Metadata Management in Federated Multimedia Systems*

Mark Roantree

School of Computer Applications
Dublin City University,
Glasnevin, Dublin 9, Ireland.
Email: mark.roantree@compapp.dcu.ie

Abstract

A Federated Information System requires that multiple (often heterogenous) information systems are integrated to the extent that they can share data through views. One issue faced during the construction of federated view schemata is the continuous need to extract metadata from cooperating systems. This is more pressing in the case of federated multimedia systems where needless transfer of large binary objects affects system performance. Where participating systems employ an object-oriented common model to communicate, obtaining metadata is problematic due to the type and complexity of object-oriented metamodels. In the IOMPAR project, we specified and developed a high-level query interface for the ODMG schema repository in order to simplify this task for integration system engineers. This paper describes a schema repository query language which greatly reduces the efforts of integration engineers.

Keywords: Federated Databases, Interoperability, Metadata, ODMG, Object-Oriented Databases.

1 Introduction

Database applications often require a means by which 'generic' applications can determine a database's structure at run-time for functions such as graphical browsers, dynamic query construction, and the specification of view schemata. This property, often referred to in programming languages as *reflection*, has been a feature of databases for many years, and an early version of the ODMG metamodel [Cattell and Barry, 1997] provided a standard API for metadata queries in object-oriented databases. As part of our earlier research into federated healthcare systems [Roantree et al. (2), 2001], we specified and implemented a global view mechanism to facilitate the creation of views for ODMG databases, and the subsequent integration of view schemata to form federated schemata. Please refer to [Pitoura et al., 1995, Sheth and Larson, 1990] for a complete background on federated databases. As the integration of information systems requires many metadata queries to determine structural and semantic heterogeneities, we focus on the construction of a metadata interface to ODMG information systems. This paper is structured as follows: the remainder of this section highlights the importance of metadata to our view mechanism, and the motivation for this research; in §2 the main concepts of the metadata query language are discussed; in §3 the query language is presented through a series of examples; in §4 we present details of the implementation; and finally in §5 we offer some conclusions.

1.1 Background and Motivation

The IOMPAR project involves research into secure federated multimedia systems [Roantree, 2000], and while reusing much of the output and engineering tools from the OASIS project [Roantree et al., 1999], it faces new problems in dealing with the transfer of large multimedia objects, and key management and encryption issues for secure communication. The main focus of OASIS was to extend the ODMG 2.0 model to provide views in a federated database environment. This work yielded the specification and implementation of a global view mechanism, using the ODMG model as the common model for a *federation* of databases. The concept of a federation of databases [Sheth and Larson, 1990] is one where heterogeneous databases (or information systems) can communicate with one another through an interface provided by a common data model. In the OASIS and IOMPAR projects, the common data model is the ODMG model, the standard model for object-oriented databases since 1993 [Cattell, 2000]. The most common architecture for these systems is as follows: data resides in many (generally heterogeneous) information systems or databases; the schema of each Information System (IS) is translated to an O-O format, and this new schema is called the component schema; view schemata are defined as shareable subsets of the component schema; the view schemata are exported to a global or federated server where they are integrated to form many global or federated schemata. In this paper we use the term *view* (or ODMG view) to refer to an ODMG subschema which may contain multiple classes, where those classes may be connected through either inheritance or association relationships.

Our focus was to extend the ODMG model so that it was possible to define the view schemata on top of each component schema, and define integration operators which facilitated the construction of federated schemata. This extension provided a layer of ODMG views on top of the component schema. However, it was also necessary to provide a mapping language which could bind the component schema to its local model representation. This facilitates the translation of ODMG queries (to the their local IS equivalent), and enables data transfer to the ODMG database when views are defined. In IOMPAR the same view mechanism is used although it has been adapted to the ODMG 3.0 standard, and is currently being implemented for an object-relational database (ORDB) in order that ORDBs can define semantically rich views, and that both database system models provide a standard view interface. The effect is that the ORDB appears to provide an object-oriented

*Funded by Enterprise Ireland Strategic Research Grant ST/2000/091

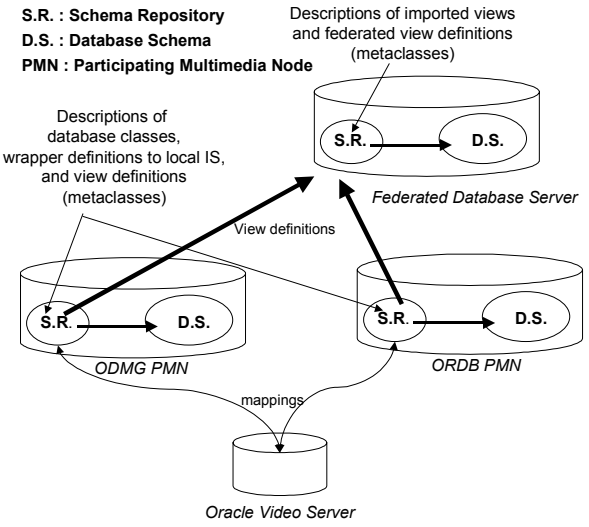

Figure 1: *Federated Multimedia Architecture.*

view mechanism.

