Using a Metadata Software Layer in Information Systems Integration*

Mark Roantree[1], Jessie B. Kennedy[2], and Peter J. Barclay[2]

[1] School of Computer Applications, Dublin City University, Dublin, Ireland
mark.roantree@compapp.dcu.ie
[2] School of Computing, Napier University, Edinburgh, Scotland

Abstract. A Federated Information System requires that multiple (often heterogenous) information systems are integrated to an extent that they can share data. This shared data often takes the form of a federated schema, which is a global view of data taken from distributed sources. One of the issues faced in the engineering of a federated schema is the continuous need to extract metadata from cooperating systems. Where cooperating systems employ an object-oriented common model to interact with each other, this requirement can become a problem due to the type and complexity of metadata queries. In this research, we specified and implemented a metadata software layer in the form of a high-level query interface for the ODMG schema repository, in order to simplify the task of integration system engineers. 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.

1 Introduction

Many database applications require a mechanism by which 'generic' applications can determine a database's structure at runtime, for functions such as graphical browsers, dynamic queries 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 the 2.0 specification of the ODMG metamodel [3] has provided a standard API for metadata queries in object-oriented databases. As part of our research into federated databases, 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 [6,10] for a complete background on federated databases. In this paper we do not concentrate on the topic of federated databases but instead focus on the construction of a metadata interface to ODMG information systems. This paper is structured as follows: the remainder of this section provides a brief overview of the nature of our research, the importance of metadata to our view mechanism, and the motivation for this research; in Sect. 2 the main concepts in

* Supported by Forbairt Strategic Research Programme ST/98/014

K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): CAiSE 2001, LNCS 2068, pp. 299–314, 2001.

this form of research are discussed, together with an informal description of the metadata layer; in Sect. 3 the pragmatics of the language are presented through a series of examples; in Sect. 4 we present details of the implementation; and finally in Sect. 5 we offer some conclusions.

In this paper we use the term *view* (or ODMG view) to refer to an ODMG subschema which may contain multiple classes, and is defined on an ODMG database, or on another view which has been defined on an ODMG database.

1.1 Background and Motivation

The main focus of our research 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 [10] is one where heterogeneous databases (or information systems) can communicate with one another through an interface provided by a common data model. In our case, the common data model is the ODMG model, the standard model for object-oriented databases since 1993 [3]. 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. 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.

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, we 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 we 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, the request to display a view, or extract data from the local IS often requires powerful query facilities in order to retrieve the required meta-information. In Fig. 1 the role of the schema repository within a federated database environment is illustrated. Both the *Component Database* and *Federated Database Server* are ODMG databases. The schema repository contains a description of the database classes, hence the

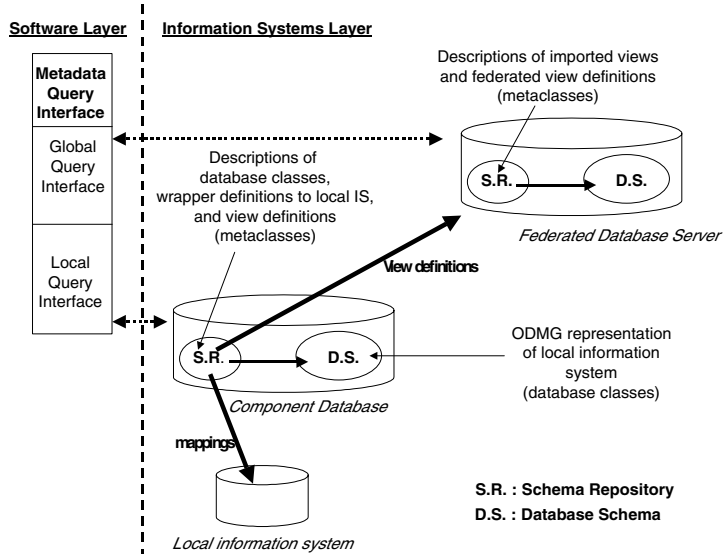

Fig. 1. Metadata architecture within ODMG information systems.

arrow towards the database schema. However, in this type of architecture, the schema repository will contain a large amount of additional data required by view definitions.

This research involved extending the ODMG metamodel in order to construct view schemata. However, due to the complex nature of both the base metamodel and extensions, many of the OQL queries which were required both to retrieve base and view class metadata would necessitate long expressions. For this reason we specified some language extensions to OQL for the specific purpose of easy retrieval of metadata from the schema repository. The contribution of this work is to provide a software metadata layer which facilitates the easier expression of ODMG metadata queries. In fact all metadata queries can be expressed in a single line. In addition, we believe it is possible to improve the performance of metadata queries as some of our experiments (discussed briefly in Sect. 4) have shown.

