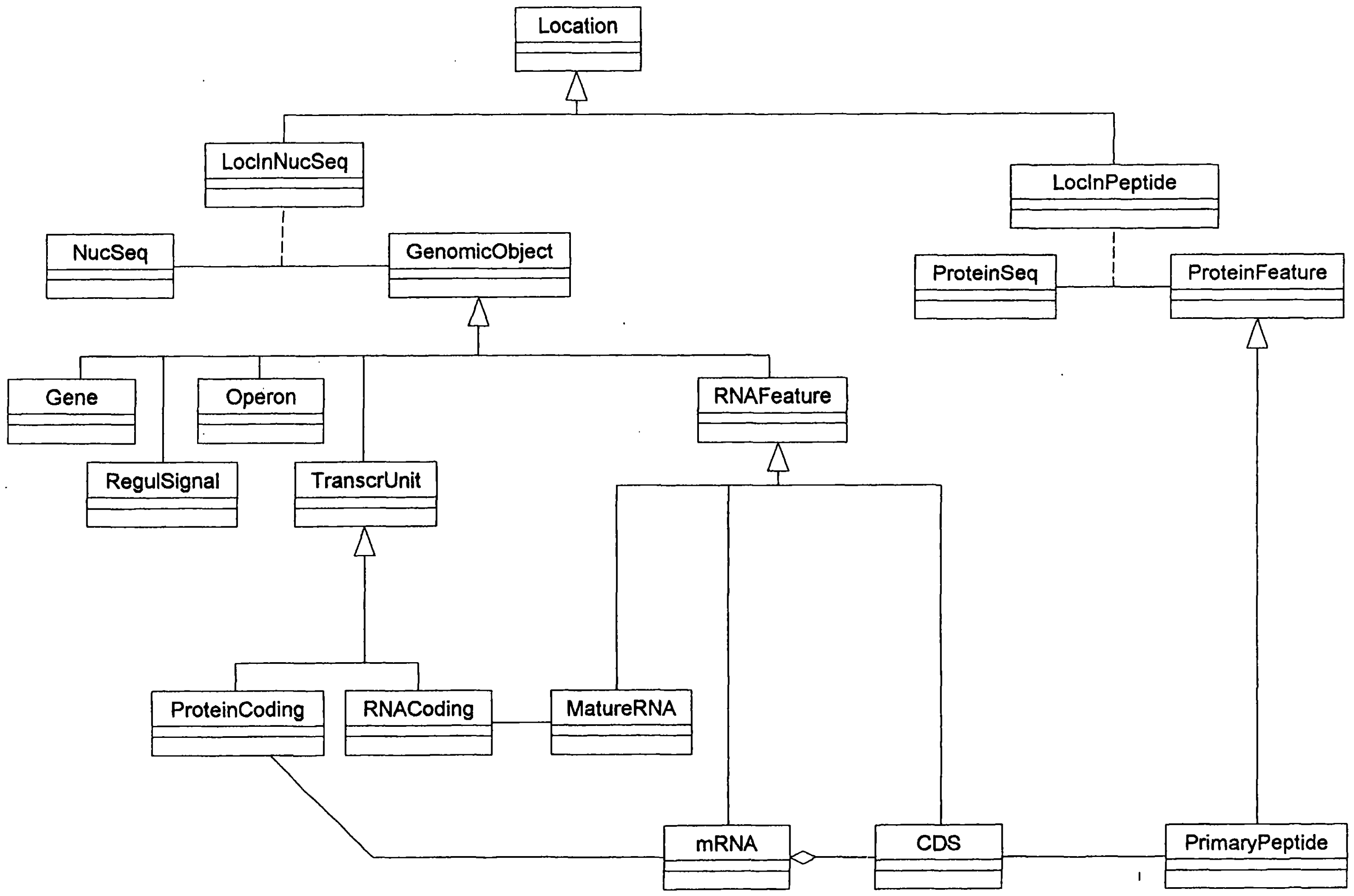

Figure C-1: Overall model of main and interconnecting objects of feature, sequence and location information