The classes which comprise the database schema are used to model the real world entities which are stored in the database. Additionally, there is a set of metaclass instances which are used to describe the database classes. Thus, one can think of an ODMG database as having two distinct sets of classes: those which reside in the *database schema*, and the abstract classes (metaclasses) which reside in the *schema repository*. Whenever it is necessary to process and store a *view* definition, a set of metaclass instances are stored in the database. Where a view definition involves multiple classes, each with their own extents, this combination of meta-objects can become quite complex. Thus, a command to display view data, or to extract data from the local IS often requires powerful query facilities in order to retrieve the required meta-information. In *figure 1* the role of the schema repository within a federated database environment is illustrated. In federated database terminology, the Oracle Video Servers act as local database systems, and the ODMG and ORDB databases act as component databases. In reality, the federation can consist of many multimedia servers including ODMG servers where the ODMG database is at the level of Oracle Video Server in *figure 1*. The global (or federated) database will always be an ODMG database. Each schema repository contains a description of the database classes, hence the arrow towards the database schema. However, in this type of architecture, the schema repository will contain a large amount of additional data describing both view definitions and query results. Before the transfer of multimedia data, it is necessary to determine the size of result sets to decide if that transfer is necessary or worthwhile.

Due to the complex nature of both the base metamodel and extensions, many of the OQL queries which are required both to retrieve base and view class metadata would necessitate long expressions. In practical terms (using an ODMG vendor database), it was not possible to express all of the desired queries. In IOMPAR, the focus is on creating federations of multimedia databases which use ODMG 3.0 databases, Oracle Video Server and Object-relational databases as storage devices. The contribution of the work in this paper is the specification of language extensions to OQL for the specific purpose of efficient retrieval of metadata from the schema repositories of ODMG databases, and object-relational system catalogs. Two clears benefits have emerged: the reduced complexity of metadata queries during system integration (and federated schema construction); and a reduced learning curve for programmers who need to use the ODMG schema repository.

2 Query Language Concepts

In this section we provide a description of the main concepts involved in the Query Language: the metadata objects and the queries used to manipulate metadata. Metadata objects are used to describe both the structural elements of the participating systems (base metadata objects), and the virtual elements which are generated by the view language. Note that virtual entities (classes, properties etc.) are always mapped to base entities. Metadata queries are categorized into groups representing the type of metadata information required. These groupings are later described in §2.2.

2.1 Metadata Elements

For an ODMG DBMS containing a view mechanism, it is necessary to distinguish between base and virtual classes, and in both cases one must be capable of obtaining the same structural information. The view mechanism, its specification language, and implementation are described in [Roantree et al. (2), 2001]. This paper assumes that view definitions have already been processed and stored, and a requirement exists to retrieve meta-information in order to display views or process global queries. A view is represented by the meta-objects outlined below. See [Jordan, 1998] for a similar description of base metadata object construction.

1. **Subschema Construction.** The definition of a virtual subschema requires the construction of a `v_Subschema` instance.
2. **Class Construction.** Where it is necessary to construct new virtual classes, a `v_Class` object is instantiated for each new virtual class.
3. **Attribute and Relationship Construction.** A single `v_Attribute` instance and a `v_Primitive_Type` instance is constructed for each attribute property, and a `v_Relationship` and `v_Ref_Type` instance is constructed for each relationship property.
4. **Inheritance Construction.** A `v_Inheritance` meta-object connects classes to subclasses.
5. **Class Scope.** When `v_Attribute` and `v_Relationship` instances are constructed, it is necessary to associate these properties with a specific `v_Class` instance. This is achieved by updating the `v_Scope` object which the `v_Class` object inherits from.
6. **Subschema Scope.** When `v_Class` instances are constructed, it is necessary to associate these virtual classes with a specific `v_Subschema` instance.

2.2 Metadata Query Language

The Schema Repository Query Language (SRQL) is an extension to ODMG's Object Query Language, and resides between the client database application and the ODMG database. The language comprises seventeen productions listed in the appendix to this paper. In this section we provide an informal description of the types and usage of query language expressions. The language resembles OQL in the fact that

it employs a `select` expression. However, SRQL expressions employ a series of keywords, and are always single line expressions. As shall be demonstrated in the next section, this has practical advantages over using standard OQL to retrieve metadata information.

