IRO-DB
A Distributed System Federating Object and Relational Databases*

Georges Gardarin, Sofiane Gannouni, Béatrice Finance

Université de Versailles, Laboratoire PRiSM
45 avenue des Etats-Unis, 78000 Versailles, FRANCE
email: {georges.gardarin,sofiane.gannouni,beatrice.finance}@prism.uvsq.fr

Peter Fankhauser, Wolfgang Klas

GMD-IPSI, Integrated Publication and Information Systems Institute
Dolivostr. 15, D-64293 Darmstadt, GERMANY
email: {fankhaus,klas}@darmstadt.gmd.de

Dominique Pastre, Régis Legoff

EDS
4 Avenue Pablo Picasso, 92024 Nanterre Cedex, FRANCE

Antonis Ramfos

INTRASOFT
2, Adrianiou Str., 11525 Athens, GREECE
email: {antonis}@isoft.intranet.gr

In Omran Bukhres, Ahmed K. Elmagarmid (Eds.):
Object Oriented Multidatabase Systems: A Solution for Advanced Applications.
Chapter 20. Prentice Hall, Englewood Cliffs, N.J., 1995

---

*This work is initiated and partially supported by the European IRO–DB ESPRIT project. The project is developed in cooperation with GMD, GOPAS, IBERMATICA, EDS, GRAPHAEL, INTRASOFT, FAW, and O2 Technology. PRiSM acts as a subcontractor of GRAPHAEL and EDS.

Contents

1 IRO–DB: A Distributed System Federating Object and Relational Databases-DB 1

1.1 Introduction 1

1.2 Design Choices 4

1.3 System Architecture 8

1.4 Main System Services 12

1.4.1 Local Database Adapters 12

1.4.2 Interchange Object-Oriented Model with Associated Protocols 16

1.4.3 Interoperable Object Management and Application Programming Interface 19

1.4.4 Schema Integrator Workbench 21

1.4.5 Global Query Decomposition 22

1.4.6 Repository Management 24

1.5 Conclusion 25

IRO–DB: A Distributed System Federating Object and Relational Databases*

Georges Gardarin, Sofiane Gannouni, and Béatrice Finance

Université de Versailles, Laboratoire PRiSM
45 Avenue des Etats-Unis, 78000 Versailles, FRANCE
{georges.gardarin,sofiane.gannouni,beatrice.finance}@prism.uvsq.fr

Peter Fankhausera and Wolfgang Klas

GMD-IPSI, Integrated Publication and Information Systems Institute
Dolivostr. 15, D-64293 Darmstadt, GERMANY
{fankhaus,klas}@darmstadt.gmd.de

Dominique Pastrea and Régis Legoff

EDS
4 Avenue Pablo Picasso, 92024 Nanterre Cedex, FRANCE

Antonis Ramfos

INTRASOFT
2, Adrianiou Str., 11525 Athens, GREECE
{antonis}@isoft.intranet.gr

1.1 Introduction

There is a growing need for tools that ease the close interworking of pre-existing relational databases and evolving object-oriented databases and applications. The commercially available distributed database management systems intend to share

*This work is initiated and partially supported by the European IRO–DB ESPRIT project. The project is developed in cooperation with GMD, GOPAS, IBERMATICA, EDS, GRAPHAEL, INTRASOFT, FAW, and O2 Technology. PRiSM acts as a subcontractor of GRAPHAEL and EDS.

data and to interoperate but are confronted with inadequate support both for integrated access to multiple databases and for integrating multiple applications into a comprehensive framework. Products like INGRES/STAR [ML87] and ORACLE/STAR are *homogeneous* distributed database systems, supporting gateways to external DBMSs (Oracle/Star provides SQL*Connect to access DB2 databases and others, Ingres/Star provides Gateway-SQL to access, for example, Rdb and Oracle databases) with only limited capabilities. Thus, they require a complete change of the organizational structure of existing databases to cope with heterogeneity. Programming environments with dedicated gateways to various DBMSs, like UNIFACE, NATURAL, UDMS (User Data Management System) or Visual Basic, provide only structural (table-oriented) interfaces to external databases, while requiring a complete re-engineering of existing applications. They are restricted to down loading retrieval with only limited cross-database joins, and updates can be issued to single sites only. Unfortunately, actual object-oriented databases rarely offer tools to interoperate or to exchange data.

The IRO–DB system described in this paper adopts the federated approach to database interoperability [SL90] to overcome these limitations. Unlike homogeneous distributed database management systems, a federated database management system adds layers of software to pre-existing heterogeneous database management systems without privileging one system. These layers provide for syntactically uniform export schemas and data manipulation languages and also for semantically integrated schemas with global transaction management and concurrency control over multiple sites. Most importantly, this approach does not violate the autonomy of the pre-existing databases; that is, database providers participating in a federation do not lose control over their data and ideally do not have to re-engineer their DBMSs and applications to allow interoperability with other DBMSs. Several federated database systems have already been prototyped [TTC+90]. They can be characterized by the degree of integration they achieve: loosely coupled federations such as MRDSM [LA87] and tightly coupled federations such as DDTS [DL87], Multibase [Shi81, LR82], and Mermaid [TBC+87]. None of these systems fully support object-oriented features, and all of them are research prototypes for experimental applications. Pegasus [ASD+91] is a new system developed at HP Research Laboratories, which is based on the object-oriented approach. Another example is InterBase* [MBE95, ME93, BCD+93], a multidatabase system providing an object-oriented SQL-based common query language for writing distributed transactions. We briefly discuss the key features of Multibase, Mermaid, and Pegasus below.

Multibase [LR82], unlike IRO-DB, provides a global integrated schema and a single functional query language for retrieving data from autonomous relational, hierarchical, and network pre-existing databases. Multibase supports a data definition and a data manipulation language called DAPLEX, which is based on a functional data model [Shi81, LR82]. Each local host schema is translated into a local schema expressed in DAPLEX and then integrated into a global schema. The system architecture of Multibase consists of three major types of components: global data manager (GDM), local DB interfaces (LDIs), and internal DBMS. The

global data manager provides global query manipulations. A global query is parsed into queries expressed over individual local DBs. These subqueries are modified to compensate for operations that cannot be performed at local DBs and sent to correspondent LDIs. These subqueries are translated from DAPLEX into the local DB query language by the LDIs. When these subqueries are processed, results generated by local DBMSs are translated into DAPLEX and returned to the GDM by the LDIs. Data retrieved by the LDIs are performed by the internal DBMS, under direction of the GDM, for any processing that cannot be performed at local DBMS.

Mermaid [TBC+87, TBC+86, TLW87b] provides integrated access to heterogeneous and autonomous DBs. It integrates relational DBMSs (Britton-Lee's IDM-500, Ingres, Rhodnius Mistress, and Oracle) and can also retrieve data from files. Unlike IRO-DB, Mermaid is not a DBMS, but a front-end system that locates and integrates data that are maintained by local DBMSs. From each local database schema, Mermaid defines a distributed local schema. Each distributed local schema represents the subset of a local database that a local system allows to be shared. Distributed schemas are integrated to define a global schema. All these schemas are described with the relational data model. SQL, Quel, and ARIEL (a query language with semantic features) are provided as query languages. The system architecture of Mermaid consists of four components: the user interface (UI), the server, the data dictionary/directory, and the DBMS interfaces. Using the UI, users submit global queries formulated with the SQL or ARIEL query language. A global query is processed by the server and translated in a data intermediate language (DIL). A DIL request is decomposed into DIL subqueries using the DD/D, which contains information about DBs and the environment. The server includes also the optimizer that plans query processing and the controller that controls the execution. The DIL subqueries are transmitted to the DBMSs interfaces, which translate them into the query language of local DBMSs, interact with appropriate local DBMSs, receive results, and send data to the server. Like IRO-DB, each Mermaid component uses the RPC remote procedure call protocol above Transmission Control Protocol/Internet Protocol (TCP/IP) to communicate with the other components.

