CORBA and XML: Design Choices for Database Federations

Graham J.L. Kemp[1], Chris J. Robertson[1,2], Peter M.D. Gray[1]
and Nicos Angelopoulos[1,2]

[1] Department of Computing Science, University of Aberdeen,
King's College, Aberdeen, Scotland, AB9 2UE
[2] Department of Molecular and Cell Biology, University of Aberdeen,
Polwarth Building, Foresterhill, Aberdeen, Scotland, AB25 2ZD

Abstract. The newly established standards of CORBA and XML make it much easier to interoperate between different database software running on different platforms. We are using these in a mediator-based architecture that supports integrated access to biological databases. We discuss, in turn, design issues that arise from using each of the standards. In CORBA an important design issue is the use of *coarse grain* access, which supports a query language over an extensible integrated data model, as compared with *fine grain* access, which is tailored for specific queries. We discuss experience in using CORBA in these two ways. On the other hand we describe scenarios where returning results are communicated in XML format. We present a classification based on design choices.

1 Introduction

The Common Object Request Broker Architecture (CORBA) and the Extensible Markup Language (XML) are currently promoted as providing standardisation in distributed computing applications. However, these technologies do not in themselves provide application designers with a single solution to their design problems. Rather, they serve as infrastructure standards that can be employed in many alternative ways in building a system.

In this paper we describe a federated database architecture that we are building to support integrated access to biological databases. We concentrate on the design of the interoperability layer and present a number of approaches based on the technologies of CORBA for implementing a client-server database architecture, and XML for formatting the results of database queries.

A major problem in integrating multiple internet data resources is that they do not all make their data intelligible to remote programs. Some resources depend entirely on communicating directly with a person observing a screen. This prevents one from using intelligent *mediator* programs (as proposed by Weiderhold [13]) to intercept the communication and automate the data fusion and integration. Better resources present data in well defined text formats that can be used by programs as well as humans. Still better data resources will provide

access through database management software, thus enabling remote programs to generate their own queries and process the answers. This requires the structured framework of a schema for integration with data from other resources. It is this latter kind of access that we are seeking to provide through our work with the P/FDM object database system.[1]

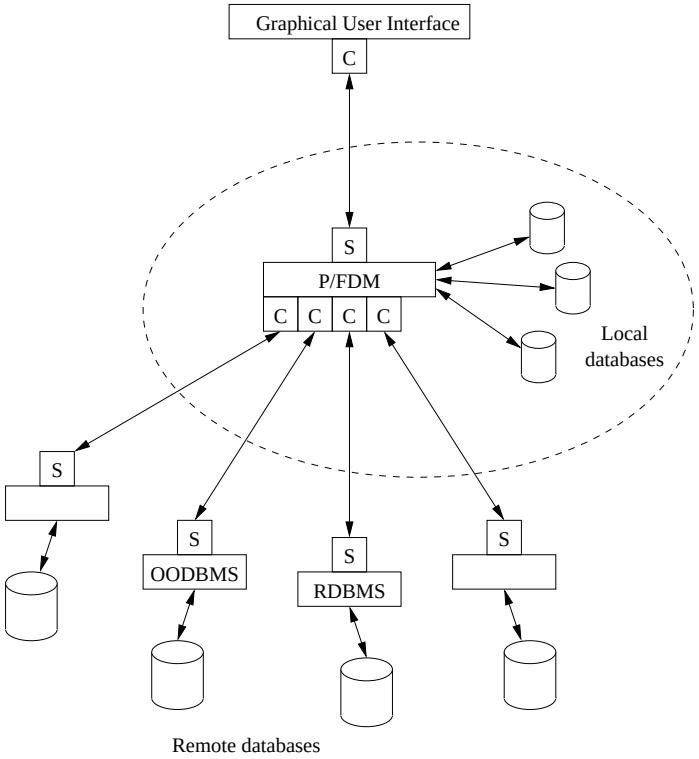

Fig. 1. CORBA and database access. Clients [C] send queries to CORBA servers [S] and sets of results are returned.

Our general architecture is shown in Figure 1 which shows how each remote data source can act as a server surrounded by a CORBA wrapper, while the integrated output from the mediator can itself be accessed through another CORBA wrapper.

In this paper we discuss some limitations of the access styles provided by many current biological data resources, and contrast this with the query capabilities of an object database. We describe alternative ways in which CORBA can be used to provide access to biological databases and show how coarse grain CORBA interfaces can provide efficient access to databases, illustrating this with

[1] http://www.csd.abdn.ac.uk/~pfdm/

an example using our antibody structure database. Finally, we discuss the role that XML can play in making the results easier to process or browse.