- **Subschema Expressions**. This type of query is used to retrieve subschema objects, which are container objects for all elements contained within a view definition. The `subschema` keyword identifies this type of expression.
- **Class Expressions**. This type of query can be used to retrieve specified base or virtual class objects, the entire set of base class objects, the entire set of virtual class objects, or the set of virtual classes contained within a specified schema. The `class` keyword identifies the type of expression, with the qualifier `virtual` specified for virtual classes, and a further qualifier `in` used when retrieving virtual classes for a specific subschema (or view).
- **Attribute Expressions**. This type of query is used to retrieve single base or virtual attribute objects, the entire set of base or virtual attribute objects, or all attributes for a specific base or virtual class, by specifying the `attribute` keyword. The query can also be expressed in a shallow form (retrieve only those attributes for the named class) or a deep form (retrieve attributes for the named class and all derived classes). The qualifiers (`virtual` and `in`) are used in attribute query expressions, and a further qualifier `inherit` is used to determine between shallow and deep query expressions.
- **Relationship Expressions**. This type of query is similar to attribute queries. Syntactically, the `attribute` keyword is replaced with the `relationship` keyword.
- **Link and Base Expressions**. These queries return the meta-objects to which virtual objects are mapped. In a view mechanism, each virtual element which has been generated as a result of a view definition must map to an equivalent base or virtual element. For example, a virtual attribute object may map to another virtual attribute object, which in turn maps to a base attribute object. The `link` query expression will return either a virtual class, attribute or relationship object if the specified object is mapped to a virtual element, or NULL, if it is mapped directly to a base element. The base query expression will always return either a base class, attribute or relationship object, but never NULL as *all* virtual elements must eventually map to a base element.
- **MetaName and MetaCount Expressions**. Both query expressions take a single SRQL expression as an argument and return the names of the meta-objects and the count of the meta-objects respectively.
- **Type Expressions**. This query is used to return the type of (base or virtual) attribute or relationship meta-objects. Each ODMG attribute and relationship type is taken from a predefined set of types.
- **Data Count Expressions**. This query is used to retrieve information regarding the number of objects in a result set, or a view class extent.
- **Data Size Expressions**. This query is used to retrieve the size of objects in result sets or view class extents.

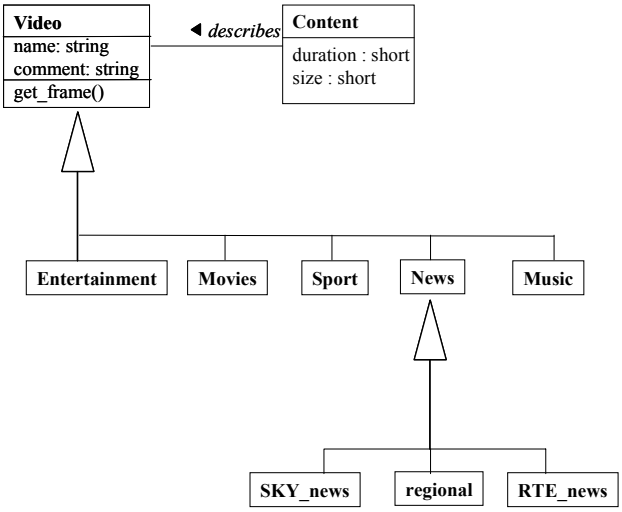

Figure 2: *Partial structure of the local_ news view.*

3 Using the SRQL

The ODMG model provides an API specification for access to the schema repository. However, it is quite complex and often difficult to formulate OQL metadata queries as shall be shown. Since metadata queries can be regarded as a small static group of queries, we have developed a query sub-language for the ODMG schema repository. This query language is based on OQL but extends the base language with a series of constructs which are specifically employed in metadata querying. For this reason, we called this metadata sub-language, the Schema Repository Query Language (SRQL). An early version of this language was presented in [Roantree et al. (1), 2001], and this has now been updated for ODMG 3.0 and object-relational databases, and with query expressions for determining the size of multimedia objects.

3.1 Sample Metadata

The ODMG view can have any number of base or (newly derived) virtual classes, and some of these classes are connected using inheritance or relationship links. Where a view contains both base and virtual classes, it is not possible to connect classes from both sets. In this case, the view contains disjoint schema subsets. A view definition is placed inside an Object Definition Language View file (ODL$_v$ file), passed through the *View Processor* to generate a set of meta-objects in the database's schema repository. These view descriptions are then exported to a global database for integration with other view schemata. It is these meta-objects which are queried by system integrators as they seek to discover similarities and differences between schemata which are to be merged. The IOMPAR project primarily deals with sharing multimedia objects, where objects are stored in *Participating Multimedia Node* (LMN) databases which are often connected to Oracle Video Server. For the purpose of the examples used later in this paper, assume that the view illustrated in *figure 2* contains video files, companion data and metadata, and is stored as *local_ news* in the database. The *Video* class has five subclasses and one of these, the *News* subclass, has a further three subclasses. The structure of these classes is not shown for reasons of space. The types of meta-objects constructed for this view (classes, properties etc.) were described in §2.1.