2 Metadata Objects and Queries

In this section we provide a description of the main concepts involved in this research: 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 defined using a view language and are mapped to base metadata objects. Metadata queries are categorized into groups representing the type of metadata information required.

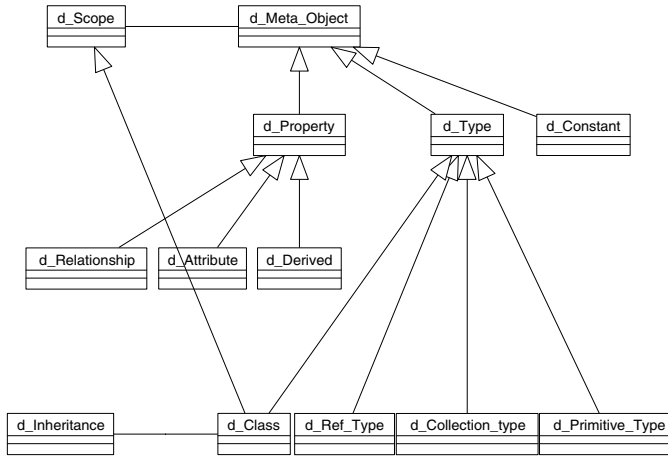

Fig. 2. The ODMG metamodel (subset).

2.1 Metadata Elements

In Fig. 2 a brief outline of the ODMG schema repository interface (or ODMG *C++* metamodel) is illustrated. For integration engineers the main points of interest are classes and their properties, and issues such as scoping and inheritance. The illustration attempts to show a hierarchy of metaclasses with the `d_Scope`, `d_Inheritance` and `d_Meta_Object` metaclasses at the top of the hierarchy. Otherwise, all metaclasses derive from `d_Meta_Object` (identified by arrows) and multiple inheritance occurs where some metaclasses derive from `d_Scope`. By deriving from a `d_Scope` object, meta-objects can place other meta-objects inside a (conceptual) container class. For example, a `d_Class` object derives from `d_Scope`, and can use its properties to bind a set of `d_Attribute` objects to it. Specifics of the roles of each of the metaclasses is best described in [5].

In an ODMG database that contains a view mechanism, it is necessary to be able to distinguish between base and virtual classes, and for virtual classes one must be capable of obtaining the same information regarding structure as can be obtained for base classes. The view mechanism, its specification language, and implementation are described in [8][9]. 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 or wrapper is represented by the meta-objects outlined below.

1. **Subschema or Wrapper 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 `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 every relationship property.

4. **Inheritance Construction.** This type of meta-object `connects classes to subclasses.`
5. **Class Scope Update.** When `v_Attribute` and `v_Relationship` instances are constructed, it is necessary to associate these properties with a specific (virtual) `v_Class` instance. This is done by updating the `v_Scope` object which the `v_Class` object inherits from.
6. **Subschema Scope Update.** 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 has been implemented as a software layer which resides between the client database application and the ODMG database. The language comprises fifteen productions detailed in an appendix in [7]. 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 the 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 as a shallow retrieval (only those attributes for the named class) or a deep retrieval (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 semantically identical 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, nut 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.

3 Pragmatics of SRQL Usage

Although the ODMG model provides a specification for access to the schema repository, it is quite complex and often not easy to formulate OQL metadata queries as we shall later demonstrate. 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).

3.1 Sample Metadata

In our previous work [9] we described how view schemata can be defined using our `subschema` statement. The resulting 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 file (ODL file), passed through the *View Processor*[1], and the result is the storage of the view definition as a set of meta-objects in the database's schema repository. It is these meta-objects which are queried by system integrators as they seek to discover similarities and differences between schemata which are due to be merged, as it is generally view schemata that are merged, rather than the entire base schemata of participating systems. Most of

[1] This is the same process as is used for defining the base schema.