2 Degrees of Accessibility

2.1 Browsing and viewing

Many internet resources are intended for interactive inspection and interpretation by users observing a screen. Images, such as a metabolic pathway diagram or a graphical representation of an enzyme with its active site highlighted, or free text descriptions, such as journal abstracts or entries in biological data resources such as PROSITE (a database of protein families and domains [4]), contain information which is readily digested by users viewing these. However, the information contained within these is not easily available to programs. Image processing and natural language processing technologies still have a long way to advance before they can be used by remote programs to interpret the content of such data resources, let alone relate these observations to data from other sources and form scientific hypotheses.

Many web-based data resources only provide this kind of access. The World-Wide Web has evolved to suit human users, who tend to search for particular pages of interest, controlling navigation interactively. The search techniques used are generally those developed by the information retrieval community for free-format text. Since the program does not understand the text, it uses probabilistic matching, for example by scoring keywords, and expects to retrieve some *false matches*. Database queries, by contrast, are expected to return sets which *exactly* fit the given criteria, and may involve numerical comparisons. The data model they use effectively describes the meaning of data in the form of carefully checked formatted records, and does not have to deal with free-format natural language sentence structures. This is what allows queries to be more precise.

2.2 Formatted text files

Formatted data files, such as SWISS-PROT entries and Protein Data Bank entries, can be used more easily by programs. Many such collections are now accessible via the World-Wide Web. Using command line web interfaces like Lynx, it is possible for programs (e.g. UNIX scripts or application programs written in C or Java) to fetch a web page automatically and then process it like any other external file. However, this requires the user to have programming skills, knowledge of the data file formats, and time to write a new program whenever they wish to access different data values or different data banks or to use these for a different purpose.

The SRS[2] system [6] goes some way towards providing indexes and a query interface for flat file data banks on the same server, but the querying capabilities are not as expressive or flexible as database query languages. For example, all result fields requested must be from the same data bank.

[2] http://srs.ebi.ac.uk/

2.3 Database management systems

Database management systems (DBMS) provide *ad hoc* querying capabilities, enabling complex data retrieval requests to be expressed in a high level query language. These requests can include sophisticated data selection criteria and may traverse the data in ways not envisaged by those who created the database. The DBMS will provide optimisation capabilities that can use all of the information in the query in designing an efficient execution strategy.

Some may argue that a collection of indexed web pages constitutes a primitive database. However, the search capabilities provided are far below what one would expect from a database management system. When web pages are indexed for searching, this usually takes the form of a keyword index which enables searches for links to pages containing a specified word or phrase. Hypertext links between web pages do provide a kind of index for interactive browsing, but these links cannot be queried easily by automatic programs. If one does implement an automatic searching program which can follow links to retrieve related pages, then it is still necessary for the related pages to be retrieved one-at-a-time and for these to be processed on the client machine; each would have to be scanned sequentially to see if it matches the search criteria. It is more efficient to send selection conditions across to a remote part of a distributed database and to send back just the items required than it is to transport the data as whole web pages, only to reject much of it on arrival. This principle is at the heart of the design of the coarse grain CORBA interface described in Section 3.3.

3 CORBA interfaces to biological databases

CORBA is an architecture standard proposed by the Object Management Group[3] and widely supported by manufacturers. It provides a way to present an interface on the local machine to remote objects and their associated methods. CORBA is currently attracting widespread interest in the bioinformatics community as a technology which can assist in integrating distributed heterogeneous resources, e.g. the EMBL data model for genome data[4] is designed around the use of CORBA interfaces. We believe that CORBA can assist with the low level integration of distributed software components and with wrapping legacy systems. However, a straightforward mapping between CORBA objects, defined using the interface definition language (IDL), and database objects can lead to inefficient systems and lose the benefits of data independence which database management systems, by definition, provide. In this section we describe the use and misuse of CORBA with database systems with examples from our antibody database [9].