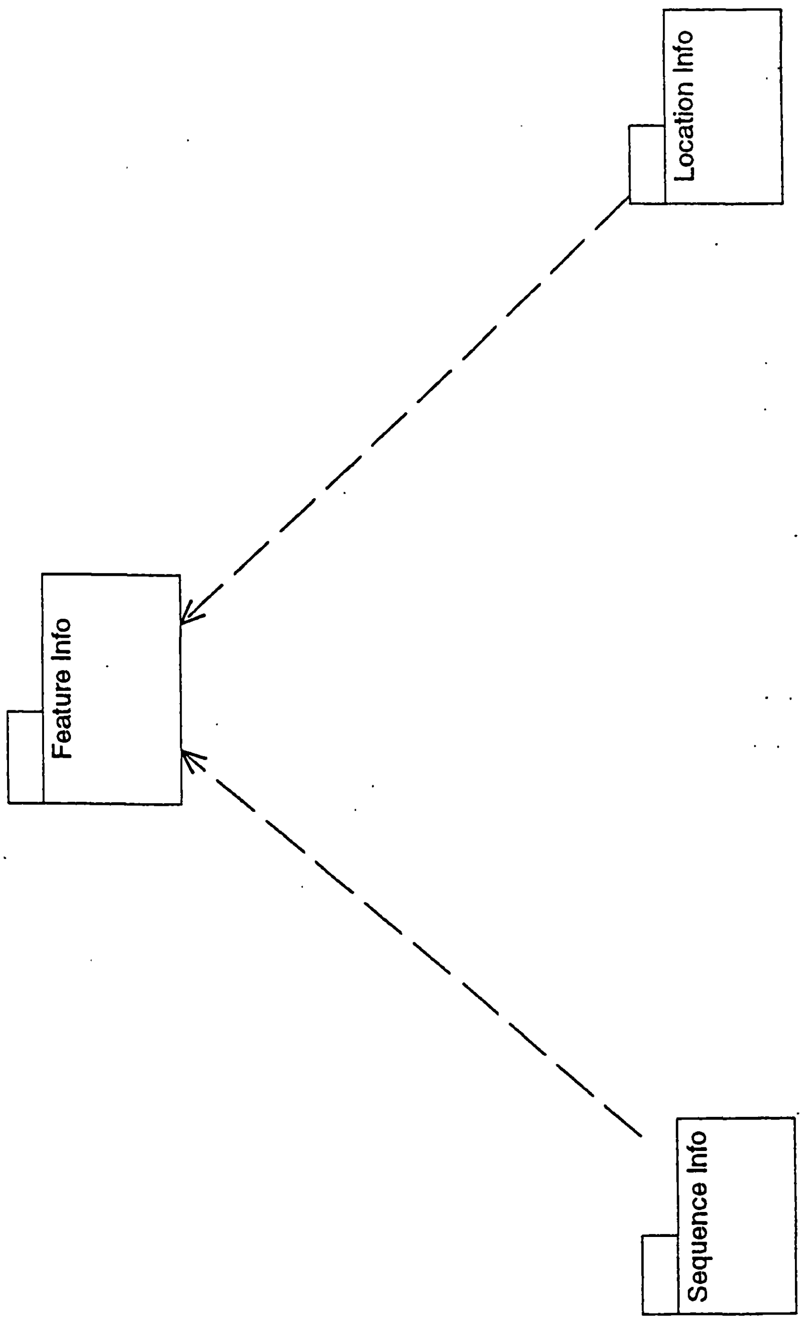

Figure C-2: Container packages for feature, sequence and location information

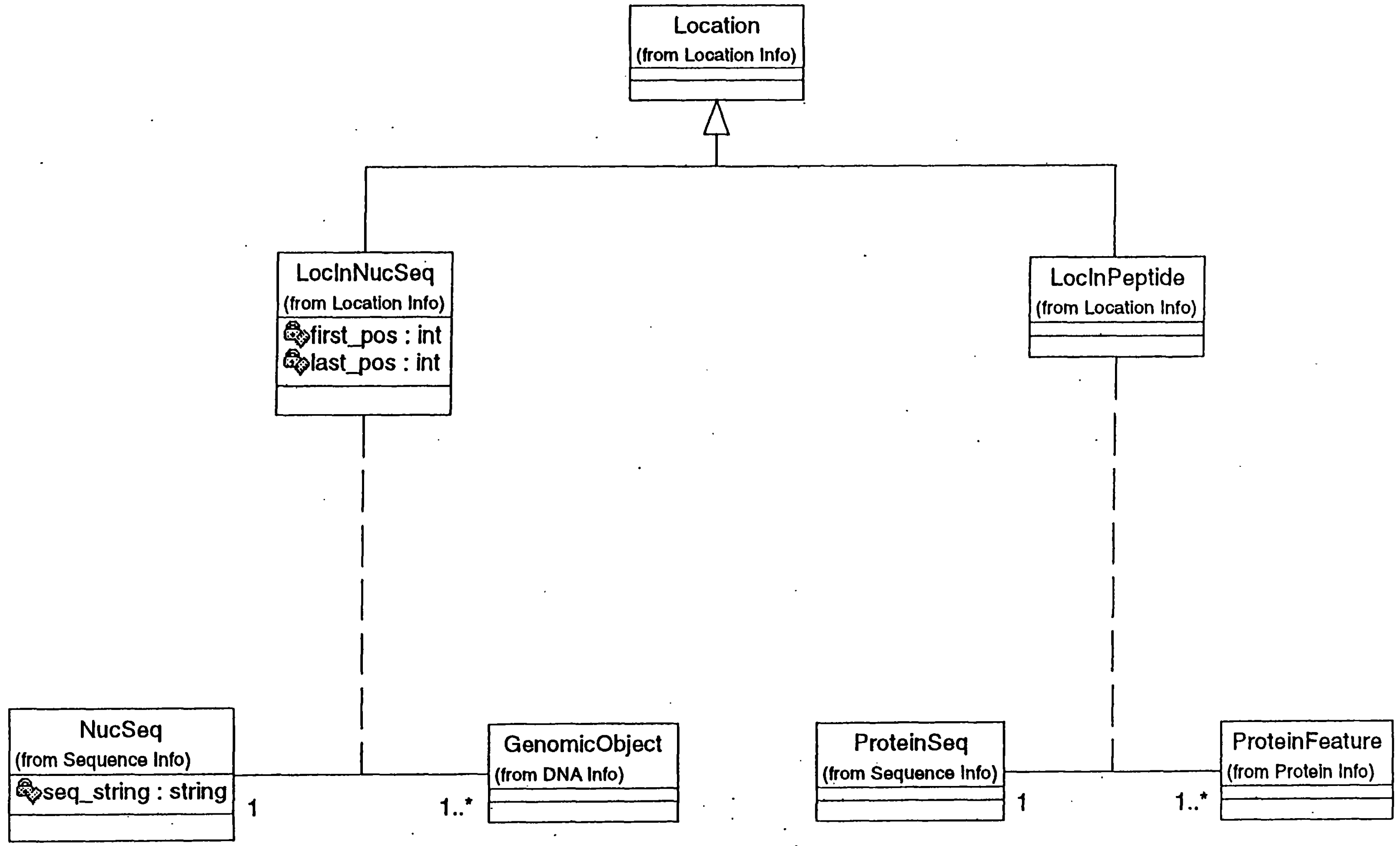

Figure C-3: Overall model of feature, sequence and location information

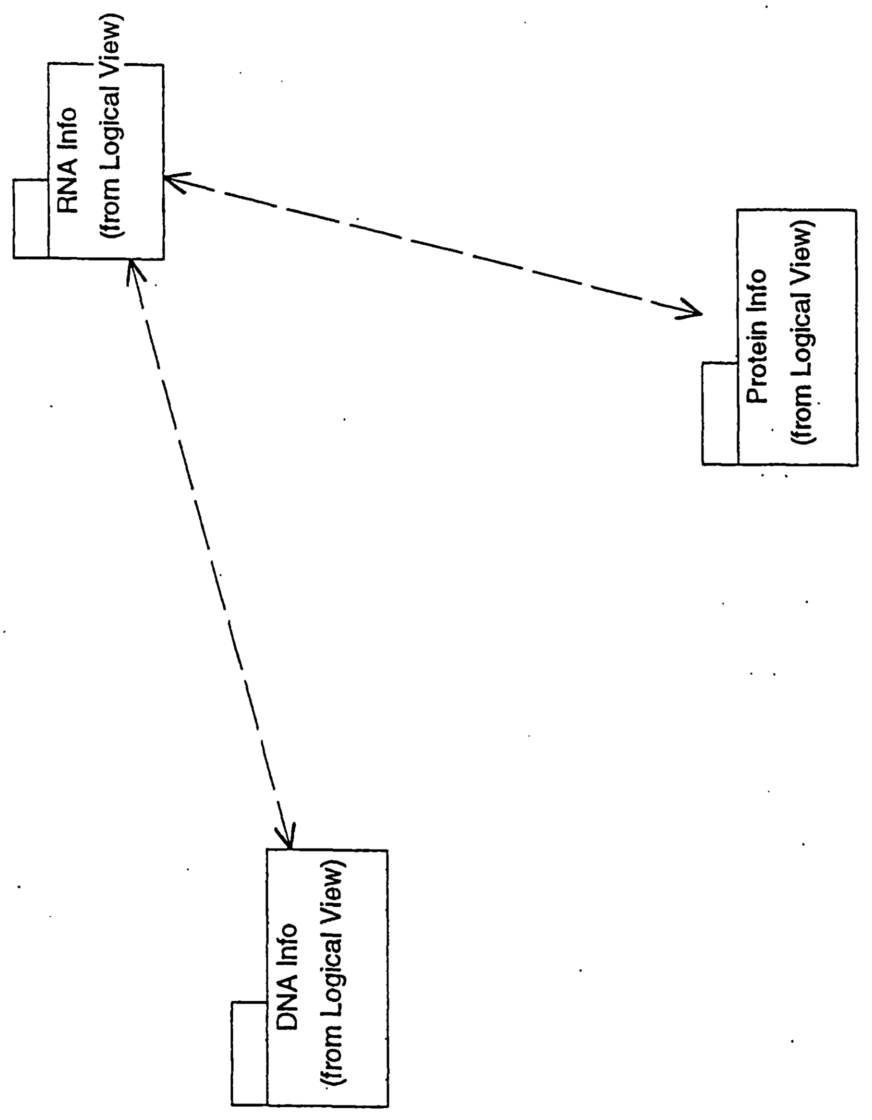

Figure C-4: Container packages for information at the DNA, RNA and protein levels