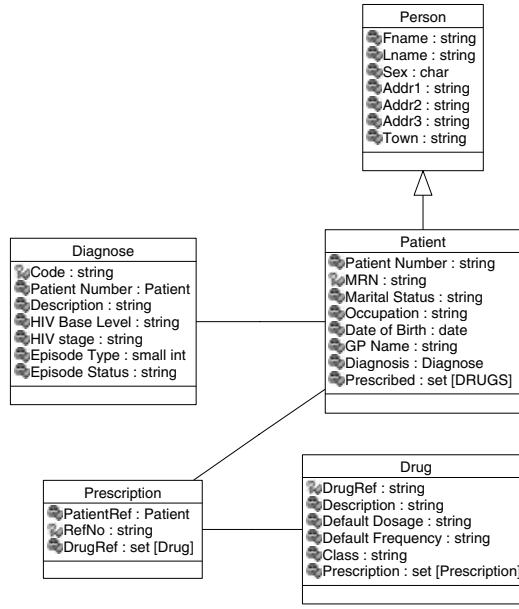

Fig. 3. The v3 view of the PAS database.

our work has involved healthcare systems, and one of these systems is the *Patient Administration System* (PAS) database at a hospital in Dublin. This database contains a wide range of information including a patient's demographic data, treatment of each illness, consultant information, and details of prescribed medication. For the purpose of the examples used later in this paper, assume that the view illustrated in Fig. 3 as patients, details of a particular illness, and the medication prescribed, is stored as *v3* in the database. The types of meta-objects constructed for this view were described in Sect. 2.1.

As it stands, the ODMG metamodel or its C++ API provides an 'open' standard for retrieving ODMG metadata, 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. This work necessitates executing many metadata queries during the course of the schema integration process.

3.2 Metadata Query Samples

In this section, we argue the requirement for the proposed language 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 which appears to treat all derived attributes as local to each specialized class [3] (pp. 84), we have also opted to use the C++ bindings

as provided by the Versant O-O database product [12][2]. Our motivation is 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.

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

This query is expressed by again querying its `d_Meta_Object` superclass, but in this example, a set of references to `d_Class` objects should be generated as output.

Example 3.1(a): ODMG OQL
select C **from** d_Class
where C.name = 'Person'
Example 3.1(b): Vendor OQL
select oid **from** d_Class
where d_Meta_Object::name = 'Person'

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

Example 3.1(c): SRQL
select class Person

The result of this query will be the set of base class (`d_Class`) instances which have the `name` attribute value of *Person*. 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 [9] where each subschema has its own scope and class hierarchy, and thus class names can be repeated across different subschema definitions.

Example 3.2: *Retrieve virtual class object called 'Person' from subschema v3.*

Assume we now wish to retrieve a specific class *Person* from subschema *v3*.

Example 3.2(a): ODMG OQL
select C **from** v_Class
where C.name = 'Person'
and C.SchemaContainer =
(**select** S **from** v_SubSchema **where** S.name = 'v3')
Example 3.2(b): Vendor OQL
s_oid = (**select** oid **from** v_Subschema
where v_Meta_Object::name = 'v3');
select oid **from** v_Class
where v_Meta_Object::name = 'Person'
and SchemaContainer = s_oid;

[2] With one small exception: we use the term *oid* instead of the vendor term *SelfOid*, as we feel that oid is more generic. Before executing any of the queries in this section the term *oid* should be replaced with *SelfOid*.

Example 3.2(c): SRQL
select virtual class v3.Person

In this example, the SRQL makes the OQL query easier as we use a subschema qualifier to specify the correct class. Although the query can be expressed easily in OQL, 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), any implementation is hidden behind the language extensions.

Example 3.3: *Retrieve a virtual attribute 'name' from class 'Person' in subschema v3.*

In this example we again have the problem of first selecting the correct `v_Class` reference and then selecting the appropriate `v_Attribute` object reference. Assume that the virtual class is from the same subschema (*v3*) as the previous example.

Example 3.3(a): ODMG OQL
select A **from** v_Attribute
where A.in_class in
(**select** C **from** v_Class
where C.name = 'Person'
and C.SchemaContainer in
(**select** S **from** v_SubSchema **where** S.name = 'v3'))
Example 3.3(b): Vendor OQL
s_oid = (**select** oid **from** v_Subschema
where v_Meta_Object::name = 'v3');
c_oid = (**select** oid **from** v_Class
where v_Meta_Object::name = 'Person'
and SchemaContainer = s_oid);
select oid **from** v_Attribute
where v_Meta_Object::name = 'name'
and in_class = c_oid);

In Example 3.3(a) the pure OQL version of the query is expressed by simply 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 using some of the OQL implementations when building O-O database software. In Example 3.3(c) the SRQL version of the query is a simple expression.