3.1 Fine grain and coarse grain database access with CORBA

Cormac McKenna [11] describes two alternative ways to use CORBA with an object-oriented database system (OODBS). That paper describes his experience

[3] http://www.omg.org/
[4] http://corba.ebi.ac.uk/EMBL/embl.html

using the ODB-II OODBS and the DAIS CORBA-compliant object request broker (ORB). In the first approach each object class and each OODBS instance is represented as an ORB object. An example of this kind of interface is given in Section 3.2 and Figure 3. This approach requires the programmer to specify the execution plan in terms of the limited set of access functions provided in the IDL. Each step needed to achieve a task has to be programmed explicitly. This is laborious for the programmer and the resulting code is potentially very inefficient since an application task may require many small requests to be sent from the client to the database server and many small packets of data being sent from the client to the server (fine grain access) with data selection being done on the client. McKenna comments that "the main difficulty with this approach is developing the infrastructure required to access objects from the database using queries" and that "queries cannot be passed very naturally to IDL methods".

In the second approach a single ORB object is used as the interface to the entire database, providing a server that can execute queries expressed in a high level query language. Here, the IDL resolves down to a single object rather than multiple objects which enables large queries containing several selections to be sent to the server for bulk execution (coarse grain access), taking advantage of optimisation strategies and indexes available on the server. McKenna's paper shows how queries expressed in ODQL (ODB-II's query language) can be embedded in C applications. Section 3.3 describes a coarse grain interface to a P/FDM server.

CORBA does not in itself provide a general purpose way to query data held at distributed sites. However, it does provide an architecture within which one can build the infrastructure and services to support the distribution of data. The language and platform independence which it promotes gives it an edge over sockets as a technology for enabling interprocess communication across platforms from different vendors (e.g. UNIX and Windows-NT) by providing programmers with a higher level of abstraction that hides details of the transport layer. With Object Request Brokers becoming more freely available, we should see more client-server database applications migrating to CORBA. However, we believe that it is important that database services accessible through an Object Request Broker should at least allow coarse grain access via a high level query language.

3.2 An example of fine grain access using CORBA

Suppose we want to find out whether the values of the phi and psi angles of the residues at Kabat position "27C" in antibody light chains from high resolution antibody structures correlate with the chain class. In this section and the next, we will consider two alternative approaches to this task.

Figure 2 shows how an interface to part of the antibody database can be defined in CORBA's IDL. Figure 3 shows part of a possible client program. Note particularly the cost of using a CORBA call, such as `get_residues`, to bring across a whole array of results, each of which has to be set up as a CORBA stub for a Residue object, capable of calling across the net to get its data values! Worse still, many of these objects will turn out not to be required. The advantages of

returning a batch of results as tuples, possibly wrapped in XML tags, is clear in this case.

The approach taken in this example is rather extreme, and has been chosen to show IDL code that closely resembles a database schema. Of course, additional access methods could be provided in the interface. However, the client programmer would still be responsible for designing the execution plan and the server programmer would have to write additional implementation code to support any additional methods.

```
module Antibody {

        interface Residue {
                unsigned long           pos();
                string                  name();
                string                  kabat_position();
                double                  phi();
                double                  psi();
        };

        interface Chain {
                string                  chain_id();
                unsigned long           num_residues();
                string                  chain_class();
                sequence<Residue>       get_residues();
        };

        interface Structure {
                string                  protein_code();
                string                  protein_name();
                string                  source();
                string                  authors();
                double                  resolution();
                string                  ig_name();
                sequence<Chain>         get_chains();
        };

        interface StructureDatabase {
                sequence<Structure>     get_structures()
        };

};
```

Fig. 2. IDL code for a fine grain CORBA interface.

```
// CLIENT CODE
//
// ("Pseudo-Java")

import Antibody.*;
import org.omg.CORBA.*;


// Variable declarations

org.omg.CORBA.Object    obj;

StructureDatabase       dbRef;
Structure[]             structureRefs;
Chain[]                 chainRefs;
Residue[]               residueRefs;
int                     i, j, k;

// Connect to the database using CORBA
ORB orb = ORB.init( this, null );
// ... etc. ...
dbRef = StructureDatabaseHelper.narrow( obj );

structureRefs = dbRef.get_structures();
for (i=0; i < structureRefs.length; i++) {
    if (structureRefs[i].resolution() < 2.5) {
        chainRefs = structureRefs[i].get_chains();
        for (j=0; jchainRefs.length; j++) {
            if (chainRefs[j].chain_id() == "L") {
                residueRefs = chainRefs[j].get_residues();
                for (k=0; k < residueRefs.length; k++) {
                    if (residueRefs[k].kabat_position() == "27C") {
                        System.out.println(
                                    structureRefs[i].protein_code()
                                +"  "+chainRefs[j].chain_class()
                                +"  "+residueRefs[k].name()
                                +"  "+residueRefs[k].phi()
                                +"  "+residueRefs[k].psi());
                    }
                }
            }
        }
    }
}
```

Fig. 3. "Pseudo-Java" code to compare the phi/psi angles of the residues at Kabat position "27C" in light chains of high resolution antibody structures.