The ODMG metamodel through its C++ API provides an 'open' standard for retrieving ODMG meta-

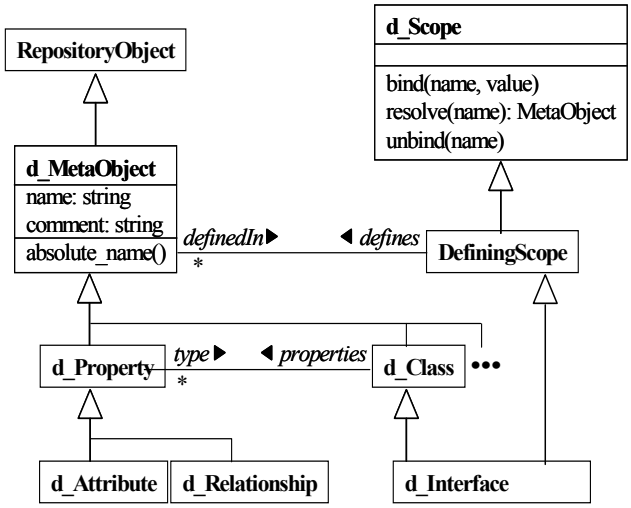

Figure 3: *API subset of the ODMG metamodel.*

data, and a basis for developing a method to insert data into the schema repository. This provides a powerful mechanism both for creators of dynamic software applications (views and dynamic querying), and federated database engineers, whose role it is to extract schema information from participating systems. In practical terms however, it is not possible to build truly generic object-oriented database applications due to the imprecise specification of the standard and the heterogeneous interpretations of ODMG vendors. For developing pure generic applications, a number of additional features are required. Such a discussion is outside the scope of this paper but is discussed in [Roantree and Subieta, 2001] where possible solutions and improvements to the standard are offered.

3.2 Metadata Query Samples

In this section, we demonstrate why the language was developed by illustrating a series of metadata queries using conventional OQL, and demonstrate how these queries might be simplified using an extension to OQL. In addition to using 'pure' OQL syntax, we have also opted to use the C++ bindings as provided by the Versant O-O database product [Versant, 1999]. Note that we use the term *oid* instead of the vendor term *SelfOid*, as *oid* is more generic. The motivation was to ensure that these OQL queries can actually be expressed in at least one vendor product, and to demonstrate the mappings between 'native' OQL and a typical O-O database vendor. Most of the examples use the *local_news* view which is partially displayed in *figure 2* and represent a federation of video clips from multiple video servers. These examples demonstrate how a query language based on OQL could be used to simplify querying operations against the schema repository. These types of queries are crucial to schema integrators who require metadata information in order to determine the structural makeup of a schema, before subsequently restructuring and merging different schemata. Initial queries when connecting to a database for the first time will be: *what are the names of export schemata? What are the names of classes within a specific export schema? How many attributes does a particular class contain? What is the type of attribute x?* In examples 3.1 to 3.3 it is assumed that this type of data has already been acquired, and more specific class level meta-information is needed. In the following examples metaclasses with a *d_* prefix are ODMG base metaclasses [Jordan, 1998], and those with a *v_* prefix represent the extension metaclasses originally specified in the OASIS project [Roantree et al. (2), 2001]. A small subset of the complex ODMG metamodel is shown in *figure 3* to illustrate how the major metaclasses relate to each other.

Example 3.1: *Retrieve a base class object called 'RTE_news'.*

This query must use the `d_Meta_Object` superclass (of `d_Class`), although a set of references to `d_Class` objects are generated as output.

Example 3.1(a): ODMG OQL

select C **from** d_Class
where C.name = 'RTE_news'

Example 3.1(b): Vendor OQL

select oid **from** d_Class
where d_Meta_Object::name = 'RTE_news'

The mapping between the ODMG specification and the C++ implementation is clear by the vendor OQL expression in 3.1(b). The SRQL equivalent is provided in *3.1(c)* below.

Example 3.1(c): SRQL

select class RTE_news

The result of this query will be the set of base class (`d_Class`) instances which have the `name` attribute value of *RTE_news*. For base class instances, this will always be a single object reference, but for virtual classes it is possible for many to share the same name, providing they belong inside different subschemata. This is explained in [Roantree et al. (2), 2001] where each subschema has its own scope and class hierarchy, and thus class names can be repeated across different view definitions.

Example 3.2: *Retrieve virtual class object called 'RTE_news' from subschema local_news.*

Example 3.2(a): ODMG OQL

select C **from** v_Class
where C.name = 'RTE_news'
and C.SchemaContainer =

(**select** S **from** v_SubSchema **where** S.name = 'local_news')

Example 3.2(b): Vendor OQL

s_oid = (**select** oid **from** v_Subschema
where v_Meta_Object::name = 'local_news');
select oid **from** v_Class
where v_Meta_Object::name = 'RTE_news'
and SchemaContainer = s_oid;

Example 3.2(c): SRQL

select virtual class local_news.RTE_news

In this example, the SRQL makes the OQL query easier as a subschema qualifier is used to specify the correct class. It was necessary to break the vendor query into two segments as it was not possible to pass object references from an inner query. With our SRQL approach in *example 3.2(c)*, the implementation is hidden behind the language extensions.

Example 3.3: *Retrieve a virtual attribute 'broadcast_date' from class 'RTE_news' in subschema local_news.*

In this example we again have the problem of retrieving the correct `v_Class` reference and then selecting the appropriate `v_Attribute` object reference. Assume that the virtual class is from the same subschema (*local_ news*) as the previous example.