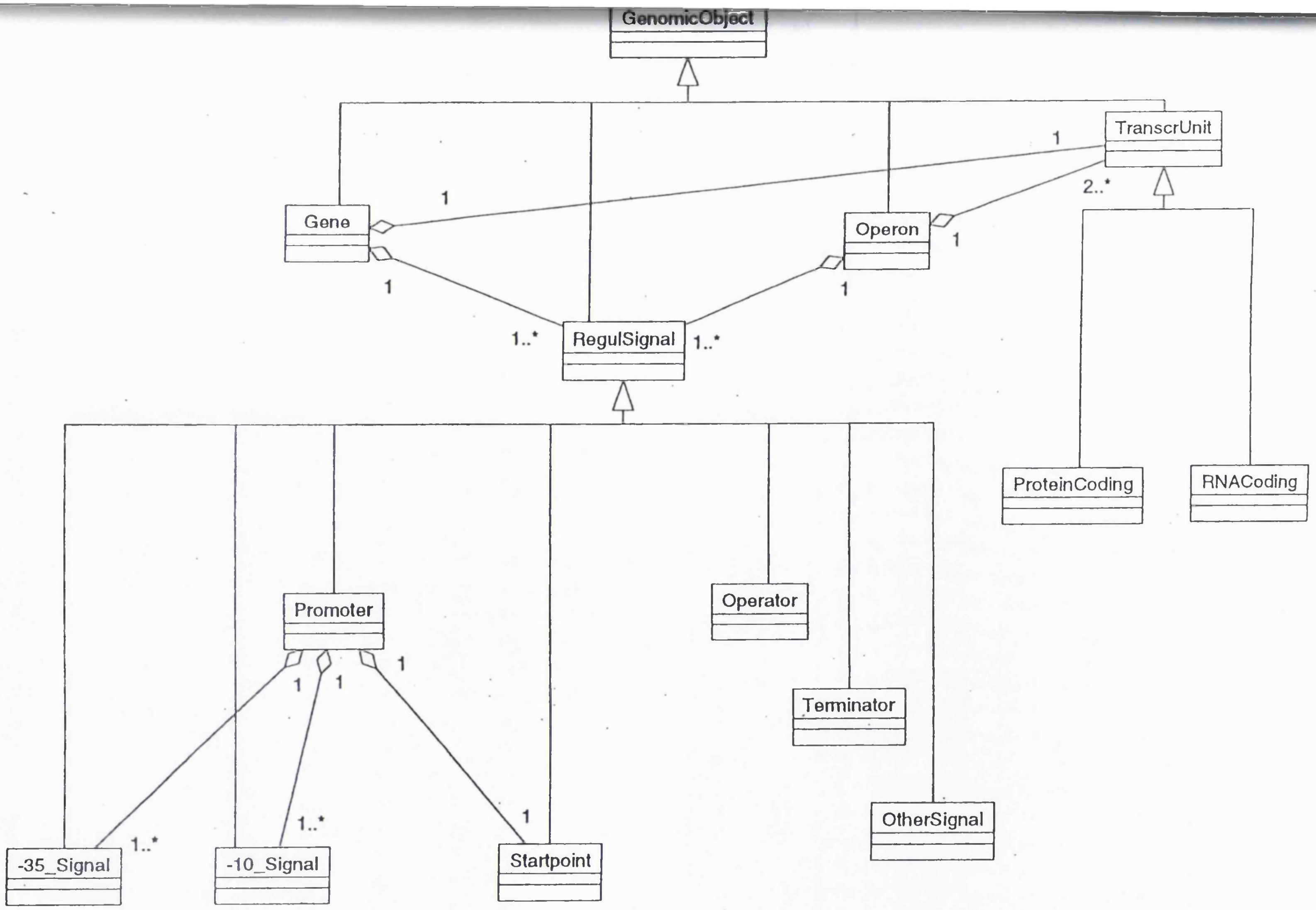

Figure C-5: Genomic objects at DNA level

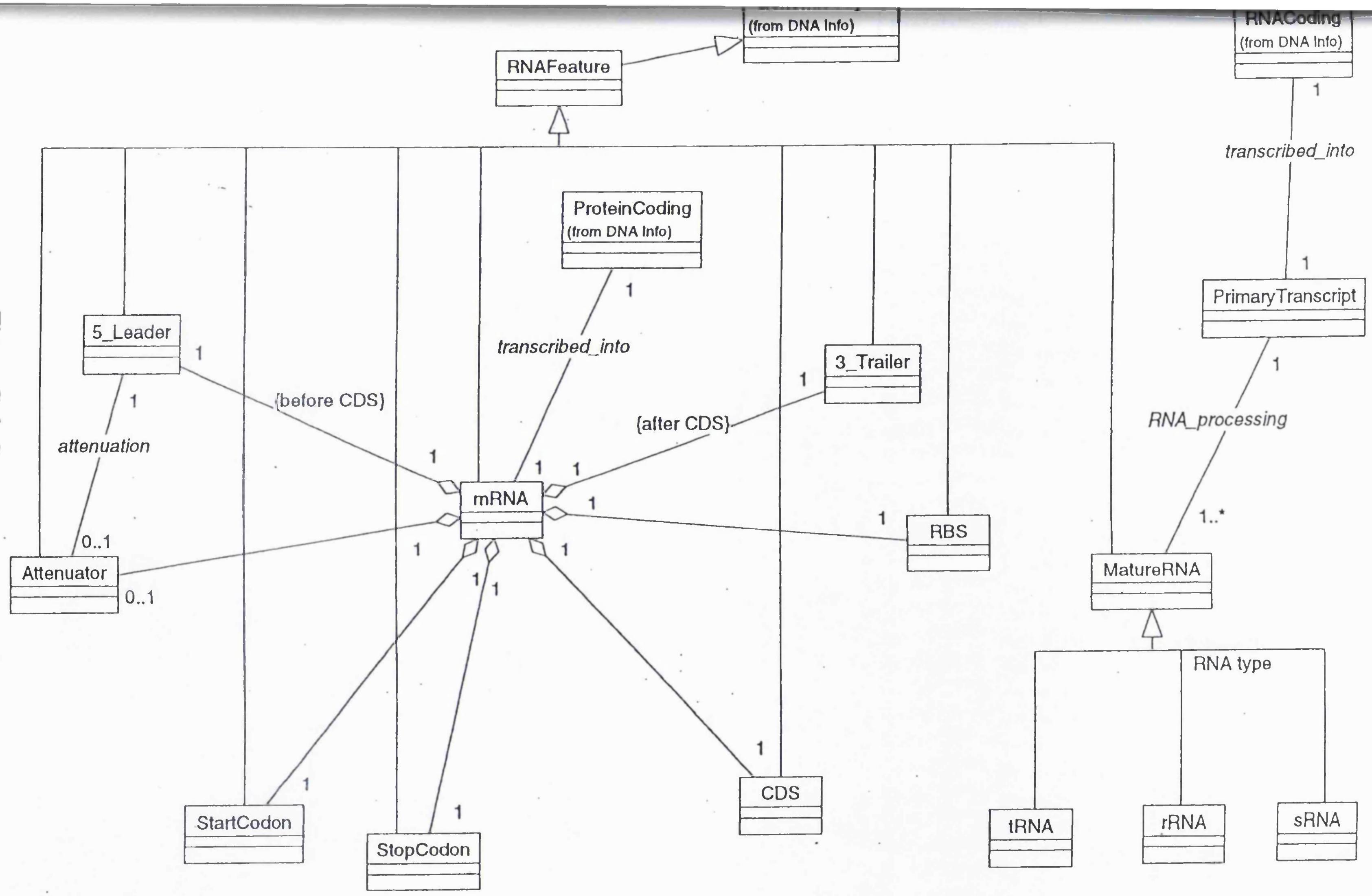

Figure C-6: Genomic objects at RNA level

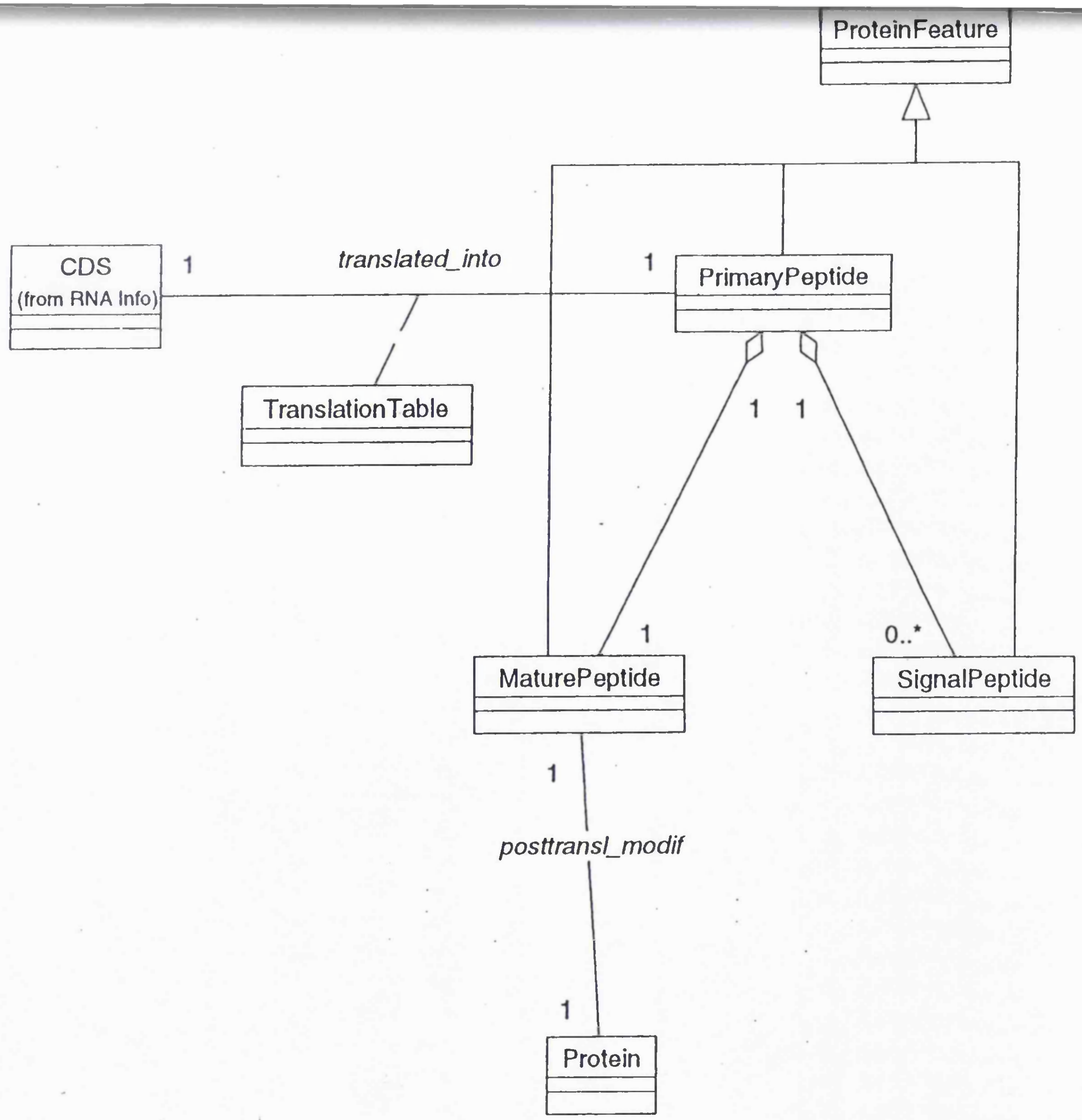

Figure C-7: Protein features

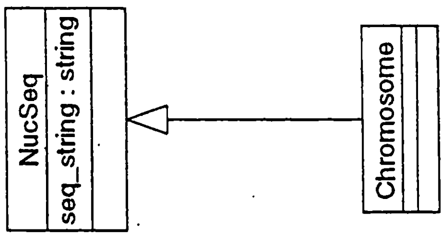

Figure C-8: Nucleotide sequence

Glossary for the Bacterial Genomes model

3_Trailer: A number of nucleotides that follow a **CDS**.