3.3 An example of coarse grain access using CORBA

We have implemented a coarse grain CORBA interface to P/FDM in Java using the ORBacus[5] object request broker. Part of the IDL code for this interface is shown in Figure 4.

```
module PFDM  {

        // Exceptions here ...

        // Type definitions

        typedef sequence<string> clauseArray;   // Holds schema clauses

        interface pfdm {

                void            openModule( in string module_name )
                                raises (ModuleUnknownException,
                                        OpenModuleFailedException,
                                        UnableToOpenDBException);

                clauseArray     getSchemaClauses()
                                raises (ModuleUnknownException);

                void            sendQuery( in string daplex_query )
                                raises (IncorrectSyntaxException,
                                        UnableToOpenDBException);

                string          getTuple()
                                raises (NoResultsException);

                boolean         hasMoreTuples();

        };

};
```

Fig. 4. IDL code for a coarse grain CORBA interface.

The IDL shows that a single object is used to represent the entire database. This object provides the methods openModule, getSchemaClauses, sendQuery, getTuple and hasMoreTuples.

openModule Data in P/FDM are partitioned into modules, with each module containing data on a related set of entity classes. Significantly for integrating

[5] http://www.ooc.com/ob/

heterogeneous databases, different modules can have different physical representations. Our antibody database consists of two modules; one containing structural data and the other containing sequence data from the Kabat data bank. Once connected to a P/FDM database, the user can choose which database modules should be opened. Several modules may be open at the same time.

getSchemaClauses This method returns the database schema as an array of Daplex data definition clauses. These can be used by client applications to generate a graphical representation of the schema at run-time. This method will not be needed if the client application is a customised user interface which has knowledge of the database module's schema encoded within it, or if the user interface is simply an entry box for users to type their queries and the users are already familiar with the schema.

sendQuery This method takes a Daplex query string as its argument. The query is passed to the database and is executed.

getTuple This method returns a string containing one row of results for the query. The client process can retrieve all of the results by calling this method repeatedly, giving flow control. An alternative approach would be implement method returning *all* results for the query as a sequence of strings.

hasMoreTuples This is a predicate which returns true if there are further result tuples to be retrieved.

The task described in the Section 3.2 can be achieved much more simply using the following high level Daplex query (Daplex is the query language for P/FDM and plays the role of SQL for an object database):

```
for each r in residue such that kabat_position(r) = "27C"
  for each c in residue_chain(r) such that chain_id(c) = "L"
    for each s in chain_structure(c) such that resolution(s) < 2.5
print(protein_code(s), chain_class(c), name(r), phi(r), psi(r));
```

The query can either be typed directly or, more likely, composed incrementally using a graphical user interface. Several interfaces are possible as shown in Figure 5. In the simplest case, a canned query can be parametrised, and the parameters entered into an HTML form. This uses the well known CGI-bin mechanism to call out to C and hence to CORBA. A richer graphic interface can be provided by using a Java applet, provided the CORBA implementation is known and trusted by the browser that is hosting the applet. The most general solution we have built is a Visual Navigator [7] which is a Java application that displays the integration schema as a diagram, so that the user can then interact with it in order to build up successively more complex queries. This is done by displaying query results as mouse-active tables; simply highlighting data values of interest results in their incorporation in refined selection conditions. The query can then be extended by the user to search for further details on the data of interest, which can then in turn be refined.

When complete, the query string can be passed to the P/FDM server using the sendQuery method. The database management system has an optimiser

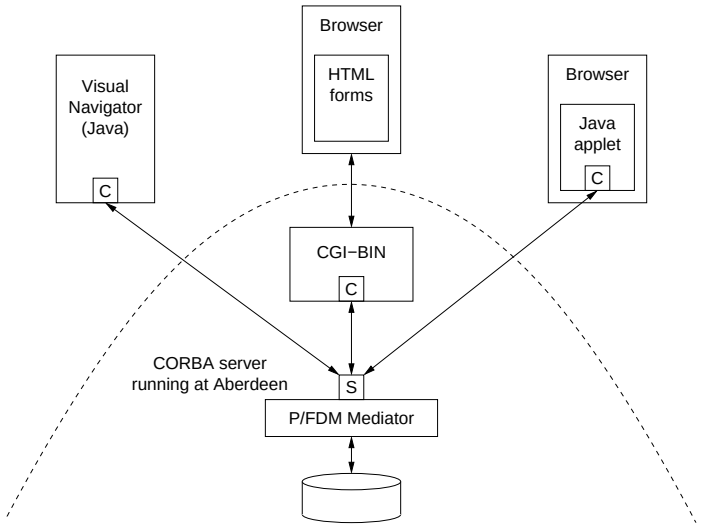

Fig. 5. Three different interfaces being used with a single shared CORBA server. They range from simple web forms to a complex Java application.