Example 3.3(a): ODMG OQL

select A **from** v_Attribute
where A.name = 'broadcast_date'
and A.in_class in
(**select** C **from** v_Class
where C.name = 'RTE_news'
and C.SchemaContainer in
(**select** S **from** v_SubSchema
where S.broadcast_date = 'local_news'))

Example 3.3(b): Vendor OQL

s_oid = (**select** oid **from** v_Subschema
where v_Meta_Object::name = 'local_news');
c_oid = (**select** oid **from** v_Class
where v_Meta_Object::name = 'RTE_news'
and SchemaContainer = s_oid);
select oid **from** v_Attribute
where v_Meta_Object::name = 'broadcast_date'
and v_Attribute::in_class = c_oid);

In *example 3.3(a)* the pure OQL version of the query is expressed by adding another layer to the nested query. However, the vendor product in 3.3(b) requires three separate queries, and thus, it will be necessary to embed the OQL inside a programming language such as C++ or Java. Since this is the most likely scenario for an O-O database program, it does not raise any major problems, but it does demonstrate the unwieldy nature of some of the OQL implementations when building O-O database software. In *example 3.3(c)* the SRQL version of the query is illustrated.

Example 3.3(c): SRQL

select virtual attribute
local_news.RTE_news.broadcast_date

In the following examples we will illustrate more general queries, but will use only OQL and SRQL as the problem regarding vendor-specific versions of OQL has now been shown.

Example 3.4*: Retrieve all classes within subschema local_ news*

In *examples 3.4(a) and (b)* the syntax for both expressions to retrieve all references to `v_Class` objects within the subschema *local_ news* is illustrated. As these queries are simpler than those in previous examples, the OQL expressions are straightforward.

Example 3.4(a): ODMG OQL

select C **from** v_Class
where C.SchemaContainer =
(**select** S **from** v_SubSchema
where S.name = 'local_news')

Example 3.4(b): SRQL

select virtual class in local_news

The keyword `virtual` is used to distinguish between base and virtual classes, but this *predicate* can be dropped in circumstances where all instances are required. In *example 3.4(c)* five possible formats are illustrated. The semantics for the selection of base classes is clear: in example (i) the complete set of `d_Class` object references is returned, and in example (ii) a single `d_Class` reference is returned. For virtual classes, there are three possibilities with examples (iii) and (v) similar to their base query equivalents. However, example (iv) is different: all virtual classes called *RTE_ news* are returned.

Example 3.4(c): class selection formats

(i) **select class**
(ii) **select class** RTE_news
(iii) **select virtual class**
(iv) **select virtual class** RTE_news
(v) **select virtual class** local_news.RTE_news

A subschema can comprise both base and virtual classes [Roantree et al. (2), 2001]. The `in` keyword was used in the previous section to select classes within a specified subschema. If base classes are required, the keyword `virtual` is dropped. Both formats are illustrated in *example 3.4(d).*

Example 3.4(d): retrieve classes within a specified subschema

select class in local_news
select virtual class in local_news

Example 3.5*: retrieve all relationships within RTE_ news within the local_ news schema.*

In this example a reference to all relationship objects inside the *RTE_ news* class is required.

Example 3.5(a): ODMG OQL

select R **from** v_Relationship
where R.defined_in_class **in**
(**select** C **from** v_Class
where C.name = 'RTE_news'
and C.SchemaContainer =
(**select** S **from** v_SubSchema
where S.name = 'local_news')

Example 3.5(b): SRQL

select virtual relationship in
local_news.RTE_news

In *example 3.5(b)* it is clear that the SRQL format is far easier to express than the base OQL query. Additionally, queries regarding inheritance are unwieldy due to the complexity of the O-O model. This is demonstrated in *example 3.6.*

Example 3.6*: Retrieve all attributes, including derived ones for the class Radio_ Sport.*

Assume it were necessary to retrieve all attributes for class *Radio_ Sport*, which is derived from classes *Radio*, *Audio*, *Sport*, and *Recent* (in subschema *Sports_ Coverage*).

Example 3.6(a): ODMG OQL

select A **from** v_Attribute
where A.in_class **in**
(**select** C **from** v_Class
where C.name = 'Radio_Sport'
and C.SchemaContainer in

(**select** S **from** v_SubSchema

where S.name = 'Sports_Coverage')
union
select A **from** v_Attribute
where A.in_class **in**
(**select** i.inherits_to **from** v_Inheritance
where i.inherits_to.name = 'Radio_Sport'
and i.inherits_to.SchemaContainer in
(**select** S **from** v_SubSchema

where S.name = 'Sports_Coverage'))

In *example 3.6(a)* the OQL query to return the required `v_Attribute` references for the class *Radio_Sport* is illustrated. In the first segment (before the union operator is applied) it is necessary to provide nested queries to obtain the correct `v_Subschema` instance, and then the correct `v_Class` instance, before the attributes for class *Radio_Sport* are retrieved. In the second segment, it is necessary to retrieve all `v_Class` references which are superclasses of class *Radio_Sport*, and perform the same operations on these classes. This is not shown in full in 3.6(a) due to the length of the query expression.