Pegasus [ASD+91], like IRO-DB, provides facilities for multiple-DB applications to access and to manipulate multiple autonomous, heterogeneous, and distributed object-oriented and relational systems through a uniform interface. Dealing with heterogeneous DB models, Pegasus defines a common object data model and data language based on the object model of Iris, an OODBMS developed by HP. The data manipulation language is called HOSQL. Pegasus provides the facility of attaching to a Pegasus database (native DB) other external databases (foreign data sources), providing in this way access to multiple DBs. A client is connected to a native DB, called the root DB, through which he may have access to other attached DBs. For each foreign data source, Pegasus associates an imported DB, which is considered as a native DB except that its data are maintained at a foreign system. The Pegasus architecture consists of three major functional layers. The intelligent information access (IIA) layer provides services and interfaces for communicating with Pegasus. The cooperative information management (CIM) layer is responsible for data in-

tegration, query decomposition and execution, and transaction management. The foreign data access (FDA) layer provides necessary services for translating foreign schema to Pegasus schema, for transforming HOSQL queries to requests expressed with the query language of the foreign system, and for converting foreign data to Pegasus data.

The IRO-DB (Interoperable Relational and Object DataBases) ESPRIT project provides a set of tools to achieve interoperability of pre-existing relational databases and new object-oriented databases. In contrast with current available technology, the project is based on the object-oriented paradigm, both for describing the heterogeneous databases and for accessing the federated databases. The system architecture is divided into three layers: (1) The local layer alleviates model differences and provides a standard ODMG interface for each participating system; (2) the communication layer assures the exchange of requests and objects between client and server sites; (3) the interoperable layer provides integrated views of the various participating databases and integrates them within the environment of a home system.

At the interoperable layer, the user interoperates with its home DBMS, which is supposed to conform with the ODMG standard. It should provide an ODL and OQL with C++ binding interface. At the communication layer, IRO-DB offers a C/C++ library to access, in an integrated way, heterogeneous databases supported by communication protocols to exchange OQL queries and responses composed of collections of objects. The library is an object-oriented extension to the world of evolving (de facto) standards, that is, the call level interface of the SQL access group. At the local layer, the same OQL interface must be provided, but restricted to the local functionalities. Furthermore, at the interoperable layer, tools to design and dynamically maintain integrated applications on large federations of heterogeneous databases are also designed. Together, the developed tools constitute a *federated object-oriented database system*, mediating between heterogeneous databases and integrated applications. They provide uniform ODMG interfaces on top of dynamically adaptable integrated schemas for data exchange and sharing, global transaction management and concurrency control, without violating the autonomy of the participating databases.

This paper is organized as follows. In Section 1.2, we present the design choices. In Section 1.3, we present the IRO-DB system architecture. In Section 1.4, we describe the system services. We focus successively on the local database mapping, the query and object interchange protocol, the interoperable object manager, the schema integrator workbench, and the user interface. In Section 1.5, we conclude with a discussion of the status of the project and likely future directions.

1.2 Design Choices

The heterogeneous federated database system developed in IRO-DB supports existing relational databases and new object-oriented databases. Network or hierarchical databases have not been included because of resource problems, not for technical

reasons, because the object-oriented interchange model is rich enough to support networks or hierarchies. IRO-DB provides attractive tools allowing existing data-intensive applications to cooperate with and complement object-oriented technology. In light of past experiences, including the SIRIUS project in France [LBE+82] and the POREL project in Germany [NB77], IRO-DB specifically focuses on system autonomy, avoiding static and complete global schemas and supplying flexible tools to describe data on a user or user group basis at various levels. It also provides a uniform interface at each layer of the system based on the ODMG recommendations [Cat93].

Based on these requirements, the following design choices have been made for IRO-DB:

- *Object-Oriented Data Modeling*: Local databases are federated through partial local descriptions that are exchanged with other sites. Such descriptions are called export schemas. Export schemas must be defined with a rich data model, both to make clear the semantics of exported data and to make possible the integration of several export schemas in a common application. We select an object-oriented data model to describe exchanged data. Among them, several candidates are possible. Because the ODMG data model (i.e., an extension of the OMG model for databases) appears as a de facto standard for object databases, the IRO-DB interchange data model is based on this standard, including the concepts of types, subtypes, classes, relationships, and operations [LAC+93]. Each local database is described using these concepts, including relational databases that require only the use of types and classes.

- *Procedural Local Adapters*: The mapping between a local schema and an export schema has to be specified, for example, to specify the mapping of local data types to the ODMG data types, to map object identifiers, and to unify response codes. To keep the system simple, we choose to use a procedural approach for specifying mappings. Mappings are defined through an extensible toolbox composed of C procedures. The toolbox includes services to translate OQL queries in specific local queries, services to map standard data types of relational and OO systems to the interchange data types, services to adapt object identifiers of local OO systems to a common representation. It also includes generic interfaces to integrate new data type mappings. The integration of a new system in the federation requires a specific adaptation with possible enrichment of the mapping library.

- *Extended RDA Protocols*: RDA is an ANSI standard. It comprises a generic part, which permits the sending, preparation, and execution of a database program, and a specific SQL part for relational database accesses. The SQL access group is promoting a practical adaptation of this protocol. Our goal is to specify an object-oriented counterpart for RDA (OORDA). We proceed by generalizing SQL RDA to integrate the OQL query language of ODMG, that

is, method calls in queries, support of object identity, extended data types, transfer of objects, and generalization.

- *OQL/CLI Communication Interface*: The X/OPEN Group, in collaboration with the SQL access group, is promoting a C application program interface (API) for database access, under the name CLI. This interface is composed of a set of functions to establish a connection, send an SQL query and receive the results, and to control transactions. We intend to follow this specification and extend it for fully supporting OQL, the ODMG query language. Although defined for C, this interface can be bound to multiple user languages. C++ is a de facto standard for object-oriented systems; thus, it is desirable to provide C++ interfaces for application programming.

- *Integration of Clients Through a Home OODBMS*: To support client applications efficiently, persistent objects should be stored in a home OODBMS locally on the client site. This home OODBMS is a key component for integrating a client site to IRO-DB, because it is accessed directly, without going through the communication layer. Also, it is the natural support for the interoperable layer, both for global view definition and manipulation. The project intends to supply C++ as a language to program user applications. C++ database programming interfaces are already well supported by most of the object-oriented systems. Furthermore, the ODMG is proposing standard binding for persistent object and C++. Following this specification should provide facilities to integrate all C++ application development tools already available on the selected home OODBMS.

- *Global Transaction Management*: The transaction management in IRO–DB is based on the open nested transaction model [Wei91]. Open nested transactions consist of independently managed subtransactions at different levels of abstraction. This makes the integration of the transaction managers of existing systems feasible. To support local transaction control, a basic transaction processing protocol (TP) should be implemented at the communication and local layer. This allows users to reliably begin and commit local subtransactions on any system. In order to support transactions that spawn multiple databases, an interoperation layer facility is needed. The interoperation layer provides facilities for maintaining concurrency control and recovery in presence of local execution autonomy.

- *Assisted Schema Browsing and Integration*: To retain the flexibility of loosely coupled federations without sacrificing the higher degree of usability and correctness of more tightly coupled federations, IRO-DB provides automated tools for the design and maintenance of partially integrated views. Integrated views overcome inconsistencies in naming, structure, and scaling between export schema constituents with overlapping instances. For this purpose, we adopt a rule-based methodology. (For a survey see [BLN86].) Rules form a

flexible instrument to model a design theory for schema integration and to put it into operation in the form of a design tool. They can be used to transform schema constituents into a normal form to overcome structural inconsistencies; to test such normal forms for equivalences, subsumption, overlap, and inconsistencies; and to generate, on this basis, integrated schema constituents and their mappings to the underlying schemas. Several rule-based tools for schema integration have already been prototyped, for example, [SLCN88], [HR90]. These tools support mainly the retrieval of similar schema constituents and the generation of integrated schemas based on declarative correspondence assertions given by the user. However, these tools typically aim at a single completely integrated schema, do not offer adequate support for coping with local schema evolution, and lack a proper integration with more comprehensive application programming environments. (See also [RR89].) The tools developed within IRO-DB include multiple schema browsers together with straightforward schema retrieval facilities to identify corresponding schema constituents. They also support rules to infer mappings and integrated views from correspondence assertions. Unlike previous tools, these rules also guide in overcoming structural inconsistencies. Control strategies to trigger these rules for incremental, goal-oriented design as described in [NS88], and for propagating export schema changes into the integrated views, are designed and integrated with tools for browsing schemas.