which can analyse the query as a whole and design an efficient execution plan using knowledge of the database's internal structure and indexes. Thus, selection is done efficiently on the server; not on the client, as in Section 3.2. In Section 3.2, the client does most of the work, many small access requests are sent from the client to the server, and lots of data are passed back from the server. While more access methods could be added to the interface in Section 3.2 so that larger chunks of the problem could be passed to the server, providing specific methods to cover all possible access requests would require an unending programming effort, and efficiency would still rely on the client programmer making best use of the available methods.

4 Role of CORBA in providing integrated access to biological databases

Figure 1 shows an architecture in which graphical user interface connects to the P/FDM system via the coarse grain CORBA server interface described in Section 3.3. The P/FDM system will coordinate query processing, generating code to access both the local and remote databases. CORBA client code can be used to enable P/FDM to connect to the remote database server. To allow best use to be made of these remote resources, these should also provide coarse grain interfaces enabling large queries to be passed for processing.

Why have P/FDM at the centre of Figure 1? Others may prefer to have a different software component performing the task of splitting queries, designing execution plans and sending sub-queries to remote databases. To support

such architectures, the coarse grain query interface to P/FDM described in this article is available for others to use in their programs.[6] However, there are several reasons why P/FDM is well suited for this role. First, it is based on the Functional Data Model (FDM) [12] which was itself designed for use in the MULTIBASE project [10], an earlier project in integrating distributed heterogeneous database systems. A strong semantic data model like the FDM provides data independence [8]. Thus the Daplex query makes no assumptions about the storage model that is used in each remote database (O-O, relational, hash table, flat file etc.) and we have experimented with several alternative physical storage formats [9] P/FDM has an expressive query language, called Daplex, which we have found to be better able to combine calculation with data retrieval than the query languages provided by other remote database management systems for which we have tried generating queries automatically. The P/FDM system is itself implemented largely in Prolog (with some calls to C routines) and we have chosen this because it is very efficient for transforming queries into an internal term structure representation, matching and manipulating patterns within user queries and then translating queries into the languages (such as SQL and OQL) used by other remote databases.

As usual, the mediator is capable of generating queries to send to separate databases and joining the results. This fits with the philosophy of sending the selection conditions to the data rather than sending all the data to the selection process, as noted earlier. However, some data are cached to avoid repeating the same query.

5 XML for Query Results

The data returned through a coarse grain interface is in the form of tuples containing structured values taken from remote objects. Recently we have experimented with formatting these structured values in XML. XML offers a simple and flexible mark-up language syntax for documents containing data values along with their description. This is of interest in the design of a database federation based on coarse grain database interfaces. Thus, instead of returning results wrapped up in CORBA objects, as in the fine grain case, we return them as tagged tuples in streams. These streams are essentially lines of text that have to be parsed by client processes. The presence of XML mark-up tags within the result strings can make the task of parsing the results much easier. For example, we can use downloadable XML parsers to spot patterns in the results and to display the results in a mouse-active form.

5.1 Tags based on the result format

The simplest style of mark-up for query results is one in which tags are used to reflect the tabular structure of the results (Figure 6). A P/FDM query typically

[6] Contact the authors for further details.

for each s in structure
print(protein_code(s), resolution(s))

```
<RESULTS>                    <RESULTS>
1BBD 2.8                     <ROW> 1BBD 2.8 </ROW>
1BBJ 3.1                     <ROW> 1BBJ 3.1 </ROW>
1CBV 2.6                     <ROW> 1CBV 2.6 </ROW>
</RESULTS>                   </RESULTS>
    (a)                              (b)

<RESULTS>
<ROW><FIELD> 1BBD </FIELD><FIELD> 2.8 </FIELD></ROW>
<ROW><FIELD> 1BBJ </FIELD><FIELD> 3.1 </FIELD></ROW>
<ROW><FIELD> 1CBV </FIELD><FIELD> 2.6 </FIELD></ROW>
</RESULTS>
                        (c)

<RESULTS query="for each s in structure
                print(protein_code(s),resolution(s))">
<ROW><FIELD type="string"> 1BBD </FIELD>
     <FIELD type="float"> 2.8 </FIELD></ROW>
<ROW><FIELD type="string"> 1BBJ </FIELD>
     <FIELD type="float"> 3.1 </FIELD></ROW>
<ROW><FIELD type="string"> 1CBV </FIELD>
     <FIELD type="float"> 2.6 </FIELD></ROW>
</RESULTS>
                        (d)
```