Example 3.6(b): SRQL

select virtual attribute in

Sports_Coverage.Radio_Sport **inherit**

Using the SRQL, the query expression is very simple: the keyword `inherit` is applied to the end of the expression to include the additional `v_Attribute` objects in the result set.

Both the `select attribute` and `select relationship` expressions can take different forms as illustrated in *example 3.6(c)*. Only examples (iii) and (vii) will definitely return a collection containing a single object reference.

Example 3.6(c): attribute selection formats

(i) **select attribute**
(ii) **select attribute** duration
(iii) **select attribute** RTE_news.duration
(iv) **select virtual attribute**
(v) **select virtual attribute** duration
(vi) **select virtual attribute**
RTE_news.duration
(vii) **select virtual attribute**
local_news.RTE_news.duration

The area of query transformation and the resolution of mappings between virtual and (other virtual objects and) base objects requires a different form of metadata query expression. Suppose it is necessary to retrieve the base attribute to which a particular virtual attribute is mapped.

Example 3.7: *retrieve mapped attribute (without SRQL)*

Assume that *local_news.RTE_news.broadcast* is mapped to *RTE_news.date* in the base schema. It is necessary to retrieve the mapped attribute *date* to assist in the query transformation process. Assuming the query expressed in *example 3.3* returns an object reference *R* (the *broadcast* attribute in *RTE_news* class in *local_news*), then *example 3.7(a)* can be used to retrieve its mapped base attribute.

Example 3.7(a): ODMG OQL (requires result set R)

select A.VirtualConnector **from** v_Attribute
where A **in** R

Example 3.7(b): SRQL (full query expression)

select link attribute

local_news.RTE_news.broadcast

In *example 3.7(b)* the entire query expression is illustrated. Whereas the basic OQL expression requires three nested queries, the entire expression using SRQL can be expressed in a single `select link` statement. The resolution of mappings becomes even more complex when there are a series of mappings from virtual entities to the base entity, eg. where a number of subschema definitions are stacked on top of each other. To retrieve the base attribute in this type of situation requires an unwieldy OQL expression, whereas a single `select` statement will suffice using SRQL.

Example 3.8: *retrieve size details for objects*

Before moving physical data from one server to another, it can be useful to determine the size of each object in the result or view collection. This is not a feature of a query language such as OQL, but has been added to the repository query language to aid the performance of transactions which require the movement of data.

Example 3.8:

select datasize local_news.RTE_news

The result of the expression is a set of meta-objects containing the identifier of each object in the collection together with the size of each object in bytes.

4 Implementation

The ODMG 2.0 standard uses the '*d_*' prefix to denote metaclasses and to avoid confusion with standard metaclasses we employed a '*v_*' prefix to denote virtual metaclasses in our extended model. A small portion of the schema repository for ODMG databases was illustrated in *figure 3* with full details available in [Jordan, 1998]. Two design goals were identified before planning our extension to the ODMG metamodel: virtual metaclasses need contain only mapping information to base (or virtual) metaclasses; yet virtual subschemata must contain enough information for high-level graphical tools to browse and display virtual types. This has been well-documented in the past: for example [Subieta, 1996] stating that modern database systems require a richer means of querying data than that offered by simple ASCII-based query editors.

A prototype system was built using Visual C++ 6.0 for the Versant O-O database running on NT platforms. It is assumed that client applications may be either software modules or user interfaces that have a requirement for dynamic queries. In *example 4.1* a query is required to return the names of all classes inside a subschema named *local_news*. The SRQL implementation parses the expression, opens the appropriate database, and generates the result set as a collection of objects of type *Any*,[1] to which the program has access. The objects in each collection can then be 'repackaged' as objects of a specific metaclass depending on the type of query.

[1] Most databases and template software will use an *Any* (or similarly named) class as an abstract class for all possible return types. In this object library, the small number of possible return types keeps the type conversion simple.

Example 4.1 Repository Query

```
database NEWS_ARCHIVE

srql {

MetaName ( select virtual class in
   local_news ) ; };
```

Internally, the program accesses the metadata query layer by creating an instance of type `srql` and passing the query string to the constructor. The same `srql` instance provides access to the result set.

Ideally, all ODMG databases should provide a standard interface for functions such as opening the database and accessing the schema repository. However, this is not the case, and it was necessary to develop an adaptor for the Versant ODMG database for all I/O actions. The problems involved in constructing generic applications are covered in [Roantree and Subieta, 2001]. However, those functions which are non-generic were isolated and placed inside the `srql` class implementation. Thus, to use the metadata software layer with other implementations of ODMG databases, it is necessary to implement only those functions isolated within the `srql` class. Specifically, these functions are opening and closing the database, query execution, the construction of the result set, and the transfer of data from database-specific (eg Versant) objects to C++ objects. The SRQL parser, and the *semantic actions* for each production are all generic, and thus require no modification. Similarly, it was necessary to implement the view mechanism for an ORDB implementation (to provide views of virtual classes connected using inheritance and association relationships) and the subsequent 'repository' interface. Although this has been fully implemented for a Versant ODMG database, it is only partially completed for an Oracle object-relational database.