5_Leader: A number of nucleotides that precede a **CDS**.

-10_Signal: Unwinding domain of a **Promoter**.

-35_Signal: Recognition domain of a **Promoter**.

Attenuator: It controls whether or not transcription will go on.

CDS: Protein coding nucleotide sequence.

Gene: A region of DNA encoding for proteins/RNAs.

MaturePeptide: The peptide that remains after the leading **SignalPeptide** is removed.

MatureRNA: Mature and stable RNA that could be any of **tRNA**, **rRNA**, or **sRNA** types.

Operator: A region of DNA that acts as a "traffic light", a go or stop signal for transcription. It is usually positioned before the **TranscrUnit**. Positioning of the **Operator** relative to the **Promoter** may vary in different cases.

Operon: A group of genes regulated coordinately.

PrimaryPeptide: Amino-acid sequence; the result of **CDS** translation.

PrimaryTranscript: RNA sequence that is cleaved post-transcriptionally to yield the mature RNA products.

Promoter: The region of DNA where RNA polymerase initiates transcription. It contains the **Startpoint** where transcription begins. It is usually positioned before the **TranscrUnit**.

RBS: (Ribosome Binding Site) Usually 5-10 nucleotides before the initiation codon where certain short sequences of the 16S RNA (part of the Ribosome) bind. RBS could also be after the initiation codon.

SignalPeptide: It is a "leader" peptide that directs proteins to specific places. In bacteria, signal peptides direct proteins across the bacterial plasma membrane. Signal peptides are usually removed from the protein after the sorting process has been completed.

StartCodon: The region of **mRNA** where translation begins. It is translated but later removed. Start codons found in *Bacillus subtilis*: AUG, UUG, GUG, AUU, CUG [Kunst 97].

Startpoint: The region of DNA where transcription begins.

StopCodon: The region of **mRNA** where translation ends. Examples of stop codons: UAA, UGA, UAG.

sRNA: Small cytoplasmic RNA.

Terminator: The region of DNA where transcription ends. It is positioned after the **TranscrUnit**.

TranscrUnit: The region of DNA to be transcribed.

TranslationTable: It is generally universally applied, however there are some small differences in different species.