Fig. 6. XML for query results, based on the result format. The results of the query shown at the top have been formatted with tags that delimit the set of results (a,b,c,d), result rows (b,c,d) and result values (c,d). Information about the original query and data types can also be returned (d).

produces many rows of results, where each row comprises a list of values for the expressions given as arguments to the print statement. When parsing the results of a query we need to be able to identify where the results start and end, so the results can be delimited by tags such as <RESULTS> and </RESULTS>. Next, we want to identify the start and end of each row, and then the individual fields within a result row. Which of the formats shown in Figure 6 is most appropriate depends on whether the parser knows about the expected structure of the row or whether it needs extra tags in order to distinguish between strings, integers and floating point values.

```
<RESULTS>
<ROW><FUNCTION name="protein_code"  type="string"> 1BBD </FUNCTION>
     <FUNCTION name="resolution" type="float"> 2.8 </FUNCTION></ROW>
<ROW><FUNCTION name="protein_code"  type="string"> 1BBJ </FUNCTION>
     <FUNCTION name="resolution" type="float"> 3.1 </FUNCTION></ROW>
<ROW><FUNCTION name="protein_code"  type="string"> 1CBV </FUNCTION>
     <FUNCTION name="resolution" type="float"> 2.6 </FUNCTION></ROW>
</RESULTS>
                        (a)

<RESULTS>
<ROW><FUNCTION name="protein_code"  argtype="structure"
          arg="structure(1)" resulttype="string"> 1BBD </FUNCTION>
     <FUNCTION name="resolution" argtype="structure"
          arg="structure(1)" resulttype="float"> 2.8 </FUNCTION></ROW>
<ROW><FUNCTION name="protein_code"  argtype="structure"
          arg="structure(2)" resulttype="string"> 1BBJ </FUNCTION>
     <FUNCTION name="resolution" argtype="structure"
          arg="structure(2)" resulttype="float"> 3.1 </FUNCTION></ROW>
<ROW><FUNCTION name="protein_code"  argtype="structure"
          arg="structure(3)" resulttype="string"> 1CBV </FUNCTION>
     <FUNCTION name="resolution" argtype="structure"
          arg="structure(3)" resulttype="float"> 2.6 </FUNCTION></ROW>
</RESULTS>
                        (b)
```

Fig. 7. XML for query results, based on the data model. The results of the query in Figure 6 have been formatted with FUNCTION tags separating function values.

5.2 Tags based on the data model

As an alternative to focusing on the result structure, we can instead concentrate on the concepts that are central to data model (Figure 7). In the functional data model these are *entities* and *functions*.

We can extend this to include information about function arguments, their types and the function's result type.

5.3 Tags based on the schema

Another approach is to use tags that are based on the entity class names and function names that are declared in the schema (Figure 8). Taking this to an extreme, the XML tags could be organised to match the structure of the full query graph.