When developing semantic actions (*C++* program code) for each production, it was possible to take one of two routes: map the SRQL expression to the equivalent OQL expression (shown informally in §3), and then to the vendor implementation of OQL (VQL for the Versant product in our case); or alternatively, use *C++* to retrieve the objects from the database directly. Our initial prototype used the former approach [Roantree et al. (1), 2001] but this lead to problems as not all SRQL (or OQL) queries could be expressed in the vendor implementation of the language. For this reason we adopted the latter solution and bypassed the ODMG OQL and vendor specific OQL query transformations. By doing so we had more control over the performance of queries as we could take advantage of the structure of the schema repository, and construct intermediate collections and indices depending on the type of query expression.

The parser component was developed using ANTLR [ANTLR, 1999] and the semantic actions for each of the query expressions were written in C++. Thus many of the federated services (such as querying and data extraction from local ISs) use the SRQL for metadata retrieval. The system also contains a *view display* module which generates extents for virtual classes and displays them. The implementation of this software was greatly simplified by the SRQL.

5 Conclusions and Future Research

In this paper we described the design and implementation of a metadata query language for ODMG and object-relational databases. Due to the complex nature of the schema repository interface, we defined simple constructs to facilitate the easy expression of metadata queries. It was felt that these extensions provide a far simpler general purpose interface to both the ODMG schema repository, and the extended repository used by our view mechanism. In particular, these extensions can be used by integration engineers who have a requirement to query metadata dynamically, the builders of view (and wrapper) software, and researchers and developers of high-level query and visualization tools for ODMG databases. Furthermore, in multimedia federations, the language can be used to query the size and content of view or result data in circumstances where data is required for transfer between database systems.

Although the ODMG group has not addressed the issue of O-O views, their specification of the metamodel provides a standard interface for metadata storage and retrieval, a necessary starting point for the design of a view mechanism. Our work extended this metamodel to facilitate the storage of view definitions, and provided simple query extensions to retrieve base and virtual metadata. A cleaner approach would have been to re-engineer the schema repository interface completely rather than implement a software layer to negotiate the complexity problem, and this has been suggested by some ODMG commentators. On the other hand, standards are an obvious benefit to systems interoperability. The complex nature of the ODMG metamodel requires a substantial learning curve for programmers and users who require access to metadata: we believe that we have reduced this learning curve with our metadata language extensions. By implementing a prototype view system, we have also shown how this metadata query service can be utilized by other services requiring meta-information.

The view mechanism and original schema repository query language have been implemented for ODMG databases, and are nearing completion for object-relational databases. Future work is focused on two key areas: the development of an XML-based transaction language for multimedia queries; and the inclusion of behaviour in views. The former project will facilitate federated transactions is a controlled federation of multimedia databases, and uses the SRQL to determine the size of view objects. The latter work will permit methods to appear in ODL_V views, and will require a further extension to the SRQL to retrieve behavioural metadata.

References

[ANTLR, 1999] ANTLR Reference Manual. *http://www.antlr.org/doc/* 1999.

[Booch et al., 1999] Booch G., Rumbaugh J., and Jacobson I. *The Unified Modelling Language User Guide.* Addison-Wesley, 1999.

[Cattell, 2000] Cattel R. et. al. (eds.) (2000). *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann.

[Cattell and Barry, 1997] Cattell R. and Barry D. (eds), *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[Jordan, 1998] Jordan D. *C++ Object Databases: Programming with the ODMG Standard.* Addison Wesley, 1998.

[Pitoura et al., 1995] Pitoura E., Bukhres O. and Elmagarmid A. Object Orientation in federated database Systems, *ACM Computing Surveys*, 27:2, pp 141-195, 1995.

[Roantree, 2000] The IOMPAR Project: Secure Transport of Complex Objects. *IOMPAR Technical Report IOM-01*, Dublin City University, (www.compapp.dcu.ie/~iompar), November 2000.

[Roantree, 2001] Roantree M. The ODL$_V$ View Language for ODMG Databases. *IOMPAR Technical Report IOM-02*, Dublin City University, February 2001.

[Roantree et al. (1), 2001] Roantree M., Kennedy J., and Barclay P. Using a Metadata Software Layer in Information Systems Integration. *13th Conference on Advanced Information Systems Engineering (CAiSE 2001)*, pp. 299-314, LNCS 2068, Springer, June 2001.

[Roantree et al. (2), 2001] Roantree M., Kennedy J., and Barclay P. Constructing View Schemata Using an Extended ODL. *9th International IFCIS Conference on Cooperative Information Systems* (CoopIS 2001), pp. 150-162, LNCS 2172, Springer, September 2001.

[Roantree et al., 1999] Roantree M., Murphy J. and Hasselbring W. The OASIS Multidatabase Prototype. *ACM SIGMOD Record*, 28:1, March 1999.

[Roantree and Subieta, 2001] Roantree M. and Subieta K. Generic Applications for Object Oriented Databases. *Submitted for publication* 2001.