- *Distributed Repository Maintenance*: To federate the multiple databases, schema definitions and mapping informations are available at various levels. Each local system should maintain an object-oriented description of the export schemas with mapping informations. The interoperable system on client sites should maintain a copy of the export schemas, the defined integrated views, and the mapping information. To simplify the schema management, we maintain two types of repository: the server repository and the client repository. They both contain descriptions of export schemas that are duplicated (consistency is maintained through a version number), different mapping informations (i.e., local mapping definition for servers and integrated view mapping definitions for clients), and integrated view definitions for client sites. All schemas having been defined using the ODMG data model, we select a common repository organization based on the ODMG metaschema [LAC+93]. Therefore, server and client repositories are defined as an object base whose schema is the ODMG metaschema. Extensions are necessary to handle mapping information. On the client site, mapping information is maintained as OQL queries, while they can be ad hoc on servers.

- *Global Query Optimization*: It is necessary to decompose a user query referencing an integrated view into local subqueries referencing export schemas and in object transfers. This decomposition can be purely syntactic, acting on a syntax tree representation of the query. However, for optimization purpose, an algebraic approach to global query optimization is considered; that is, an

OQL query is decomposed into a tree of operations of an object algebra transformed into a canonical form. Then, for each operation, an execution site is chosen, based on simple heuristics. Finally, groups of colocated operations are sent to a local site through the extended CLI interface. Final integration of local objects computed locally in response to subqueries is performed on the home OODBMS. As much as possible, we choose a lazy evaluation strategy, transfering object identifiers when possible in place of values. In other words, an object will only be instanced when required by the application. Partial instantiation is also possible when only certain attributes are required.

In summary, IRO-DB is a federated database management system that supports existing relational databases and new object-oriented databases. A federated database system is a collection of local database systems that cooperate through a home OODBMS to answer queries and handle updating transactions, while maintaining the local autonomy of the participant systems [SL90]. The IRO-DB system complements existing data-intensive applications with capabilities provided by object-oriented technology. It provides to the user the full benefit of object-oriented systems for constructing new applications using a persistent and interoperable C++ compatible with ODMG requirements. The current prototype is being developed based on three object systems—O2, MATISSE, and ONTOS—and one relational system, INGRES. ONTOS has been selected in the first experience as the home OODBMS, but any system compliant with ODMG could be used.

1.3 System Architecture

The system follows the standards for open system interconnection. Therefore, a client-multiserver application protocol is the central component of the architecture, around which other components are organized. This protocol is a specialization of the RDA protocol for object-oriented databases. The external interface follows the SQL Access Group call level interface recommendations, but extends it to handle objects. Local database adapters are provided to interface local DBMSs. One more layer, the interoperable layer, is necessary to integrate the various databases in a reliable and consistent way and to interface them to an object-oriented programming language (C++) in a transparent way. This layer is built upon a home OODBMS running on client sites.

In addition to the external layer composed of existing tools, the architecture of the system is organized in three layers of components as represented in Figure 1.1. At the interoperable layer, object definition facilities stand for specifying integrated schemas, which are integrated views of the federated databases. Views and export schemas are stored in a client repository. Object manipulation facilities include an embedding of OQL in the C++ user language and modules to decompose global queries in local ones and to control global transactions. Object definition and manipulation facilities are built upon the home OODBMS, which should be ODMG compliant to guarantee portability of the interoperable layer. The open storage

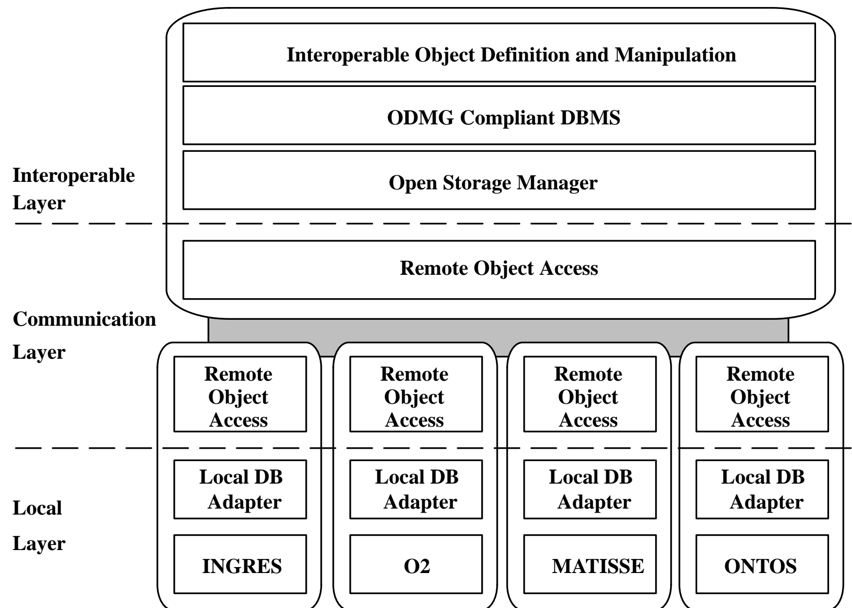

Figure 1.1. System architecture.

manager associates to each remote class defined in an export schema a virtual class defined on the home OODBMS. Objects in the virtual home class are virtual in the sense that this class is overloaded with functions to retrieve objects from a remote site and to store them in the remote site. The communication layer implements object-oriented remote data access (OO RDA) services through the remote object access (ROA) modules, both on clients and servers. Integration of the ROA protocol within the home OODBMS is provided through the open storage manager. Thus, it is possible to apply OQL/CLI functions to retrieve objects in virtual classes representing the classes of export schemas on the home site. OQL/CLI functions include connection and deconnection, preparation, and execution of OQL queries with direct transfer of results. In addition, the communication layer manages global identifiers for objects. The local layer is composed of a local database adapter. The local database adapter provides functionalities to make a local system able to answer OQL queries on an abstraction of a local schema in terms of ODMG schema (export schema). Because an export schema only describes locally implemented types, only locally available functions are available for querying through OQL. That means, for example, that methods and access paths cannot be invoked with simple relational systems. Only flat objects without relationships are handled at the local layer for relational systems. Of course, if the local system supports the full ODMG model, all syntactically correct ODMG queries are acceptable.

The schema architecture is described in Figure 1.2. A local schema is specific to

a local DBMS. An export schema describes exported data using the ODMG object-oriented data model derived from the OMG object model. An imported export schema is identical to an export schema, but on a client site. It is indeed a copy of an export schema. An interoperable schema is an integration of part of the export schemas included in a federated database, with a unique semantics for all object types and operations. It can be perceived as an integrated view of the federated database.