A drawback with this approach is that the tags used will vary from query to query and from database to database. This means that application programs will have to be able to parse a wide variety of named tags, and process the data values within these appropriately. It also means that the XML Document Type

```
<RESULTS>
<ROW><STRUCTURE>
          <PROTEIN_CODE> 1BBD </PROTEIN_CODE>
          <RESOLUTION> 2.8 </RESOLUTION></STRUCTURE></ROW>
<ROW><STRUCTURE>
          <PROTEIN_CODE> 1BBJ </PROTEIN_CODE>
          <RESOLUTION> 3.1 </RESOLUTION></STRUCTURE></ROW>
<ROW><STRUCTURE>
          <PROTEIN_CODE> 1CBV </PROTEIN_CODE>
          <RESOLUTION> 2.6 </RESOLUTION></STRUCTURE></ROW>
</RESULTS>
                         (a)

<RESULTS>
<ROW><STRUCTURE id="structure(1)">
          <PROTEIN_CODE> 1BBD </PROTEIN_CODE>
          <RESOLUTION> 2.8</RESOLUTION>
     </STRUCTURE></ROW>
<ROW><STRUCTURE id="structure(2)">
          <PROTEIN_CODE> 1BBJ </PROTEIN_CODE>
          <RESOLUTION> 3.1 </RESOLUTION>
     </STRUCTURE></ROW>
<ROW><STRUCTURE id="structure(3)">
          <PROTEIN_CODE> 1CBV </PROTEIN_CODE>
          <RESOLUTION> 2.6 </RESOLUTION>
     </STRUCTURE> </ROW>
</RESULTS>
                         (b)

<RESULTS>
<ROW><STRUCTURE id="structure(1)">
          <PROTEIN_CODE> 1BBD </PROTEIN_CODE></STRUCTURE>
     <STRUCTURE id="structure(1)">
          <RESOLUTION> 2.8 </RESOLUTION></STRUCTURE></ROW>
<ROW><STRUCTURE id="structure(2)">
          <PROTEIN_CODE> 1BBJ </PROTEIN_CODE></STRUCTURE>
     <STRUCTURE id="structure(2)">
          <RESOLUTION> 3.1 </RESOLUTION></STRUCTURE></ROW>
<ROW><STRUCTURE id="structure(3)">
          <PROTEIN_CODE> 1CBV </PROTEIN_CODE></STRUCTURE>
     <STRUCTURE id="structure(3)">
          <RESOLUTION> 2.6 </RESOLUTION></STRUCTURE></ROW>
</RESULTS>
                         (c)
```

Fig. 8. XML for query results, based on the schema. The results of the query in Figure 6 have been formatted with tags derived from entity class names and function names that are declared in the database schema.

Definition (DTD) will be much more complex. In contrast, the representations described in the previous subsections have the advantage of uniformity and can be used more easily by generic applications.

5.4 Combinations of tags

It is possible to use combinations of tag styles to describe a query's results. For example, we can mark up results with tags describing the result structure inside which are tags based on the schema. Such an approach can make the same database output useful to application programs that need to know only about the tabular structure of the results in order to display these in a simple table or spreadsheet, and also to applications that generate more sophisticated displays based on the semantic information encoded in schema-based tags.

6 Discussion and related work

The way in which we have used CORBA contrasts with the approach taken by Barillot *et al.* [2]. In their work, they are using the Oracle 7.3 relational database management system to store genome map data, and have implemented a CORBA server in Java that uses JDBC to access the Oracle system. Significantly, the CORBA interface does not provide coarse grain query access for remote users. Instead, remote users must use the specific methods defined in the IDL to access individual attributes or execute canned queries. If one wanted to achieve *ad hoc* query access then it would be necessary to implement a query processor on the client that breaks queries into individual access requests that call out to methods provided by the CORBA interface, i.e. fine grain access. Since these requests would have to be made from a language such as Java or C++ which has a CORBA binding it would also be necessary to first compile the generated calls before these could be executed. To get efficient, flexible access to the genome map database, remote users would need access either to a coarse grain CORBA interface, or direct access to the JDBC driver.

The use of XML described in this paper is different to that in work by Abiteboul (e.g. [1]) where the aim is to query data collections held as semi-structured data in an XML format. We are only using XML as a format for delivering results from a database server. We are not currently querying data in XML files, although we envisage generating queries (e.g. in XQL [7]) to access remote XQL data sets in the database federation as XQL query engines become more widely used.

Some biological database do have query-level interfaces that can be accessed via HTML forms. The appearance of such systems is encouraging since these provide a way for whole queries to be sent to remote databases for efficient processing. These web-based query interfaces make use of CGI programs to access databases, and these CGI programs return the query results as HTML pages. Modifying these programs to generate XML data descriptions in addition to HTML formatting tags is a natural extension that we can expect to see as the

use of XML becomes more widespread and browsers and client applications capable of exploiting XML tags are developed.

6.1 Object Identifiers in CORBA and XML

A crucial difference between the use of CORBA and XML to return objects as results turns on the need for further processing of these objects. If the objects are to be processed just for their data content, like records, then XML is much more efficient, since it transfers results in batches and does not have to create object stubs. If instead the intention is to apply methods defined on the remote object class, then CORBA is better because the object stub is in the right form for method application. A compromise is possible where XML is used to return the unique object identifier (within its class) for the object instance (e.g. `structure(3)` as in Figure 8). This allows one to formulate and send a query to the remote object concerned, where a Daplex function can carry out the method call. A similar principle is used by the Visual Navigator [7] discussed earlier. However, a more natural way to do this in XML would be to represent the object identifier as a hypertext link. If the server was capable of generating and returning the objects as XML mini-pages in response to such requests then it could be a good functional alternative to CORBA.