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

[Subieta, 1996] Subieta K. Object-Oriented Standards: Can ODMG OQL be extended to a Programming Language? *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pp. 546-555, Japan, 1996.

[Versant, 1999] Versant Corporation. *Versant C++ Reference Manual 5.2*, April 1999.

Appendix I: SRQL BNF

The Schema Repository Query Language (SRQL) is an extended version of OQL which provides special constructs to facilitate metadata queries. The core BNF for OQL is provided in [Cattell and Barry, 1997] (pp 115-119) and we have extended the production rules with 15 *repository* rules, numbered *r1* to *r15*.

Production r1: outline expression

specification :
("database" db:Identifier { definition })+
EOF

Production *r1* (*Pr1*) is simply a container expression for all the queries which are passed to the database. The expression also expects the name of the database.

Production r2: format for query definition

definition :
(srql_dcl TOK_SEMIC)

Production r3: srql expression

srql_dcl :
("srql" {
(sr_query_dcl | name_dcl | count_dcl) TOK_SEMIC)+
}

Pr2 and *Pr3* define the format and content of the types of query expressions.

Production r4: srql expression

sr_query_dcl :
"**select**"
(subschema_dcl |
class_dcl |
attribute_dcl |
relationship_dcl |
link_dcl |
base_dcl)

Pr4 provides a further breakdown of the `sr_query_dcl` declarator: it begins with the `select` keyword, and can take one of six possible expressions. These six expressions are provided in the next six productions.

Production r5: subschema selection expression

subschema_dcl :
("**subschema**" (Identifier)?)

Pr5 uses the `subschema` construct with an optional identifier.

Production r6: class selection expression

class_dcl : (
("**class**" (Identifier)?)
|
("**virtual**" "**class**" (qualifier_dcl)?)
|
(("**virtual**")? "**class**" "**in**" Identifier)
)

Pr6 uses the `class` construct with three possible uses: the first for retrieval of base classes; the second is used to retrieve virtual classes; and the final construct can be used to return all base or virtual classes, depending on the usage of the `virtual` keyword.

Production r7: select attribute expression

attribute_dcl : (
("**attribute**" (qualifier_dcl)?)
|
("**virtual**" "**attribute**" (double_qualifier_dcl)?)
|
(("**virtual**")? "**attribute**" "**in**" qualifier_dcl ("**inherit**")?)
)

Pr7 is similar to *Pr6* but the third form of the expression uses the `inherit` construct in conjunction with the `in` construct. When the `in` keyword is used, then all attributes for the specified class are returned; when both `in` and `inherit` are used together, then attributes from derived classes also form part of the result set.

Production r8: select relationship expression

relationship_dcl : (
("**relationship**" (qualifier_dcl)?)
|
("**virtual**" "**relationship**" (double_qualifier_dcl)?)
|
(("**virtual**")? "**relationship**" "**in**" qualifier_dcl ("**inherit**")?)
)

Production r9: select link expression

link_dcl : (
("**link**" "**class**" qualifier_dcl)
|
("**link**" "**attribute**" double_qualifier_dcl)
|
("**link**" "**relationship**" double_qualifier_dcl)
)

Production r10: select base expression

base_dcl : (
("**base**" "**class**" qualifier_dcl)
|
("**base**" "**attribute**" double_qualifier_dcl)
|
("**base**" "**relationship**" double_qualifier_dcl)
)
)

In *Pr9* and *Pr10* the `link` and `base` constructs are used to return a mapped `d_Class`, `d_Attribute` or `d_Relationship` class instance. Using the `class` construct a *qualifier* is needed (with the name of the virtual subschema and class); and for both `attribute` and `relationship` constructs, a *double qualifier* is needed (with the names of the virtual subschema, class and attribute or relationship name).

Production r11: the MetaName expression

name_dcl :
("**MetaName**" (sr_query_dcl))

Production r12: the MetaCount expression

count_dcl :
("**MetaCount**" (sr_query_dcl))

The `MetaName` and `MetaCount` constructs in *Pr11* and *Pr12* take a select query expression as argument, with the former returning a set of (name) values, and the latter returning a single (integer) value.

Production r13: the type expression

type_dcl : (
("**type**" qualifier_dcl)
|
("**virtual**" "**type**" double_qualifier_dcl))

The `type` construct is used to return the type of a base property (attribute or relationship), and when `virtual` and `type` are used in conjunction, the type of a virtual property is returned.

Production r14: the qualifier expression

qualifier_dcl :
((Identifier .)? Identifier)

Production r15: the double qualifier expression

double_qualifier_dcl :
(((Identifier .)? Identifier .)? Identifier)

A `qualifier_dcl` takes the form *x.y*, and a `double_qualifier_dcl` takes the form *x.y.z*. In the first declarator, the *x* portion is optional, and in the second declarator, valid formats are *z*, *y.z* and *x.y.z*.

Production r16: the datasize expression

datasize_dcl :
("**datasize**" qualifier_dcl)

Production r17: the datacount expression

datasize_dcl :
("**datacount**" qualifier_dcl)

The final two expressions return the size of all objects, and the count of all objects in a named class.