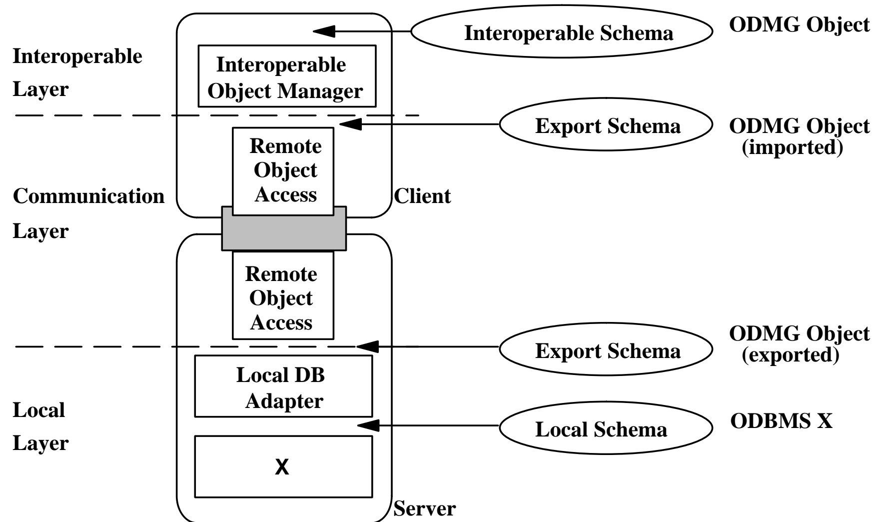

Figure 1.2. Schemas at the various levels.

More precisely, the three layers of the system assure the following functions:

1. *The local layer* provides remote clients with export schemas defined with the object-oriented interchange data model. The local DB adapter processes remotely OQL queries and updates. OQL queries conform to the export schema stored in a local repository. For example, if the export schema does not include relationships, the query should not refer to relationships, which guarantees the feasibility of the mapping to local queries. The design tools provide the required facilities to define the necessary mappings both for queries and updates, including transaction mechanisms. These mapping definitions are expressed as procedures that translate data from the local database to the export format and vice versa.

2. *The communication layer* provides object-oriented remote data access services through the CLI interface with the necessary communication translation. The run-time module provides mechanisms to open and close remote database sessions, to send and remotely prepare a database program for queries or updates,

conforming to the SQL Access Group call level interface extended to support objects (i.e., OQL). It also provides object transfer services in both directions. In summary, it provides services for objects and query exchanges expressed in OQL on a point-to-point basis. It further includes the necessary transaction management commands to open and commit local subtransactions.

3. *The interoperable layer* provides services to hide the multiple databases. It includes a tool to design interoperable schemas; the schema integrator workbench, which allows the database administrator to define integrated object-oriented schemas; and mappings with the export schemas. It also supplies the necessary mapping facilities to alleviate semantic heterogeneity, including distributed query decomposition and optimization. This layer includes global transaction management, with concurrency control and recovery algorithms. Finally, it allows applications to access the federated databases using the C++ object-oriented language of the home system through a C++ binding with OQL.

In summary, a language view of the architecture is portrayed in Figure 1.3. At each layer, the interface is based on the ODMG recommendations. At the local and communication layers, OQL queries are embedded in C following the SQL access group CLI interface. At the user level, the interface of the home OODBMS is available. As the home OODBMS should be compliant to ODMG, the binding

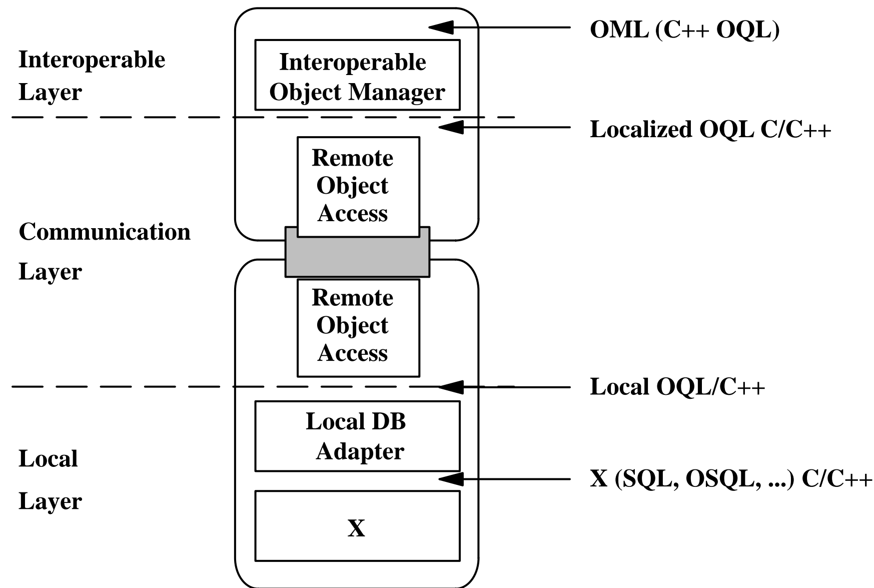

Figure 1.3. The language interfaces in IRO-DB.

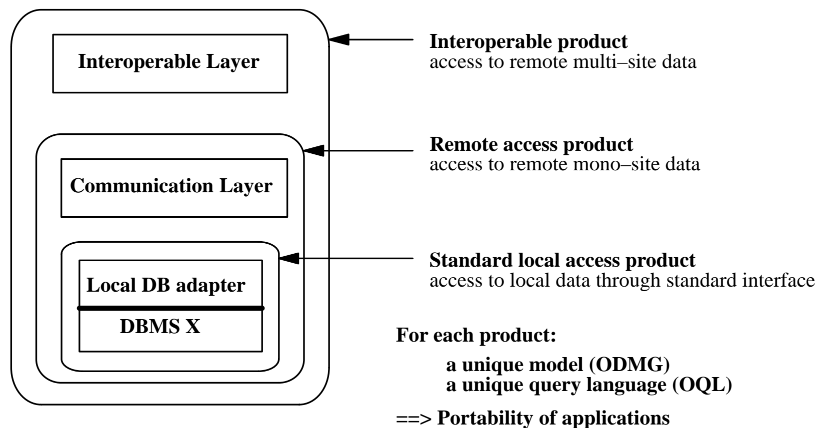

Figure 1.4. The IRO-DB products.

of OQL in C++ (called OML in ODMG) should be provided. Each layer of the IRO-DB system provides a clear interface to the users. Thus, three products can be composed by encompassing progressively the three layers (see Figure 1.4). The three of them offer an OQL-based interface, which should permit portability of applications from one layer to another. The argument for supplying a uniform query and update language with facilities to invoke methods is to avoid the application programmer having to know about different query languages. Furthermore, the resulting application is easier to maintain: Data can be moved from one system to another without rewriting the code.

1.4 Main System Services

1.4.1 Local Database Adapters

Local database adapters are located on servers. They are responsible for presenting views of local databases, called export schemas, to the clients, according to the common object-oriented interchange data model. Note that a site can be both a client and a server: Being a client or a server is a function provided by the corresponding IRO-DB modules. A local database adapter includes two parts (see Figure 1.5), a generic one and a specific one. The generic part is independent of the local system. It includes the export schema manager and the repository for both export schema and mapping information. It also includes an OQL parser, which parses the query in a syntactic tree to prepare translation to the local system.

The specific part includes the translation into local system commands. For example, procedures to translate a query in standard SQL queries would be supplied for relational systems. Conversion of local data types in ODMG data types and vice versa would also be provided. Furthermore, a global object identifier will be

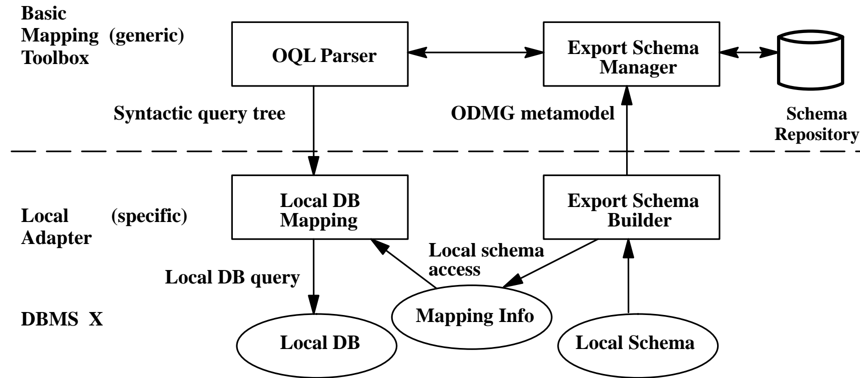

Figure 1.5. Local database adapter overview.

```
PART (prt_id: VarChar, prt_dsc: VarChar,
     prt_upd_date: Date, prt_type: Char(3)),
      Key is prt_id;
MASTER_PRODUCTION_SCHEDULE
     (prt_id: VarChar, mps_date: Date,
     quantity: Float(8), quantity_type: Char(1))
      Key is prt_id, mps_date, quantity_type,
     prt_id  ref PARTS;
FACTORY (fcr_id: Char(2), fcr_dsc: VarChar)
      Key is fcr_id;
MANUFACTURED
     (fcr_id: Char(2), prt_id: VarChar,
      manufacturing_flag: Char(1))
       Key is fcr_id, prt_id,
      fcr_id  ref FACTORY,
      prt_id  ref PART;
```

Figure 1.6. A relational database schema.

generated in a standard format (site identifier + character string), both for relational systems and object systems. Keys will be used for relational systems, and OIDs will be converted in external format for object systems. The reverse conversion procedures for global object identifiers to local identifiers will also be needed. The mapping of data structures from local schema to export schema will be as direct as possible, thus reducing the need for mapping information.