Example 3.3(c): SRQL
select virtual attribute v3.Person.name

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?* Previous queries assumed that this type of data had already been acquired. In the following examples we will illustrate these types of queries, and will now express queries in OQL and SRQL only as the problem regarding vendor-specific versions of OQL has already been shown.

Example 3.4*: Retrieve all classes within subschema v3*

In Examples 3.4(a) and (b) the syntax for both expressions to retrieve all references to `v_Class` objects within the subschema *v3* is illustrated. As these queries are simpler than those in previous examples, the OQL expressions are fairly straightforward.

Example 3.4(a): ODMG OQL
select C **from** v_Class
where C.SchemaContainer.name = 'v3'
Example 3.4(b): SRQL
select virtual class in v3

The keyword `virtual` is used to distinguish between base and virtual classes, and similar to the `select subschema` expression, 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 *Person* are returned.

Example 3.4(c): class selection formats
(i) **select class**
(ii) **select class** Person
(iii) **select virtual class**
(iv) **select virtual class** Person
(v) **select virtual class** v3.Person

A subschema can comprise both base and virtual classes [9]. 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 v3
select virtual class in v3
Example 3.5*: retrieve all relationships within Person within the v3 schema.*

In this example we require a reference to all relationship objects inside the *Person* class.

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 = 'Person' **and** C.SchemaContainer =
(**select** S **from** v_SubSchema **where** S.name = 'v3')
Example 3.5(b): SRQL
select virtual relationship in v3.Person

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 can be a little unwieldy due to the complexity of the O-O model.

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

Suppose it were necessary to retrieve all attributes for class *e*, which is derived from classes *a*,*b*,*c*, and *d* (in subschema *v3*). (Please refer to [5] for a description on inheritance in the ODMG metamodel.)

Example 3.6(a): ODMG OQL
select A **from** v_Attribute
where A.in_class **in**
(**select** C **from** v_Class
where C.name = 'e'
and C.SchemaContainer in
(**select** S **from** v_SubSchema **where** S.name = 'v3')
union
select A **from** v_Attribute
where A.in_class **in**
(**select** i.inherits_to **from** v_Inheritance
where i.inherits_to.name = 'e'
and i.inherits_to.SchemaContainer in
(**select** S **from** v_SubSchema **where** S.name = 'v3'))

In Example 3.6(a) the OQL query to return the required `v_Attribute` references for the class *e* 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 *e* are retrieved. In the second segment, it is necessary to retrieve all `v_Class` references which are superclasses of class *e*, and perform the same operations on these classes.

Example 3.6(b): SRQL
select virtual attribute in v3.Person **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.

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[3].

Example 3.6(c): attribute selection formats
(i) **select attribute**
(ii) **select attribute** age
(iii) **select attribute** Person.sex
(iv) **select virtual attribute**
(v) **select virtual attribute** age
(vi) **select virtual attribute** Person.sex
(vii) **select virtual attribute** v3.Person.sex

Finally, 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 *v3.Person.name* is mapped to *Person.Fullname* in the base schema. It is necessary to retrieve the mapped attribute name to assist in the query transformation process. Assuming the query expressed in Example 3.3 returns an object reference *R* (the *name* attribute in *Person* class in *v3*), then Example 3.7(a) can be used to retrieve its mapped base attribute.

Example 3.7(a): ODMG OQL (requires result set Q)
select A.VirtualConnector **from** v_Attribute
where A **in** Q
Example 3.7(b): SRQL (full query expression)
select link attribute v3.Person.name

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 (identical syntax to Example 3.7(b)). 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

[3] or possibly *NULL*.

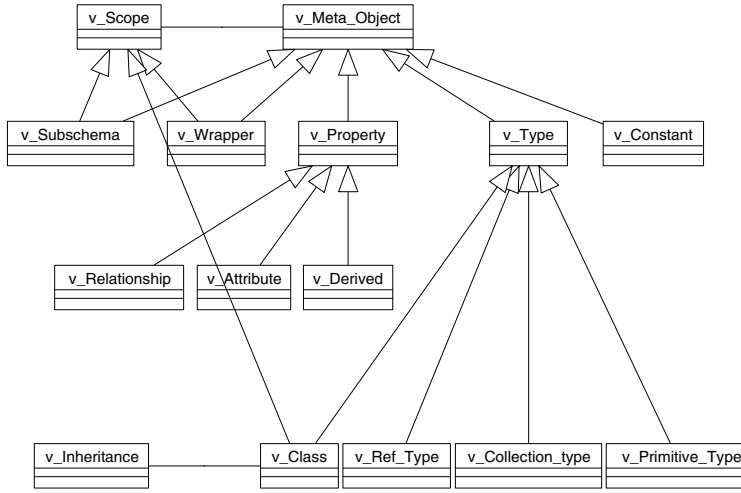

Fig. 4. ODMG metamodel extensions (subset)

OQL expression, whereas a single `select` statement will suffice if we provide the appropriate extensions to OQL.