Appendix D. Generated mediator code

```
// File BacterialGenome.idl

#include "/embl/idl/types.idl"
#include "/embl/idl/nsdb.idl"
#include "/sldb/idl/PS.idl"

module BacterialGenome {
interface GeneFactory;
interface GenomicObject;
interface FunctionalClass;
interface Gene;
interface Promoter;
interface Minus10Signal;
interface Minus35Signal;
interface Terminator;
interface ProteinCoding;

interface GeneFactory {
  BacterialGenome::Gene getById(in string id, in string embl_id)
    raises (type::NoResult,
            nsdb::Embl(interface)::Superceded,
            PS::ServerError);
};

interface GenomicObject {
  long firstPos();
  long lastPos();
  string nucSeq();
  boolean directionIsL2R();
};

interface FunctionalClass {
  string id();
  string description();
};

interface Gene {
  typedef sequence<BacterialGenome::Promoter> PromoterList;
  typedef sequence<BacterialGenome::Minus10Signal> Minus10SignalList;
  typedef sequence<BacterialGenome::Minus35Signal> Minus35SignalList;
  typedef sequence<BacterialGenome::Terminator> TerminatorList;
  typedef sequence<BacterialGenome::ProteinCoding> ProteinCodingList;
  typedef sequence<string> StringList;
  string id();
  string name();
  string ecNumber();
  string function();
  string description();
  FunctionalClass functionalClass() raises (PS::ServerError);
  string codonUsageClass();
  long pos();
  PromoterList getPromoter() raises (PS::ServerError);
  Minus10SignalList getMinus10Signal() raises (PS::ServerError);
  Minus35SignalList getMinus35Signal() raises (PS::ServerError);
  TerminatorList getTerminator() raises (PS::ServerError);
  ProteinCodingList getProteinCoding() raises (PS::ServerError);
  StringList references();
};

interface Promoter : GenomicObject {
};

interface Minus10Signal : GenomicObject {
};

interface Minus35Signal : GenomicObject {
};

interface Terminator : GenomicObject {
};

interface ProteinCoding : GenomicObject {
};

};
```

```
// File GeneFactoryImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class GeneFactoryImpl implements _GeneFactoryOperations {

   // Constructor
   public GeneFactoryImpl() {
   }

   public BacterialGenome.Gene getById(String id, String embl_id)
     throws type.NoResult, nsdb.EmblPackage.Superceded, PS.ServerError {
     SLdb.genes g = null;
     g = (Server.geneF.querySQLWhere("id_gene = '" + id + "'"))[0];
     SLdb.genomic_object[] go = null;
     go = (Server.genobjF.querySQLWhere("id_gene = '" + id + "'"));
     nsdb.EmblSeq es = null;
     es = Server.emblF.getEmblSeq(embl_id);
     BacterialGenome.Gene _ivg_GeneTie = new _tie_Gene(
       new GeneImpl(g, go, es));

     return( _ivg_GeneTie );
   }

}
```

```
// File GenomicObjectImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class GenomicObjectImpl implements _GenomicObjectOperations {
   SLdb.genomic_object go;
   SLdb.kitong k;

   // Constructor
   public GenomicObjectImpl(SLdb.genomic_object go, SLdb.kitong k) {
      this.go = go;
      this.k = k;
   }

   public int firstPos() {
     return( (int)(k.pos_kb()*1000) +
       go.direction().startsWith("+") ? go.first() : go.last() );
   }

   public int lastPos() {
     return( (int)(k.pos_kb()*1000) +
       go.direction().startsWith("+") ? go.last() : go.first() );
   }

   public String nucSeq() {
     return( myNucSeq() );
   }

   public boolean directionIsL2R() {
     return( go.direction().startsWith("+") );
   }

      String myNucReverse(String nseq) {
        String rev="";
        for (int i=0; i<nseq.length(); i++)
          rev = rev + nseq.charAt(nseq.length()-1-i);
        return rev;
      }

      String myNucComplement(String nseq) {
        String compl="";
        for (int i=0; i<nseq.length(); i++)
          switch (nseq.charAt(i)) {
            case 'a': compl = compl + "t"; break;
            case 'c': compl = compl + "g"; break;
            case 'g': compl = compl + "c"; break;
            case 't': compl = compl + "a"; break;
            default : compl = compl + "?"; break;
          }
        return compl;
      }

      String myNucSeq() {
        if (go.direction().startsWith("+"))
          return ( k.nuc_seq().substring(go.first(), go.last()) );
        else
          return myNucReverse(myNucComplement(
                 k.nuc_seq().substring(go.first(), go.last()) ));
      }
 }
```

```
// File FunctionalClassImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class FunctionalClassImpl implements _FunctionalClassOperations {
   SLdb.buff_class_gene b;
   SLdb.classification c;

   // Constructor
   public FunctionalClassImpl(SLdb.buff_class_gene b,
      SLdb.classification c) {
      this.b = b;
      this.c = c;
   }

   public String id() {
     return( this.b.id_category() );
   }

   public String description() {
     return( myClassDescription() );
   }

      String myClassDescription() {
        switch (c.id_category()) {
          case "3.5.1" : return("Information pathways; RNA synthesis; " +
                         c.description());
          case "3.5.2" : return("Information pathways; RNA synthesis; " +
                         c.description());
          case "6": return c.description();
          default : return c.description();
        }
      }
}
```

```
// File GeneImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class GeneImpl implements _GeneOperations {
   SLdb.genes g;
   SLdb.genomic_object[] go;
   nsdb.EmblSeq es;

   // Constructor
   public GeneImpl(SLdb.genes g, SLdb.genomic_object[] go,
      nsdb.EmblSeq es) {
      this.g = g;
      this.go = go;
      this.es = es;
   }

   public String id() {
     return( this.g.id_gene() );
   }

   public String name() {
     return( this.g.name() );
   }

   public String ecNumber() {
     return( this.g.ec_number() );
   }

   public String function() {
     return( this.g.function() );
   }

   public String description() throws type.NoResult {
     return( this.es.getDescription() );
   }

   public BacterialGenome.FunctionalClass functionalClass()
     throws PS.ServerError {
     SLdb.buff_class_gene b = null;
     b = (Server.classgeneF.querySQLWhere("id_gene = '" + name() +
       "'"))[0];
     SLdb.classification c = null;
     c = (Server.classF.querySQLWhere("id_category = '" +
       b.id_category() + "'"))[0];
     BacterialGenome.FunctionalClass _ivg_FunctionalClassTie =
       new _tie_FunctionalClass(new FunctionalClassImpl(b, c));

     return( _ivg_FunctionalClassTie );
   }

   public String codonUsageClass() {
     return( myCodonUsageClass() );
   }

   public int pos() {
     return( (int)(g.pos_kb()*1000) );
   }

   public BacterialGenome.Promoter[] getPromoter()
     throws PS.ServerError {
     SLdb.genomic_object[] _ivg_goL = null;
```

```
  SLdb.genomic_object go = null;
  SLdb.kitong k = null;

  _ivg_goL = (Server.genobjF.querySQLWhere(
    "(type = 'promoter') and (id_gene = '" + id() + "')"));
  BacterialGenome.Promoter[] _ivg_PromoterTieL =
    new Promoter[_ivg_goL.length];
  for (int i=0; i<_ivg_goL.length; i++) {
    go = _ivg_goL[i];
    k = (Server.dnafragF.querySQLWhere("id_kitong = '" +
      go.id_kitong() + "'"))[0];
    _ivg_PromoterTieL[i] = new _tie_Promoter(new PromoterImpl(go, k));
  }
  return( _ivg_PromoterTieL );
}

public BacterialGenome.Minus10Signal[] getMinus10Signal()
  throws PS.ServerError {
  SLdb.genomic_object[] _ivg_goL = null;
  SLdb.genomic_object go = null;
  SLdb.kitong k = null;

  _ivg_goL = (Server.genobjF.querySQLWhere(
    "(type = '-10_signal') and (id_gene = '" + id() + "')"));
  BacterialGenome.Minus10Signal[] _ivg_Minus10SignalTieL =
    new Minus10Signal[_ivg_goL.length];
  for (int i=0; i<_ivg_goL.length; i++) {
    go = _ivg_goL[i];
    k = (Server.dnafragF.querySQLWhere("id_kitong = '" +
      go.id_kitong() + "'"))[0];
    _ivg_Minus10SignalTieL[i] = new _tie_Minus10Signal(
      new Minus10SignalImpl(go, k));
  }
  return( _ivg_Minus10SignalTieL );
}

public BacterialGenome.Minus35Signal[] getMinus35Signal()
  throws PS.ServerError {
  SLdb.genomic_object[] _ivg_goL = null;
  SLdb.genomic_object go = null;
  SLdb.kitong k = null;

  _ivg_goL = (Server.genobjF.querySQLWhere(
    "(type = '-35_signal') and (id_gene = '" + id() + "')"));
  BacterialGenome.Minus35Signal[] _ivg_Minus35SignalTieL =
    new Minus35Signal[_ivg_goL.length];
  for (int i=0; i<_ivg_goL.length; i++) {
    go = _ivg_goL[i];
    k = (Server.dnafragF.querySQLWhere("id_kitong = '" +
      go.id_kitong() + "'"))[0];
    _ivg_Minus35SignalTieL[i] = new _tie_Minus35Signal(
      new Minus35SignalImpl(go, k));
  }
  return( _ivg_Minus35SignalTieL );
}

public BacterialGenome.Terminator[] getTerminator()
  throws PS.ServerError {
  SLdb.genomic_object[] _ivg_goL = null;
  SLdb.genomic_object go = null;
  SLdb.kitong k = null;

  _ivg_goL = (Server.genobjF.querySQLWhere(
    "(type = 'terminator') and (id_gene = '" + id() + "')"));
    BacterialGenome.Terminator[] _ivg_TerminatorTieL =
      new Terminator[_ivg_goL.length];
    for (int i=0; i<_ivg_goL.length; i++) {
      go = _ivg_goL[i];
      k = (Server.dnafragF.querySQLWhere("id_kitong = '" +
        go.id_kitong() + "'"))[0];
      _ivg_TerminatorTieL[i] = new _tie_Terminator(
        new TerminatorImpl(go, k));
    }
    return( _ivg_TerminatorTieL );
  }

  public BacterialGenome.ProteinCoding[] getProteinCoding()
    throws PS.ServerError {
    SLdb.genomic_object[] _ivg_goL = null;
    SLdb.genomic_object go = null;
    SLdb.kitong k = null;

    _ivg_goL = (Server.genobjF.querySQLWhere(
      "(type = 'CDS') and (id_gene = '" + id() + "')"));
    BacterialGenome.ProteinCoding[] _ivg_ProteinCodingTieL =
      new ProteinCoding[_ivg_goL.length];
    for (int i=0; i<_ivg_goL.length; i++) {
      go = _ivg_goL[i];
      k = (Server.dnafragF.querySQLWhere("id_kitong = '" +
        go.id_kitong() + "'"))[0];
      _ivg_ProteinCodingTieL[i] = new _tie_ProteinCoding(
        new ProteinCodingImpl(go, k));
    }
    return( _ivg_ProteinCodingTieL );
  }

  public String[] references()
    throws type.NoResult, type.InvalidArgumentValue {
    return( myRefs() );
  }

     String[] myRefs() throws type.NoResult, type.InvalidArgumentValue {
       bibliography.Reference   ref=null;
       type.DbXref[]            dbRefL;
       String[]                 strL=null;
       biblio.BiblioFormatter   bibFormatter=null;

       bibFormatter = new biblio.BiblioFormatter();
       dbRefL = es.getReferences();
       strL=new String[dbRefL.length];
       for (int i=0; i<dbRefL.length; i++) {
         ref = Server.libF.getReference(dbRefL[i].primary_id);
         strL[i] = bibFormatter.formatBiblioRef(ref);
          }
       return strL;
     }

     String myCodonUsageClass() {
       switch (g.codon_usage()) {
         case 0: return "Not a protein coding gene";
         case 1: return "Class 1";
         case 2: return
                 "Class 2: High expression in exponential growth";
         case 3: return "Class 3: Prophages";
         default: return "Unknown codon usage class";
       }
     }
}
```

```
// File PromoterImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class PromoterImpl extends GenomicObjectImpl
   implements _PromoterOperations {

   // Constructor
   public PromoterImpl(SLdb.genomic_object go, SLdb.kitong k) {
     super(go, k);
   }

}
```

```
// File Minus10SignalImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class Minus10SignalImpl extends GenomicObjectImpl
   implements _Minus10SignalOperations {

   // Constructor
   public Minus10SignalImpl(SLdb.genomic_object go, SLdb.kitong k) {
     super(go, k);
   }

}
```

```
// File Minus35SignalImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class Minus35SignalImpl extends GenomicObjectImpl
   implements _Minus35SignalOperations {

   // Constructor
   public Minus35SignalImpl(SLdb.genomic_object go, SLdb.kitong k) {
     super(go, k);
   }

}
```

```
// File TerminatorImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class TerminatorImpl extends GenomicObjectImpl
   implements _TerminatorOperations {

   // Constructor
   public TerminatorImpl(SLdb.genomic_object go, SLdb.kitong k) {
     super(go, k);
   }

}
```

```
// File ProteinCodingImpl.java

package BacterialGenome;

import BacterialGenome.Server;

class ProteinCodingImpl extends GenomicObjectImpl
   implements _ProteinCodingOperations {

   // Constructor
   public ProteinCodingImpl(SLdb.genomic_object go, SLdb.kitong k) {
     super(go, k);
   }

}
```

```
// File Server.java

package BacterialGenome;

import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb._OrbixWeb;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import java.net.*;
import java.io.*;
import java.lang.String;
import java.util.Vector;

public class Server {

  // Factory classes.
  public static SLdb.buff_class_gene_Factory buffclassgeneF = null;
  public static SLdb.classification_Factory classF = null;
  public static SLdb.genes_Factory geneF = null;
  public static SLdb.genomic_object_Factory genobjF = null;
  public static SLdb.kitong_Factory dnafragF = null;
  public static nsdb.Embl emblF = null;
  public static meta.Controled ctrlF = null;
  public static bibliography.ReferenceLibrary libF = null;

  private static final String server_name = "BactGen_AutoServer";

  // Constructor.
  public Server() {
  };

  // Obtain an Object Reference.
  public Object retrieveIORFromURL(ORB orb, String urlName) {
    Object      obj=null;

    try {
      URL url = new URL(urlName);
      BufferedReader inBuf = new BufferedReader(
        new InputStreamReader(url.openStream()));
      String objStr = inBuf.readLine();
      inBuf.close();

      obj = orb.string_to_object(objStr);
    }
    catch(SystemException sysEx) {
      System.err.println(
        "***CORBA.SystemException during string_to_object: ");
      System.err.println(sysEx.toString());
      System.exit(1);
    }
    catch(IOException ioEx) {
      System.err.println("***Java.IOException; Reading IOR failed: ");
      System.err.println(ioEx.toString());
      System.exit(1);
    }
    return obj;
  }

  // Get factory objects.
  public int getFactories(ORB orb) {
    try {
      classgeneF = SLdb.buff_class_gene_FactoryHelper.narrow(
        this.retrieveIORFromURL(orb, "file:/sldb/ior/cg_Fact.ior"));
      classF = SLdb.classification_FactoryHelper.narrow(
        this.retrieveIORFromURL(orb, "file:/sldb/ior/c_Fact.ior"));
      geneF = SLdb.genes_FactoryHelper.narrow(
        this.retrieveIORFromURL(orb, "file:/sldb/ior/g_Fact.ior"));
      genobjF = SLdb.genomic_object_FactoryHelper.narrow(
        this.retrieveIORFromURL(orb, "file:/sldb/ior/go_Fact.ior"));
      dnafragF = SLdb.kitong_FactoryHelper.narrow(
        this.retrieveIORFromURL(orb, "file:/sldb/ior/df_Fact.ior"));
      emblF = nsdb.EmblHelper.narrow(
        this.retrieveIORFromURL(orb,
          "http://corba.ebi.ac.uk/EMBL/IOR/Embl.IOR"));
      ctrlF = meta.ControledHelper.narrow(
        this.retrieveIORFromURL(orb,
          "http://corba.ebi.ac.uk/EMBL/IOR/Meta.IOR"));
      libF = bibliography.ReferenceLibraryHelper.narrow(
        this.retrieveIORFromURL(orb,
          "http://corba.ebi.ac.uk/EMBL/IOR/Reference.IOR"));
    }
    catch(SystemException sysEx) {
      System.err.println("***CORBA.SystemException during narrow: ");
      System.err.println(sysEx.toString());
      System.exit(1);
    }
    return 0;
  }

  // Publicise an Object Reference using the CORBA standard
object_to_string.
  public int publiciseObjRef(Object obj, String urlName, ORB orb) {
    try {
      String str = orb.object_to_string(obj);

      URL url = new URL(urlName);
      String fileName = url.getFile();
      FileWriter outFile = new FileWriter(fileName);
      outFile.write(str, 0, str.length());
      outFile.close();
    }
    catch(IOException ioEx) {
      System.err.println("***Java.IOException; Writing ObjRef failed: ");
      System.err.println(ioEx.toString());
      System.exit(1);
    }
    catch(SystemException sysEx) {
      System.err.println(
        "***CORBA.SystemException during object_to_string: ");
      System.err.println(sysEx.toString());
      System.exit(1);
    }
    return 0;
  }

  // Main.
  public static void main(String args[]) {
    Server server = null;
    ORB orb = null;

    try {
      // Initialise the ORB.
      orb = ORB.init();

      // Create a Server object.
      server = new Server();
      // Obtain ObjRefs of factory objects.
      if (server.getFactories(orb) !=0)
        return;

      // Set the server name before making object references public.
      _CORBA.Orbix.setServerName(server_name);

      // Create objects to be published and publish their IORs.
      BactGen.GeneFactory vGeneFactoryTie =
        new _tie_GeneFactory(new GeneFactoryImpl());
      if (server.publiciseObjRef(vGeneFactoryTie,
        "file:/tmp/BactGen.GeneFactory.ior", orb) != 0)
        return;

      _CORBA.Orbix.impl_is_ready(server_name);

      // Disconnect published objects.
      _CORBA.Orbix.disconnect(vGeneFactoryTie);
    }
    catch(SystemException sysEx) {
      System.err.println(sysEx.toString());
      System.exit(1);
    }
  }
}
```