The generic part of the adapter will be implemented first on MATISSE. In a second phase of the project, we will generalize the generic procedures of the adapter and extend them to more systems, more data types, and also update procedures. The adapter will be ported onto the different database systems selected for the project, and a specific part will be developed on each DBMS (INGRES, O2, ONTOS). To illustrate further the functions of the local database adapter, we assume a local database whose relational schema is given Figure 1.6.

The ODMG counterpart of this schema is described in Figure 1.7. We assume that this schema is exported to client sites under the name SITE1. Note that

```
interface PART {
     key prt_id;
     attribute String prt;
     attribute Stringprt_dsc;
     attribute Dateprt_upd_date;
     attribute Char(3) prt_type};
interface MASTER_PRODUCTION_SCHEDULE {
     keys prt_id, mps_date, quantity_type;
     attribute String prt_id;
     attribute Datem ps_date;
     attribute Float(8) quantity;
     attribute Char(1) quantity_type  };
interface FACTORY {
     key fcr_id;
     attribute Char(2) fcr_id;
     attribute String fcr_dsc };
interface MANUFACTURED {
     keys fcr_id, prt_id;
     attribute Char(2) fcr_id;
     attribute String prt_id;
     attribute Char(1) manufacturing_flag  };
```

Figure 1.7. The corresponding SITE1 export schema.

translation is direct, even for the keys, which are included in the ODMG model. Referential integrity constraints are lost. A better translation would assume replacing them by object references, which is planned to be realized in a later project phase.

```
interface PART {
     key prt_id;
     attribute String prt_id;
     attribute String prt_dsc;
     attribute Date prt_upd_date;
     attribute Char(3) prt_type };

interface PRODUCED_PART: PART {
     attribute Float(8) planned_qty;
     attribute mps: Set < Struct <Date mps_date,
                  Float(8) quantity, Char(1) quantity_type >>;
     relationship MANUFACTURED manufactured_in
                  inverse MANUFACTURED::parts };

interface PURCHASED_PART: PART {
     attribute Float(8) min_ordered_qty;
     attribute String main_supplier };

interface FACTORY {
     key fcr_id;
     attribute Char(2) fcr_id;
     attribute String fcr_dsc;
     relationship Set<MANUFACTURED> manufactures:
                  inverse MANUFACTURED::manufactured_by };

interface MANUFACTURED {
     attribute Char(1) manufacturing_flag;
     relationship FACTORY manufactured_by
                  inverse FACTORY::manufactures;
     relationship Set<PRODUCED_PART> parts
                  inverse PRODUCED_PART manufactured_in };
```

Figure 1.8. The SITE2 export schema.

Another schema that could be exported for a similar database handled by an ODMG compatible OODBMS is given in Figure 1.8. Note that manipulating a global database composed of the relational one on site 1 and the object one on site 2 requires powerful schema integration facilities.

1.4.2 Interchange Object-Oriented Model with Associated Protocols

The interchange data model is a conceptual object-oriented data model, which is the extension of the OMG proposal by the ODMG group. It is composed of abstract data types referred to as types and defined as interfaces. A type is an aggregation of named elements of given types and a set of operations designed to manipulate these elements. (See for example, PART, MASTER_PRODUCTION_SCHEDULE, FACTORY, and MANUFACTURED in Figure 1.7). Type implementations are called classes. There exists a predefined set of atomic types, including integer, real, string, Boolean, and object identifiers. A well-designed type can be used as a predefined data type to build new types. The only visible interfaces of a type are the public operations and properties. Similarly to most OO data models, the interchange data model allows for type derivation through the classical specialization relationship. Furthermore, M-N relationships are integrated in the model. Relationships can be traversed using traversal paths. The metamodel of the ODMG data model is represented in Figure 1.9 [LAC+93].

The remote object access (ROA) protocol refers to concepts described in federated database schemas to send queries and updates to remote databases. It is based on an extension of SQL-RDA, taking into account the ODMG proposal [Cat93]. A basic query is of the following form:

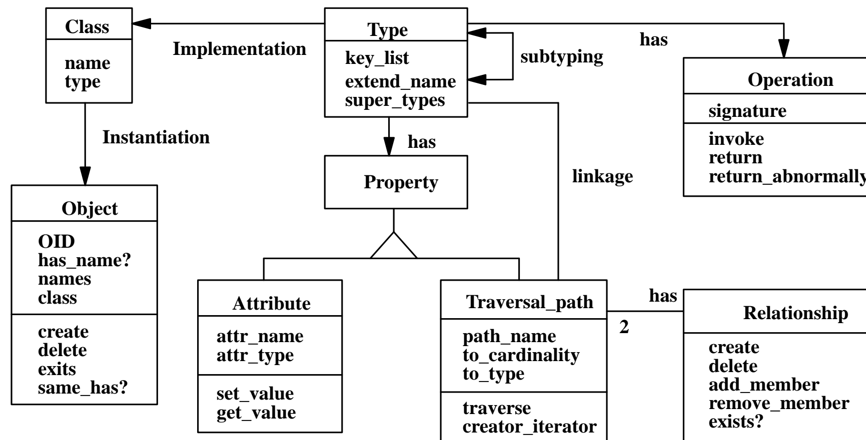

Figure 1.9. The ODMG metamodel (from [LAC+93]).

```
SELECT [<collection>] (<expression>[, <expression>])
FROM  x  in <collection> [, y  in <collection>]
WHERE <formula>
```

The result of the query is either an object or a collection of objects or literals with the specified attribute values and the structure given by the resulting collection type. Any collection type is possible, including a class name. In the latter case, new objects are created. The formula in argument of the WHERE is a logic formula of predicates. It is applied to the Cartesian product of the collections given in argument of the FROM clause. Here again, any collection type is possible, including set, list, class, and so on. As a collection, the result of a query can, in turn, be used in argument of a new query, which gives the ability to nest queries at the FROM level. Selected expressions can also be subqueries, which gives the ability to nest collections within result collections.

In summary, the protocol can be seen both as a specialization of generic RDA for object-oriented databases and as a generalization of the SQL access group SQL RDA to object-oriented databases. The basis of the generalization is the support of types, collections, and methods within SQL, including inheritance and late binding for method handling. At the communication layer also the appropriate transaction management commands are incorporated in the protocol.

Finally, it is important to note that the ODMG data model and query language (i.e., OQL) are used at several layers in the architecture:

- At the local level interface for describing in a more abstract way the schema that can be exported to remote databases (export schemas) and to query the local databases
- At the communication level interface to generate queries and collections of objects as results for allowing user remote access from C to a local database seen through one export schema
- At the interoperation layer for both, defining the integrated schema and generating the C++ classes for integrated accesses.