7 Conclusions

CORBA's interface definition language (IDL) is seen by some as an adequate way to represent data to be shared in a distributed environment. So why should anyone need a data model? One needs to be aware of what objects are in the CORBA world. As seen in Figure 2, to the onlooker IDL can look remarkably like a schema definition language used with an ODMG-compliant object database [5]. However, more detailed examination shows that it plays a very different role. IDL is used to declare types which can be used in programs written in different languages and at different sites, and data values conforming to the IDL declarations can be passed between these programs. Rather than describing long term persistent data, it is better to think of IDL as a way of declaring structs of the kind seen in C, or equivalent type definitions in an OOPL. In doing this, CORBA IDL provides for language and platform independence but, significantly, it does not provide for data independence in the way that a data model does. Thus, while CORBA IDL provides a good interface for programs, it provides nothing special for databases and must not be seen as a substitute for a proper data model.

Programs can't click. Web interfaces are designed to respond to mouse-click events, which trigger appropriate parametrised routines depending on screen location. However, a remote program cannot usefully emulate mouse-clicks. Instead, it is much more convenient to emulate a human typing to a command line interface, since it only has to generate a text string according to a given syntax. Humans may be bored by command lines but computers love them! They are

concise, easy to copy and send on to another machine, and their syntax can be carefully checked. CORBA interfaces are easily adapted to use them.

The relationship between CORBA and distributed databases is described by Brodie and Stonebraker [3]. They advocate that these should be viewed as complementary technologies, and that there are advantages in using software architectures which combine these. In Section 8.1.5 of their book they suggest variants of a combined architecture, ranging from a "minimal database architecture" in which distributed DBMSs support just the data management functions declared in the IDL, to a "maximal database architecture" in which an entire distributed database solution is constructed and then a bridge is built to make this accessible from a CORBA environment. We believe that a "maximal database architecture" with a semantic data model at its heart is the best way forward when data integration, rather than distributed computation, is the main goal.

8 Acknowledgements

We would like to thank Andreas Larsson and Lars Sundberg for assistance with testing the CORBA interface. This work is supported by a grant from the BBSRC/EPSRC Joint Programme in Bioinformatics (Grant Ref. 1/BIF06716).

References

1. S. Abiteboul. On Views and XML. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 1–9. ACM Press, 1999.
2. E. Barillot, U. Leser, P. Lijnzaad, C. Cussat-Blanc, K. Jungfer, F. Guyon, G. Vaysseix, C. Helgesen, and P. Rodriguez-Tomé. A Proposal for a standard CORBA interface for genome maps. *Bioinformatics*, 15(2):157–169, 1999.
3. M.L. Brodie and M Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, 1995.
4. P. Bucher and A. Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In R. Altman, D. Brutlag, P. Karp, R. Lathrop, and D. Searls, editors, *ISMB-94: Proceedings 2nd International Conference on Intelligent Systems for Molecular Biology*, pages 53–61, 1994.
5. R.G.G. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.
6. T. Etzold and P. Argos. SRS an indexing and retrieval tool for flat file data libraries. *CABIOS*, 9:49–57, 1993.
7. I. Gil, P.M.D. Gray, and G.J.L Kemp. A Visual Interface and Navigator for the P/FDM Object Database. In N.W. Paton and T Griffiths, editors, *Proceedings of User Interfaces to Data Intensive Systems (UIDIS'99)*, pages 54–63. IEEE Computer Society Press, 1999.
8. P.M.D. Gray and G.J.L. Kemp. Object-Oriented Systems and Data Independence. In D. Patel, Y. Sun, and S. Patel, editors, *Proc. 1994 International Conference on Object Oriented Information Systems*, pages 3–24. Springer-Verlag, 1994.

9. G.J.L. Kemp, Z. Jiao, P.M.D. Gray, and J.E. Fothergill. Combining Computation with Database Access in Biomolecular Computing. In W. Litwin and T. Risch, editors, *Applications of Databases: Proceedings of the First International Conference*, pages 317–335. Springer-Verlag, 1994.
10. T. Landers and R. L. Rosenberg. An Overview of MULTIBASE. In H.-J. Schneider, editor, *Distributed Data Bases*. North-Holland Publishing Company, 1982.
11. C. McKenna. Integrating the Object Database System ODB-II with Object Request Brokers. *ICL Technical Journal*, 1996.
12. D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.
13. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.