In [7] we provide the production rules for the schema repository query language. The same publication contains the syntax for further queries such as the retrieval of subschema and wrapper objects, the retrieval of attribute and relationship type data, and the retrieval of object names and object counts.

4 Implementation of Metadata Interface Layer

The extensions to the C++ metamodel make provision for virtual schemata, any component virtual classes, and their properties and types. The ODMG 2.0 standard uses the '*d_*' prefix to denote metaclasses and to avoid confusion with standard metaclasses we employ a '*v_*' prefix to denote virtual metaclasses. The schema repository for ODMG databases was illustrated in Fig. 2. 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 [11] stating that modern database systems require a richer means of querying data than that offered by simple ASCII-based query editors.

In Fig. 4 the segment of metamodel extensions which contains the most useful metaclasses is illustrated. For the purpose of clarity, the base metaclasses are not shown but the top layer in the hierarchy (containing `v_Scope`, `v_Inheritance` and `v_Meta_Object` metaclasses) is placed at the same level in the metaclass hierarchy as their base metaclass equivalents. We deliberately chose to extend the

metamodel by adding virtual classes rather than extending existing metaclasses, in order to keep our base metamodel compatible with the ODMG standard. In Fig. 4 a list of the metaclass types added to the set specified by the ODMG is illustrated. Please refer to [9] for a description of virtual metaclasses and their functions.

4.1 Metadata Software Layer

An object library 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 *v3*. The object library parses the expression, opens the appropriate database, and generates the result set as a collection of objects of type *Any*[4], 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.

Example 4.1 Repository Query

```
database PAS
srql {
MetaName ( select virtual class in v3 ) ; };
```

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. A suitable adaptor will be necessary for all ODMG implementations. However, we have isolated those functions which are non-generic and placed them 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.

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 Sect. 3), and then to the vendor implementation of OQL (VQL for the Versant product in our case); or

[4] 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.

alternatively, use *C++* to retrieve the objects from the database directly. Our initial prototype used the former approach 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. Space restrictions prevent a further discussion on this subject here but can found in a forthcoming technical report.

The parser component was developed using ANTLR [1] 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. All of the SRQL examples in this paper were tested using our prototype: the Versant O-O database was used to test vendor OQL queries; and a research O-O database [4] (the closest implementation of OQL to the ODMG specification) was used to test each of the native OQL queries. All of this work is based on the ODMG 2.0 specification [3].

5 Conclusions

In this paper we described the construction of a metadata software layer for ODMG 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.

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. Perhaps this could be regarded as a weakness in our approach, but we chose to retain the standard (now at version 3.0), and attempt to make its metadata interface more usable. Standards are an obvious benefit to systems interoperability, but quite often that standard can be improved.

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.

References

1. ANTLR Reference Manual. *http://www.antlr.org/doc/* 1999.
2. Booch G., Rumbaugh J., and Jacobson I. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.
3. Cattell R. and Barry D. (eds), *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.
4. Fegaras L., Srinivasan C., Rajendran A., and Maier D. $\lambda - DB$: An ODMG-Based Object-Oriented DBMS. *Proceedings of the 2000 ACM SIGMOD*, May 2000.
5. Jordan D. *C++ Object Databases: Programming with the ODMG Standard*. Addison Wesley, 1998.
6. Pitoura E., Bukhres O. and Elmagarmid A. Object Orientation in federated database Systems, *ACM Computing Surveys*, 27:2, pp 141-195, 1995.
7. Roantree M. A Schema Repository Query Language for ODMG Databases. *OASIS Technical Report OAS-09*, Dublin City University, (www.compapp.dcu.ie/~oasis), July 2000.
8. Roantree M. Constructing View Schemata Using an Extended Object Definition Language. *PhD Thesis*. Napier University, March 2001.
9. Roantree M., Kennedy J., and Barclay P. Defining Federated Views for ODMG Databases. *Submitted for publication*, July 2000.
10. 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.
11. 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.
12. Versant Corporation. *Versant C++ Reference Manual 5.2*, April 1999.