The communication layer interface does not hide the existence of multiple databases. Instead, it provides a uniform programming interface to access these databases. The communication interface looks like an extension of the SQL access group call level interface (SQL/CLI). The SQL/CLI is developed in a manner that allows a user of the communication layer interface to query multiple database servers. We adopted a client/server architecture based on the remote procedure call approach (RPC) (see for example [AT90]) to implement the SQL/CLI interface (see Figure 1.10). At the communication layer we develop the whole part of SQL/CLI routines. However, at a database server site we develop only those that invoke directly database services. The local database adapter guarantees access to heterogeneous databases. It invokes the database server for appropriate services.

The international standard SQL/CLI provides the following services:

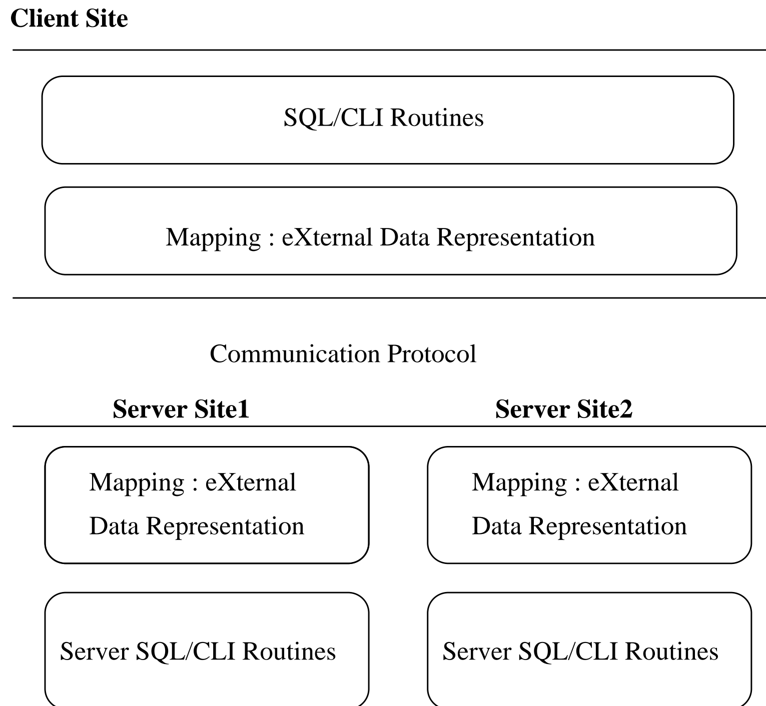

Figure 1.10. SQL/CLI architecture implementation.

- Connections, disconnections, localization of servers services. Data are identified by a name (an external name working as a persistent entry point to a database) and a database name. Given a database name, the communication layer determines the target local DBMS. This service also contains a description of each local system available.

- Query facilities. Based on OOA services, the communication layer provides a standard query language that allows the querying of any local DBMS. Note that a query expressed in OQL can be simply a local method invocation for a given class. This allows the integration of specific DBMSs within the federation, for example, a textual or geographical database system. Specific query languages can be integrated in the common framework as specific methods, without any mapping overhead. This makes the system really open.

The main commands of the OQL/CLI interface are summarized in Figure 1.11. Adaptation to objects requires a few modifications from SQL/CLI. Mainly, a FetchDirect command has been added to retrieve an object from a global oid. FetchDirect will invoke a method at the local level to deliver an object from an oid.

Connect()	Open a connection to a server
Prepare()	Prepare a statement for later execution
Execute()	Execute a prepared OQL statement
ExecDirect()	Execute an OQL statement directly
SetCursorName()	Set the name of the cursor
DescribeAtt()	Describe an attribute of a result object
Fetch()	Get the next object of a result
FetchDirect(Oid)	Get the object for the given object identifier
ObjectCount()	Get the number of objects affected by an OQL statement
Transact()	Commit or roll back a transaction
Cancel()	Attempt to cancel execution of an OQL statement
Error()	Return error information

Figure 1.11. An overview of the OQL/CLI commands.

An outline of the way that the communication layer handles the transfer of complex objects from local buffers to client buffers is appropriate. A buffer management policy has been defined to efficiently interchange complex objects. Cursor management is offered at the OOA level to facilitate the transfer of long objects. Cursor management consists of a set of primitives that allow the browsing of an arbitrary complex structure (tuple, set, list, etc.). This has to be very efficient. Browsing through complex structures is critical in terms of performance. For instance, software engineering applications require efficient scans of module-submodule hierarchies.

1.4.3 Interoperable Object Management and Application Programming Interface

The interoperable object manager (IOM) performs the tasks that are required first for accessing multiple databases in single transactions, and second for hiding multiple databases. Thus, the IOM provides object location transparency. The difference between the communication layer and the interoperation layer is that the functions supported by the communication layer concern a single local DBMS at a time, whereas those supported by the IOM integrate several local DBMSs. In addition, the interoperation layer executes the global algorithms for concurrency control and distributed query processing. It also provides the capability to define integrated schemas as views of the export schemas, as described in Section 1.4.4. The IOM mainly integrates the algorithms required to decompose and map queries (and updates) formulated against integrated schemas to local queries formulated against export schemas. Thus, queries and updates are decomposed at this level of the system into several local subqueries or updates.

Query processing includes techniques similar to distributed query processing [YC84]. However, because of the heterogeneity and the autonomy of the participating databases, there are some additional complexities. Mapping functions have to

be integrated to the query. The component databases may differ in their ability to perform local query optimization. There may be different operations provided by each participating database. Object-oriented query processing techniques should also be integrated [KGBW90]. At the interoperation layer, we have an intermediate representation of the query, which is derived from [LST91]. This is a tree-based representation, in which the root represents the result, and the leaves the collections of instances (i.e., classes and relations) described in export schemas. The query translator processes this tree to obtain a more efficient query tree. This optimized query tree represents the clustering of local and distributed operations, the scheduling of the operations, and the order of the operations (see also [MS91]). Thus, the translator generates a program composed of monosite OQL queries and object moves (see Section 1.4.5).

The transaction manager at the interoperation layer relies on the implementation of the primitive services provided at the local and communication layers. Transactions access multiple databases and thus incorporate the necessary updates to maintain interdatabase dependencies. In addition, the applications using the interoperation services introduce additional characteristics that have to be addressed by the global transaction management. The processing model typically implies implicitly structured data as opposed to the "flat" structure present in the relational platform. The existing database platforms used are autonomous and possibly heterogeneous. They follow, at least to some degree, various different policies for transaction processing. This is in contrast to homogeneous distributed databases, in which a uniform design simplifies the transaction management implementation.

We have addressed the problems at the interoperation layer by the following approaches:

- *Open nested transactions.* Our approach for implementing the transaction manager at the interoperation layer is based on the experience from the prototype of an open nested transaction manager developed by GMD. The results of this experiment are applied to the design of the global transaction manager. IRO–DB applications produce nested processes, which are naturally materialized as nested transactions.

- *Partial visibility and compensation.* The problem of interactive applications, which typically produce long-duration transactions and cause performance deficiencies, are solved by allowing results of subtransactions to be available ("open") before the top-level transaction is committed. This is done in a controlled manner, which guarantees that the global transactions can compensate results of subtransactions in cases where rolled-back subtransaction's results have already been visible to other transactions. The recovery mechanism relies also on compensating the effects of site failures.

- *Multiple acceptable states.* Heterogeneity and partial autonomy of the participating database management systems lead to an environment in which subtransactions in different systems are managed independently of each other.

The open nested transaction manager allows subtransactions to be managed by the different local transaction managers, even if different concurrency control and recovery algorithms are used. In many cases the transactions in the considered application environments can be multithreaded with alternative acceptable final states. This allows the transaction manager to relax the (in practice) too-strict atomicity property, which requires all or none of the subtransactions to commit. The global transaction manager allows a partial execution of a transaction, if some of the subtransactions can produce functionally equivalent results. If such functional redundancy is not available, IRO-DB provides a mechanism based on compensation to cope with unilateral aborts of local subtransactions, either due to a failure or administrative decision.