Appendix E. Client application code

```
// Client.java

package BacterialGenome;

import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import java.net.*;
import java.io.*;
import java.lang.String;

public class Client {

  public static BacterialGenome.GeneFactory geneFactory = null;
  public static nsdb.Embl emblFactory = null;

  // Constructor
  public Client() {
  };

  public Object retrieveIORFromURL(ORB orb, String urlname) {
    Object   obj=null;

    // Obtain an IOR.
    try {
      URL url = new URL (urlname);
      BufferedReader inBuf = new BufferedReader(new InputStreamReader(
                             url.openStream()));
      String objStr = inBuf.readLine();
      inBuf.close();

      obj = orb.string_to_object(objStr);
    }
    catch(SystemException sysEx) {
      System.err.println("Exception during string_to_object:");
      System.err.println(sysEx.toString());
      System.exit(1);
    }
    catch(IOException ioEx) {
      System.err.println("Reading IOR failed:");
      System.err.println(ioEx.toString());
      System.exit(1);
    }
    return obj;
  }

  public Object retrieveIORFromFile(ORB orb, String fileName) {
    Object   obj=null;

    // Obtain an IOR.
    try {
      BufferedReader inBuf = new BufferedReader(
        new FileReader(fileName));
      String objStr = inBuf.readLine();
      inBuf.close();

      obj = orb.string_to_object(objStr);
    }
    catch(SystemException sysEx) {
      System.err.println("Exception during string_to_object:");
      System.err.println(sysEx.toString());
      System.exit(1);
    }
    catch(IOException ioEx) {
      System.err.println("Reading IOR failed:");
      System.err.println(ioEx.toString());
      System.exit(1);
    }
    return obj;
  }

  public static void main(String args[]) {
    Client client;

    // Check input arguments.
    if (args.length < 4) {
      System.out.println("Usage: java Client " +
        "<GeneFactory-ior> <gene-id> <Embl-ior> <embl-id>");
      return;
    }

    // Get ior(s) and id(s) from args[].
    String geneFactory_ior = args[0];
    String gene_id = args[1];
    String embl_ior = args[2];
    String embl_id = args[3];

    try {
      // Initialize the ORB.
      ORB orb = ORB.init();

      // Create a Client object.
      client = new Client();

      // Obtain IORs of factory objects.
      geneFactory = BacterialGenome.GeneFactoryHelper.narrow(
        client.retrieveIORFromFile(orb, geneFactory_ior));
      emblFactory = nsdb.EmblHelper.narrow(
        client.retrieveIORFromURL(orb, embl_ior));

      // Get gene data.
      nsdb.EmblSeq emblSeq=null;
      emblSeq = emblFactory.getEmblSeq(embl_id);
      BacterialGenome.Gene gene=null;
      gene = geneFactory.getById(gene_id, embl_id);
      System.out.print("\nGene: " + gene_id
        + "\nid: " + (gene.id()==null ? "" : gene.id())
        + "\nname: " + (gene.name()==null ? "" : gene.name())
        + "\nec number: " +
          (gene.ecNumber()==null ? "" : gene.ecNumber())
        + "\nfunction: " + (gene.function()==null ? "" : gene.function())
        + "\ndescription: " +
          (gene.description()==null ? "" : gene.description())
        + "\npos: " + gene.pos()
      );

      // Get gene references.
      String[] refs = gene.references();
      System.out.print("\nreferences (no. = " + refs.length + "):");
      for (int i=0; i<refs.length; i++) {
        System.out.print("\n   " + (i+1) + ". " + refs[i]);
      }
```

```
// Get gene promoter(s).
BacterialGenome.Promoter[] promoters=null;
promoters = gene.getPromoter();
for (int i=0; i<promoters.length; i++) {
  System.out.print("\nPromoter " + (i+1) + " location: ("
    +promoters[i].firstPos()+", "+promoters[i].lastPos()+")    "
    +((promoters[i].directionIsL2R()) ? "->" : "<-")
    +"\n              sequence: " + promoters[i].nucSeq()
  );
}

// Get gene minus10Signal(s).
BacterialGenome.Minus10Signal[] m10Signals=null;
m10Signals = gene.getMinus10Signal();
for (int i=0; i<m10Signals.length; i++) {
  System.out.print("\nMinus10Signal " + (i+1) + " location: ("
    +m10Signals[i].firstPos()+", "+m10Signals[i].lastPos()+")    "
    +((m10Signals[i].directionIsL2R()) ? "->" : "<-")
    +"\n              sequence: " + m10Signals[i].nucSeq()
  );
}

// Get gene minus35Signal(s).
BacterialGenome.Minus35Signal[] m35Signals=null;
m35Signals = gene.getMinus35Signal();
for (int i=0; i<m35Signals.length; i++) {
  System.out.print("\nMinus35Signal " + (i+1) + " location: ("
    +m35Signals[i].firstPos()+", "+m35Signals[i].lastPos()+")    "
    +((m35Signals[i].directionIsL2R()) ? "->" : "<-")
    +"\n              sequence: " + m35Signals[i].nucSeq()
  );
}

// Get gene terminator(s).
BacterialGenome.Terminator[] terminators=null;
terminators = gene.getTerminator();
for (int i=0; i<terminators.length; i++) {
  System.out.print("\nTerminator " + (i+1) + " location: ("
    +terminators[i].firstPos()+", "+terminators[i].lastPos()+")    "
    +((terminators[i].directionIsL2R()) ? "->" : "<-")
    +"\n              sequence: " + terminators[i].nucSeq()
  );
}

/* For protein coding genes, get associated sequence,
   codon usage class, and functional class. */
BacterialGenome.ProteinCoding[] pCoding=null;
pCoding = gene.getProteinCoding();
BacterialGenome.FunctionalClass func_class=null;
func_class = gene.functionalClass();
for (int i=0; i<pCoding.length; i++) {
  System.out.print("\nProteinCoding " + (i+1) + " location: ("
    + pCoding[i].firstPos() + ", " + pCoding[i].lastPos() + ")    "
    + ((pCoding[i].directionIsL2R()) ? "->" : "<-")
    + "\n              sequence: " + pCoding[i].nucSeq()
    + "\n              codon usage class: " +
      gene.codonUsageClass()
    + "\n              functional class: "
      + (func_class.id()==null ? "" : func_class.id()) + ": "
      + (func_class.description()==null ? "" :
        func_class.description())
  );
}
```

```
      System.out.println();
    }
    catch(SystemException sysEx) {
      System.err.println("***CORBA Exception:");
      System.err.println(sysEx.toString());
      return;
    }
    catch(type.NoResult ex) {
      System.err.println("***type.NoResult:");
      System.err.println(ex.toString());
      System.exit(1);
    }
    catch(type.InvalidArgumentValue ex) {
      System.err.println("***type.InvalidArgumentValue:");
      System.err.println(ex.toString());
      System.exit(1);
    }
    catch(nsdb.EmblPackage.Superceded ex) {
      System.err.println(ex.toString());
      System.exit(1);
    }
    // When toPersistenceStringNoException is used.
    catch(PS.ServerError ex) {
      System.err.println(ex.name);
      System.err.println(ex.description);
      System.exit(1);
    }
  }
}
```

References

[Abiteboul 91]

S. Abiteboul, A. Bonner (1991) Objects and Views, In J. Clifford and R. King editors, ACM SIGMOD International Conference on Management of Data, pages 238-247.

[ACEDB]

http://www.acedb.org/

[Achard 98]

F. Achard, C. Cussat-Blanc, E. Viara, and E. Barillot (1998) The new Virgil database: a service of rich links, Bioinformatics 14: 342-348.

[Arocena 98]

G.O. Arocena, A.O. Mendelzon (1998) WebOQL: Restructuring Documents, Databases and Webs, IEEE International Conference On Data Engineering (ICDE), pages 24-33.

[Atzeni 97]

P. Atzeni, G. Mecca, P. Merialdo (1997) To Weave the Web, VLDB, pages 206-215.

[Bairoch 00]

A. Bairoch and R. Apweiler (2000) The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000, Nucleic Acids Research 28(1):45-48.

[Barillot 99a]

E. Barillot, U. Leser, P. Lijnzaad, C. Cussat-Blanc, K. Jungfer, F. Guyon, G. Vaysseix, C. Helgesen, and P. Rodriguez-Tomé (1999) A proposal for a standard CORBA interface for genome maps, Bioinformatics 15(2):157-169.

[Barillot 99b]

E. Barillot, S. Pook, et al. (1999) The HuGeMap Database: Interconnection and Visualisation of Human Genome Maps, Nucleic Acids Research 27(1), pages 119-122.

[Benson 02]

DA. Benson, I. Karsch-Mizrachi, DJ. Lipman, J. Ostell, BA. Rapp, DL. Wheeler (2002) GenBank, Nucleic Acids Research 30(1):17-20.

[Booch 93]

G. Booch (1993) Object-oriented Analysis and Design with Applications, Benjamin/Cummings, ISBN 0 805 35340 2.

[Carey 95]

M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers (1995) Towards Heterogeneous Multimedia Information Systems: The Garlic Approach, International Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE-DOM), pages 124-131.

[Chawathe 94]

S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom (1994) The TSIMMIS Project: Integration of Heterogeneous Information Sources, Information Processing Society of Japan (IPSJ), pages 7-18.

[Chen 97]

IM.A. Chen, A.S. Kosky, V.M. Markowitz, E. Szeto (1997) Constructing and Maintaining Database Views in the Framework of the Object-Protocol Model, Conference on Scientific and Statistical Database Management.

[Chen 95]

IM.A. Chen, V.M. Markowitz (1995) An Overview of the Object Protocol Model (OPM) and the OPM Data Management Tools, Information Systems 20(5):393-418.

[Coupaye 99]

T. Coupaye (1999) Wrapping SRS with CORBA: from textual data to distributed objects, Bioinformatics 15(4):333-338.

[Davidson 97]

S.B. Davidson, A.S. Kosky (1997) WOL: A Language for Database Transformations and Constraints, International Conference on Data Engineering (ICDE), pages 55-65.

[Davidson 96]

S.B. Davidson, C. Overton, V. Tannen, L. Wong (1996) BioKleisli: A Digital Library for Biomedical Researchers, Journal of Digital Libraries, pages 36-53.

[Dogac 96]

A. Dogac, C. Dengi, E. Kilic, G. Ozhan, F. Ozcan, S. Nural, C. Evrendilek, U. Halici, B. Arpinar, P. Koksal, S. Mancuhan (1996) A Multidatabase System Implementation on CORBA, RIDE-NDS, pages 2-11.

[Durbin 94]

R. Durbin, J. Thierry-Mieg (1994) The ACEDB Genome Database, In S. Suhai editor, Computational Methods in Genome Research, Plenum Press, pages 45-55.

[Etzold 97]

Etzold T, Verde G (1997) Using Views for Retrieving Data from Extremely Heterogeneous Databanks, Proceedings of the Pacific Symposium on Biocomputing, Hawai, USA, pages 134-141.

[Etzold 96]

Etzold T, Ulyanov A, Argos P (1996) SRS: Information Retrieval System for Molecular Biology Data Banks, Methods in Enzymology 266:114-128.

[Fernandez 97]

M. Fernandez, D. Florescu, J. Kang, A. Levy, D. Suciu (1997) STRUDEL: A Web Site Management System, ACM SIGMOD Conference.

[Fowler 97]

M. Fowler, K. Scott (1997) UML Distilled: Applying the Standard Object Modelling Language, Addison Wesley, ISBN 0 201 32563 2.

[Frankel 99]

D.S. Frankel (1999) The OMG Meta-Object Facility, Java Report, March 1999:56-71.

[Gamma 95]

E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995) Design Patterns: elements of reusable object-oriented software, Addison Wesley, ISBN 0 201 63361 2.

[Goh 94]

C.H. Goh, S.E. Madnick, M.D. Siegel (1994) Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment, International Conference on Information and Knowledge Management (CIKM), pages 337-346.

[Guerrini 97]

G. Guerrini, E. Bertino, B. Catania, J. Garcia-Molina (1997) A Formal Model of Views for Object-Oriented Database Systems, Theory and Practice of Object Systems 3(3):157-183.

[Hoebeke 01]

M. Hoebeke, H. Chiapello, P. Noirot, P. Bessieres (2001) SPiD: A subtilis protein interaction database, Bioinformatics 17(12):1209-1212.