Object-oriented application interfaces are supplied at the interface of the interoperation layer. An object-oriented application program sees an integrated database as a set of classes (i.e., abstract data types). These classes are defined in the interoperable schema and exported to the programming language structures, for example, as C++ classes. These classes provide methods that allow the program application to manipulate interoperable databases. Creating, retrieving, updating, and deleting objects are performed either by calling appropriate methods defined for the classes or by submitting OQL requests bound to the language. The OQL requests are decomposed by IRO-DB to generate subrequests mapped into the data manipulation language of participating systems and submitted to the appropriate database servers. Retrieved objects through OQL requests are processed as normal language objects.

1.4.4 Schema Integrator Workbench

Independently designed databases typically maintain overlapping information with differences in naming, granularity, degree of abstraction, and scaling. To meaningfully exchange data between them and to provide integrated access to them, these differences have to be resolved. With complex schemas this cannot be achieved realistically by implementing ad hoc mappings between corresponding export schemas as advocated by the multidatabase approach, but requires a methodology to assist the design and maintenance of such mappings.

For this purpose, we develop a schema integrator workbench to design and maintain integrated schemas to which heterogeneous schemas are mapped to resolve differences between corresponding export schemas. However, we do not aim at a complete, global integrated schema, which overcomes all heterogeneities, but rather want to assist the incremental design and maintenance of integrated schemas, according to the specific needs of an integrated application. This is achieved on the basis of a declarative methodology for schema integration. Users can declare correspondences between schema constituents that contain overlapping instances according to their integrated application. To allow for the resolution of differences in representation, we consider not only correspondences between constituents of

equal kind, as between two classes from different schemas, or between their direct attributes, but also for correspondences between classes and attributes, and, more importantly, between paths composed of several references between classes. The declared correspondences are checked for consistency, and from consistent correspondences an integrated schema is generated, together with mappings from the heterogeneous corresponding export schemas. The methodology is based on a flexible notion of schema unification, which allows for augmenting the structural granularity of corresponding schemas in order to overcome their heterogeneity. (For a more detailed overview, see Chapter 15 on *Database Integration Using the Open Object-Oriented Database System VODAK*.)

More precisely, the following components are provided:

1. *Dictionary for schema support.* The dictionary is a central component of the schema integrator workbench. It contains export schemas and the integrated schemas that result from the integration process. We design the dictionary as an object-oriented database, following simplified OMG specifications.

2. *Schema explorer for retrieving relevant constituents in export schemas.* Integration methodologies devised for the global schema approach were applied to rather small-scale applications, involving schemas of limited complexity. Thus, they employ mostly structural considerations to determine resembling schema constituents. In large-scale federated databases, in which each database schema can consist of thousands of constituents, purely structural resemblance mechanisms are likely to be too imprecise. Thus, the schema explorer uses semantic knowledge in the form of a fuzzy conceptual knowledge base not necessarily complete. Inferring from this knowledge base, it retrieves corresponding constituents in the export schemas.

3. *Rule base for goal-oriented integration.* These rules are used to test the correspondences detected with the help of the schema explorer for consistency, to infer the necessary transformations for overcoming differences in representation, and to generate the accordingly integrated representation.

4. *Graphical tools.* The integration methodology constituted by the rule case and the schema explorer is interfaced to users by a graphical schema editor, which allows for browsing through export schemas at various levels of detail, for selecting constituents of interest, for presenting constituents retrieved by the schema explorer, and for performing declarative assertional integration operations in a direct manipulation paradigm.

1.4.5 Global Query Decomposition

The global query optimizer processes queries expressed on interoperable schemas and generates query plans in terms of monosite queries expressed on exported schemas and object transfers. Input OQL queries are expressed in term of a syntactic tree. A syntactic tree is a tree whose edges represent syntactic elements and

nodes, either connectors or objects. It is a direct result of the query parsing. Let us assume the integrated view of Figure 1.12. Figure 1.13 gives an example of a syntactic tree representing the following query, which retrieves a master production schedule, that is, prt_id, date, quantity, factory, for all parts that are to be manufactured by continuous flow manufacturing, CFM, and that are due before 01-01-95:

```
SELECT struct <part: x.prt_id, date_due: z.mps_date,
              quantity: z.quantity, factory: y.fcr_id>
FROM   x in PRODUCED_PART, y in x.is_manufactured, z in x.mps
WHERE  z.mps_date <= '01-01-95'
  AND  x.prt_type = 'CFM';
```

The query decomposer applies the following transformation to a request:

1. *Expression on export schemas*: As the global query is expressed on an interoperable schema, it has to be mapped to the constituent export schemas.

```
interface PART {
     key prt_id;
     attribute String prt_id;
     attribute String prt_dsc;
     attribute Date prt_upd_date;
     attribute Char(3) prt_type  };

interface PRODUCED_PART: PART {
     attribute Float(8) planned_qty;
     attribute mps: Set < Struct <Date mps_date,
                 Float(8) quantity, Char(1) quantity_type > > };
     relationship Set<FACTORY> is_manufactured
                 inverse FACTORY::manufactured_part };

interface FACTORY {
     key fcr_id;
     attribute Char(2) fcr_id;
     attribute String fcr_dsc;
     relationship Set<PRODUCED_PART> manufactured_part
                 inverse PRODUCED_PART::is_manufactured };
```

Figure 1.12. An interoperable view example.

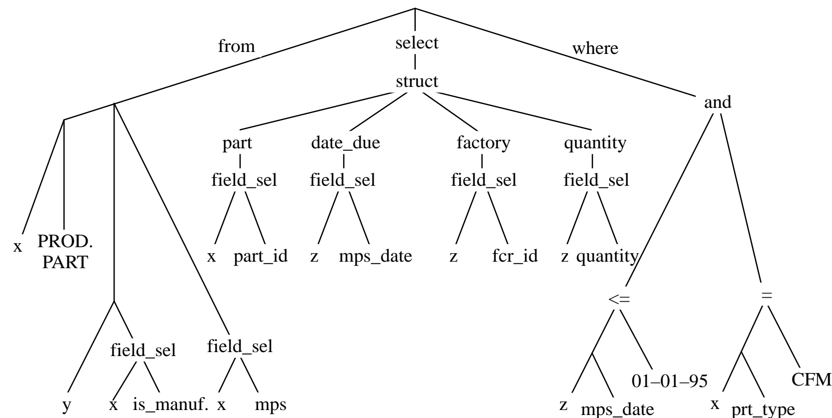

Figure 1.13. A syntax tree example.

This step has to take into account the mapping information stored in the client repository. It is, in principle, similar to the query modification process of classical DBMSs.

2. *Optimization of the query tree*: This step requires reordering of operations. Simple heuristics, such as move restriction on values and reference, joins close to the class extensions could be applied. A simple cost model based on class and object sizes could be considered for deciding on the opportunity of a transformation.

3. *Generation of the execution plan*: This step requires the generation of maximum subqueries expressed in OQL and consideration for locating intermediate results. A list of possible sites for intermediate results could be implemented, and the choice could be made at run-time according to the size of intermediate results and the estimated cost of operations.

1.4.6 Repository Management

Object descriptions are stored in a repository on server and clients. The repository is based on the ODMG metamodel and tries to be compatible with the CORBA proposal. Figure 1.14 summarizes the generalization hierarchy of the OMG dictionary, which is organized in container and contained objects. This hierarchy is convenient for defining interfaces. It is going to be implemented in IRO-DB.

Each object should have a description in accordance with the OMG architecture (i.e., CORBA), including name and oid. At first, each object (repository, database, class, attribute, type) should have four associated operations for creating and retrieving descriptions:

- description create(name,) = creation of a description

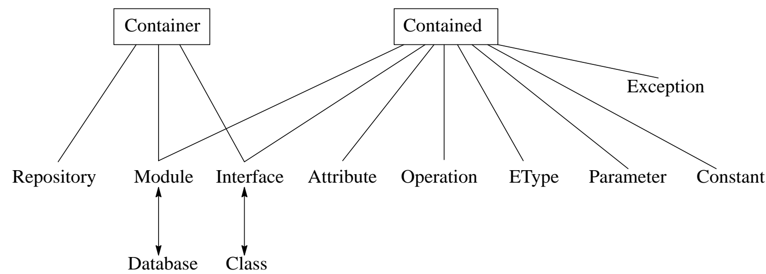

Figure 1.14. Architecture of the repository.