[Hu 98]

J. Hu, C. Mungall, D. Nicholson, and A.L. Archibald (1998) Design and implementation of a CORBA-based genome mapping system prototype, Bioinformatics 14(2):112-120.

[IONA 97]

IONA Technologies Ltd. (1997) Orbix MT 2.3c Programmer's Guide.

[Jacobson 92]

I. Jacobson, M. Christensen, P. Jonsson, G. Overgaard (1992) Object-oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, ISBN 0 201 54435 0.

[Jungfer 99]

K. Jungfer, U. Leser and P. Rodriguez-Tomé (1999) Constructing IDL Views on Relational Databases, Conference on Advanced Information Systems Engineering (CAiSE).

[Karp 96]

P.D. Karp, S. Paley (1996) Integrated Access to Metabolic and Genomic Data, Journal of Computational Biology 3(1):191-212.

[Kemp 00]

G.J.L. Kemp, C.J. Robertson, P.M.D. Gray, N. Angelopoulos (2000) CORBA and XML: Design Choices for Database Federations, British National Conferenc on Databases (BNCOD), pages 191-208.

[Kim 95]

W. Kim, W. Kelley (1995) On View Support in Object-Oriented Database Systems, In W. Kim editor, Modern Database System, ACM Press, pages 108-129.

[Kim 91]

W. Kim, J. Seo (1991) Classifying Schematic and Data Heterogeneity in Multidatabase Systems, IEEE Computer 24(12):12-18.

[Kosky 98]

A. Kosky, I.M.A. Chen, V.M. Markowitz, E. Szeto (1998) Exploring Heterogeneous Biological Databases: Tools and Applications, International Conference on Extending Database Technology (EDBT), pages 499-513.

[Kosky 96]

A. Kosky, E. Szeto, I.M.A. Chen, V.M. Markowitz (1996) OPM Data Management Tools for CORBA Compliant Environments, Technical Report LBNL-38975.

[Kuno 96]

H.A. Kuno, E.A. Rundensteiner (1996) The *MultiView* OODB View System: Design and Implementation, Theory and Practice of Object Systems 2(3):202-225.

[Kunst 97]

F. Kunst, N. Ogasawara, I. Moszer, <146 other authors>, H. Yoshikawa, A. Danchin (1997) The complete genome sequence of the Gram-positive bacterium Bacillus subtilis, Nature 390:249-256.

[Leser 98]

U. Leser, S. Tai, S. Busse (1998) Design Issues of Database Access in a CORBA Environment, Workshop on Integration of Heterogeneous Software Systems, Magdeburg, Germany.

[LSR]

OMG Life Sciences Research (LSR), http://www.omg.org/lsr/index.html

[Metamata]

Metamata, JavaCC - The Java Parser Generator, http://www.metamata.com/javacc/

[Moszer 98]

I. Moszer (1998) The complete genome of Bacillus subtilis: from sequence annotation to data management and analysis, FEBS Letters 430:28-36.

[Moszer 95]

I. Moszer, P. Glaser, A. Danchin (1995) SubtiList: a relational database for the Bacillus subtilis genome, Microbiology 141:261-268.

[OMG]

OMG, http://www.omg.org

[OMG 02]

OMG (2002) XML Metadata Interchange (XMI) Specification, v1.2, OMG Document formal/02-01-01.

[OMG 01a]

OMG (2001) The UML 1.4 Specification, OMG Document formal/01-09-67.

[OMG 01b]

OMG (2001) XMI production of XML Schema, Final Adopted Specification, OMG Document ad/01-06-12.

[OMG 01c]

OMG (2001) CORBA Meta-Object Facility (MOF) Specification, v1.3.1, OMG Document formal/01-10-41.

[OMG 99a]

OMG (1999) CORBA Components, OMG Document orbos/99-02-05.

[OMG 99b]

OMG (1999) CORBA IIOP 2.3.1 Specification, OMG Document formal/99-10-07.

[OMG 98a]

OMG (1998) CORBA IIOP 2.2 Specification, OMG Document formal/98-07-01.

[OMG 98b]

OMG (1998) CORBA Services, OMG Document formal/98-12-09.

[OMG 97]

OMG (1997) A Discussion of the Object Management Architecture, OMG Document formal/00-06-41.

[Parsons 00]

J.D. Parsons and P. Rodriguez-Tome (2000) JESAM: CORBA software components to create and publish EST alignments and clusters, Bioinformatics 16(4):313-325.

[Persistence]

Persistence, http://www.persistence.com

[Rational]

Rational, http://www.rational.com

[Ritter 94]

O. Ritter (1994) The Integrated Genomic Database (IGD), In S. Suhai editor, Computational Methods in Genome Research, Plenum Press, pages 57-74.

[Rodriguez-Tomé 97]

P. Rodriguez-Tomé, C. Helgesen, P. Lijnzaad, K. Jungfer (1997) A CORBA server for the Radiation Hybrid Database, International Conference on Intelligent Systems for Molecular Biology (ISMB), pages 250-253.

[Rohm 99]

U. Rohm, K. Bohm (1999) Working Together in Harmony - An Implementation of the CORBA Object Query Service and its Evaluation, International Conference on Data Engineering (ICDE).

[Rumbaugh 91]

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991) Object-Oriented Modeling and Design, Prentice-Hall Inc., ISBN 0 136 30054 5.

[Scholl 91]

M. Scholl, C. Laasch, M. Tresch (1991) Updatable views in Object-Oriented Databases, International Conference on Deductive and Object-Oriented Databases, pages 189-207.

[Schuler 96]

G.D. Schuler, J.A Epstein, H. Ohkawa, J.A. Kans (1996) *Entrez*: Molecular Biology Database and Retrieval System, Methods in Enzymology 266:141-162.

[Sellentin 99]

J. Sellentin, B. Mitschang (1999) Design and Implementation of a CORBA Query Service Accessing EXPRESS-based Data, DASFAA.

[Sellentin 98]

J. Sellentin, B. Mitschang (1998) Data-Intensive Intra- & Internet Applications - Experiences Using Java and CORBA in the World Wide Web, International Conference on Data Engineering (ICDE).

[Siegel 96]

J. Siegel (1996) CORBA Fundamentals and Programming, John Wiley & Sons Inc., ISBN 0 471 12148 7.

[Spiridou 00]

A. Spiridou (2000) A View System for CORBA-Wrapped Data Sources, IEEE Advances in Digital Libraries (ADL), pages 228-237.

[Stoesser 02]

G. Stoesser, W. Baker, A. van den Broek, E. Camon, M. Garcia-Pastor, C. Kanz, T. Kulikova, R. Leinonen, Q. Lin, V. Lombard, R. Lopez, N. Redaschi, P. Stoehr, MA. Tuli, K. Tzouvara, R. Vaughan (2002) The EMBL Nucleotide Sequence Database, Nucleic Acids Research 30(1):21-26.

[Sun]

Sun, JavaCC[tm] - The Java Parser Generator, http://www.sun.com/forte/ffj/resources/javacc.html

[Tateno 02]

Y. Tateno, T. Imanishi, S. Miyazaki, K. Fukami-Kobayashi, N. Saitou, H. Sugawara, T. Gojobori (2002) DNA Data Bank of Japan (DDBJ) for genome scale research in life science, Nucleic Acids Research 30(1):27-30.

[Tomasic 96]

A. Tomasic, L. Raschid, P. Valduriez (1996) Scaling Heterogeneous Databases and the Design of Disco, International Conference on Distributed Computing Systems (ICDCS), pages 449-457.

[Vinoski 97]

S. Vinoski (1997) CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, IEEE Communications 14(2).

[W3C 01a]

W3C (2001) W3C Candidate Recommendation: XML Pointer Language (XPointer) Version 1.0.

[W3C 01b]

W3C (2001) W3C Recommendation: XML Linking Language (XLink) Version 1.0.

[W3C 01c]

W3C (2001) W3C Recommendation: XML Schema Part 0: Primer, XML Schema Part 1: Structures, XML Schema Part 2: Datatypes.

[W3C 00]

W3C (2000) W3C Recommendation: Extensible Markup Language (XML) Version 1.0.

[W3C 99]

W3C (1999) W3C Recommendation: Namespaces in XML.

[Wang 00]

L. Wang, P. Rodriguez- Tomé, N. Redaschi, P. McNeil, A. Robinson, P. Lijnzaad (2000) Accessing and Distributing EMBL Data Using CORBA, Genome Biology 1(5).

[Wells 94]

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

[Wiederhold 92]

G. Wiederhold (1992) Mediators in the Architecture of Future Information Systems, IEEE Computer 25(3):38-49.