- void delete(oid) = deletion of a description
- oid lookup(name) = retrieval of an object by name
- description describe(oid) = retrieval of a description

Further possible operations are described in [OMG92]. In particular, it includes an operation to retrieve, in which object is a contained object (within(...)), an operation to find the list of objects within a container (contents(...)), to look for a specific object by name (lookup_name(...)), and to get a description of objects within a container (describe_contents(...)). Such operations could be implemented as needed.

1.5 Conclusion

In this paper, we presented an overview of the IRO-DB federated object-oriented database system. IRO-DB federates relational and object-oriented databases through an object-oriented data model with an associated object SQL derived from the ODMG proposal. The overall aim of the project, the provision of building blocks for federated database management, has been recognized world-wide as one of the most important challenges in database research and development. The project develops suitable object-oriented specializations of the SQL Access Group and RDA standards for the exchange of complex objects and data, also including protocols for transaction processing in heterogeneous environments, and taking into account relevant industrial data interchange standards or proposals, such as the OMG architecture and model.

It designs and implements a layered set of tools for interoperating autonomous databases and applications, realizing a smooth migration from merely interconnected information services without any transparency, over loosely coupled federations, to tightly coupled federations providing for semantically integrated exchange and merge of heterogeneous information. The project clearly distinguishes the local layer, the communication layer, and the interoperable layer. They all have the same

ODL/OQL interface. The local layer is basically a C library that overcomes syntactical differences among databases. It provides export schemas defined in terms of types and relationships, with a SELECT function that applies to a Cartesian product of collections. The communication layer gives the required interface and protocol to transfer queries and objects from a client to a server and vice versa. The interoperable layer provides localization transparency and performs the required query decomposition and update synchronization. It relies upon a home ODBMS through virtual classes and enhances the query facilities by achieving distributed heterogeneous accesses.

The IRO-DB project is a joint effort in Europe, which integrates components developed by various partners. A first design phase is currently ending, and implementation of components is starting in the various involved companies and research centers.

Bibliography

[ASD+91] R. Ahmed, P. de Smedt, W. Du, W. Kent. M. Ketabchi, W. Litwin, A. Rafii, and M.-C. Shan. The Pegasus Heterogeneous Multidatabase System. *Computer*, 24:12 (December 1991), pp. 19–27.

[AT90] AT&T. *UNIX System V Release 4 Programmer's Guide : Networking Interface.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[BCD+93] O. A. Bukhres, J. Chen, W. Du, A. K. Elmagarmid, and R. Pezzoli. InterBase: An Execution Environment for Heterogeneous Software Systems. *IEEE Computer*, 26:8, August 1993, pp. 57–69.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18:4 (December 1886), pp. 232–364.

[Cat93] R. Catell. *Object Databases: The ODMG-93 Standard.* Morgan Kaufmann, San Francisco, 1993.

[DL87] P. Dwyer and J. Larson. Some Experiences with a Distributed Database Tesbed System. *Special Issue on Distributed Database Systems, Proceedings of the IEEE*, 75:5, 1987, pp. 633–647.

[OMG92] Object Management Group. Object Services Architecture. Technical Report 92.8.4, OMG, 1992.

[HR90] S. Hayne and S. Ram. Multi–User View Integration System (MUVIS): An Expert System for View Integration. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990, pp. 402–409.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next–Generation Database System. *IEEE Transactions on Data and Knowledge Engineering*, 2:1 (March 1990), pp. 109–124.

[LA87] W. Litwin and A. Abdellatif. An Overview of the Multidatabase Manipulation Language MDSL. In *Proceedings of the IEEE*, 1987, pp. 621–632.

[LAC+93] M. Loomis, T. Atwood, R. Cattelland J. Duhland G. Ferran, and D. Wade. The ODMG Object Model. *Journal of Object-Oriented Programming*, 6:3 (June 1993), pp. 64–69.

[LBE+82] W. Litwin, J. Boudenant, C. Esculier, A. Ferrier, A. Glorieux, J. La Chimia, K. Kabbaj, C. Moulinoux, P. Rolin, and C. Stangret. SIRIUS Systems for Distributed Data Management. In H.-J. Schneider, ed., *Distributed Data Bases.* North Holland, The Netherlands, 1982, pp. 311–366.

[LR82] T. Landers and R. Rosenberg. An Overview of Multibase. In H.-J. Schneider, ed., *Distributed Data Bases.* North Holland, The Netherlands, 1982, pp. 153–184.

[LST91] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of Multi–Way Join Queries for Parallel Execution. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, 1991, pp. 549–560.

[MBE95] J. Mullen, O. Bukhres, and A. Elmagarmid. InterBase*: A Multidatabase System. In O. Bukhres and A. Elmagarmid, editors, *Object Oriented Multidatabase Systems: A Solution for Advanced Applications*, chapter 19. Prentice Hall, Englewood Cliffs, N.J., 1995.

[ME93] J. G. Mullen and A. Elmagarmid. InterSQL: A Multidatabase Transaction Programming Language. In *Proceedings of the 1993 Workshop on Database Programming Languages*, 1993.

[ML87] R. McCord and B. Linn. Ingres Star. In *Proceedings of the 1987 ACM Sigmod Conference, Product Session and Tutorial*, 1987.

[MS91] M. Murphy and M.-C. Shan. Execution Plan Balancing. In *Proceedings of the Seventh International Conference on Data Engineering*, 1991, pp. 698–706.

[NB77] E. Neuhold and H. Biller. POREL : A Distributed Data Base on an Inhomogeneous Computer Network. In *Proceedings of the Third International Conference on VLDB*, Tokyo, 1977, pp. 380–395.

[NS88] E. Neuhold and M. Schrefl. Dynamic Derivation of Personalized Views. In *Proceedings of the Fourteenth International Conference on VLDB*, 1988, pp. 183–194,

[RR89] A. Rosenthal and D. Reiner. Database Design Tools: Combining Theory, Guesswork, and User Interaction. In *Proceedings of the Eighth International Conference on the Entity Relationship Approach to Database Design and Querying*, Toronto, Canada. North Holland, The Netherlands, 1989, pp. 187–201.

[Shi81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *IEEE Transactions on Database Systems*, 6:1 (March 1981), pp. 140–173.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22:3 (September 1990), pp. 183–236.

[SLCN88] A. Sheth, J. Larson, A. Cornelio, and S. Navathe. A Tool for Integrating Conceptual Schemata and User Views. In *Proceedings IEEE Data Engineering Conference*, 1988, pp. 176–183.

[TBC+86] M. Templeton, D. Brill, A. Chen, S. Dao, and E. Lund. Mermaid-Experiences With Network Operation. In *Proceedings IEEE International Conference on Data Engineering*, 1986, pp. 292–300.

[TBC+87] M. Templeton, D. Brill, A. Chen, S. Dao, E. Lund, R. McGregor, and P. Ward. Mermaid: a front end to distributed heterogeneous databases. In *Special Issue on Distributed Database Systems, Proceedings of the IEEE 75*, May 1987, pp. 695-708.

[TLW87b] M. Templeton, E. Lund, and P. Ward. Pragmatics of Access Control in Mermaid. *Quartely Bulletin IEEE Technical Committee on Database Engineering*, 10:3 (September 1987), pp. 33–38.

[TTC+90] G. Thomas, G. Thompson, C.-W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22:3 (September 1990), pp. 237–266.

[Wei91] G. Weikum. Principles and Realization Strategies of Multi–Level Transaction Management. *ACM TODS*, 16:1 (March 1991), pp. 132–180.

[YC84] C. Yu and C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16:4 (December 1984), pp. 399–433.