# Extending the Object Query Language for Transparent Metadata Access

Hong Su, Kajal T. Claypool, and Elke A. Rundensteiner

Worcester Polytechnic Institute, 100 Intitute Road,
Worcester, MA, USA
{suhong, kajal, rundenst}@cs.wpi.edu
http://davis.wpi.edu/dsrg

**Abstract.** Many applications in Object Databases (ODB), for example, schema management tools, CASE tools, database development tools and integration wrappers, need extensive queries over both application data as well as metadata. However queries over metadata via OQL, a de-facto standard for object query languages defined for the ODMG 2.0 Object Model, are tied to low-level implementation details of the underlying schema repository of the database system. Hence, they are neither portable nor easily usable, requiring the application developer to have detailed knowledge of the proprietary structure of the schema repository. In this paper, we propose an extension of OQL, called MetaOQL, to address this limitation. Our proposition of MetaOQL offers several benefits: (1) it is a natural extension of OQL in terms of both its syntax and semantics; (2) it removes the dependency of metadata queries on the particular schema repository, hence providing uniformity and portability of metadata queries across different ODBs; (3) it supports transparent navigation over the metadata thus offering ease of use; (4) unlike OQL, it hides metadata querying details from the users hence the queries can be simplified and more easy to read and understand. We have also investigated implementation strategies for MetaOQL. In particular, we propose a translation strategy from MetaOQL to OQL as a preferable solution compared to development of a special-purpose MetaOQL processor. The translation strategy offers the advantage that the MetaOQL queries can be retargeted to work on top of any existing ODB engine equipped with OQL with minimal effort. Furthermore, all OQL query optimization strategies can thus still be brought to bear in our extended system.

## 1 Introduction

*Motivation: Metadata Access* Analogous to system tables in relational database systems, metadata in object database systems (ODB) provides the descriptive information about the database objects that defines the schema of a database. A large number of applications on ODBs, especially schema management tools, CASE tools, schema evolution tools and integration wrappers, need to have access to the metadata. For example, class and object browsers [Obj99a] often need to access metadata in order to display pertinent information such as the class definitions, the inheritance hierarchy or the association between classes. The schema evolution tools [CJR98a] dealing with schema restructuring such as adding new attributes to a class, adding new classes into a class hierarchy, involve extensive metadata querying and manipulation.

Most ODB systems indeed do allow access to the metadata [Tec94,Inc93]. As per ODMG, the access to metadata is often provided via a high-level declarative interface such as a query language instead of just a procedural low-level application programming interface (API) [Obj93]. An object query language, by virtue of treating the schema information as objects, is in principal capable of powerful metadata querying and manipulation. OQL [Cea97] is such a powerful standard object query language based on ODMG 2.0 Object Model that combines high-level declarative programming features with the object-oriented paradigm.

*Problems of Metadata Access via OQL* As per the ODMG 2.0 standard [Cea97], metadata is stored in an Object Definition Language (ODL) schema repository, which is accessible to tools and applications using the same operations that apply to user-defined types. However, the ODMG standard only defines the interface methods through which meta-information is manipulated, rather than defining the entire class structure and details of the internal implementation of the schema repository. Thus, today while most commercial vendors [Obj99b,Gem99] attempt to provide ODMG compliant ODB systems, the actual physical representation of the schema repository varies from vendor to vendor. Hence while OQL provides declarative access to the schema repository, the OQL queries on metadata are tied to the vendor-specific schema repository. This has several disadvantages. First, the metadata query is not portable due to its tight coupling to the vendor-specific schema repository. Second, a user cannot query the metadata without complete knowledge of the internals of the schema repository of the given ODB system. Thus, tools such as class or object browsers today need to be implemented and deployed for a particular database. Portability of these tools to other ODB systems requires extensive re-engineering of the tools, or more development from scratch.

These disadvantages are also exemplified in the SERF system (Schema evolution through a Extensible, Re-usable and Flexible framework) [CJR98b]. Schema evolution is a fundamental aspect of information and database systems [Rea00]. SERF is the first system that enables users to define new complex schema evolution transformations in a flexible and extensible manner. Most systems support schema evloution by providing a set of schema evolution primitives. However, we found most schema evolution can be broken down to a sequence of minimal set of *basic schema evolution primitives*. Hence SERF uses OQL to arbitrarily combine these basic schema evolution primitives, application object updates as well as metadata access to describe the desired transformations. OQL queries over the schema repository to gather system schema information for use in the transformations. However relying on OQL alone for the metadata access often detracts from the portability of the transformations.

*Example Illustrating the Problem* This problem is illustrated in the example below. Figure 1 depicts a complex transformation *inline* defined as the replacement of a referenced type with its type definition. In this case, the class *Person* has an attribute *address* of the complex type *Address*. In the transformation, the *Address* type is inlined into the *Person* class, i.e., all the attributes defined in the *Address* type (the referenced type) are now added to the *Person* type resulting in a more complex *Person* class. Figure 2 is an SERF representation of the *inline* transformation using OQL. In the transformation, it calls two basic schema evolution primitives provided by the underlying ODB system, *add_attribute*[1] and *delete_attribute*[2].

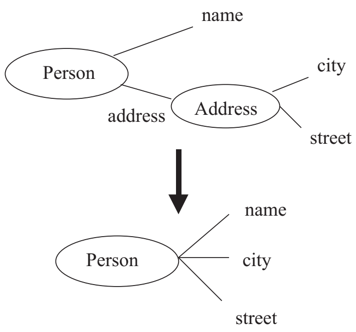


**Fig. 1.** Inline Transformation

The current SERF prototype is built using Object Design Inc.'s Persistent Storage Engine Pro 2.0 (PSE Pro) [O'B97] as the underlying ODB system. In the OQL statements in Figure 2, all words in bold font are tied to the implementation details of the structure of the underlying schema repository. Thus the codes of the transformations built on the proprietary schema repository of PSE Pro are not portable across different ODB systems due to the schema discrepancy. Suppose the application has another version built on another ODB and we also want to port this transformation built on PSE Pro ODB to the targeted ODB, we would have to

[1] *add_attribute(String className, String attrName, String attrType)* will add a new attribute *attrName* of type *attrType* into the class *className*. If successful, it returns true, otherwise false.

[2] *delete_attribute(String className, String attrName)* will delete an attribute *attrName* from the class *className*. If successful, it returns true, otherwise false.

```
// Retrieve local attributes of class cName
define localAttrs(cName) as
  element(select c.localAttrList
          from MetaClass c
          where c.className = cName);

// Get type name of attribute aName defined in class cName
define refClass(cName, aName) as
  element(select attr.attrType.typeName
          from localAttrs(cName) attr
          where attr.attrName = aName);

//call schema evolution primitive add_attribute to add the
//attributes defined in address's type to class Person
for all attr in localAttrs(refClass(Person,address)) :
  add_attribute(Person, attr.attrName, attr.attrType);

//call schema evolution primitive delete_attribute to delete
//attribute address from class Person
delete_attribute (Person,address);
```

**Fig. 2.** Inline Transformation Written in OQL

modify all the system-dependent parts of the metadata queries to adjust to the specific physical representation of the targeted ODB. For example, first, the name of the class whose instances describe application class definition information (in our example, *MetaClass*) may be different in another ODB system (for instance, in ObjectStore [Obj93], *os_class_type*). Second, besides the name discrepancy which can be solved by simple string replacement, the attribute *localAttrList* of the class *MetaClass* which describes the local attribute definitions may not be a public attribute in another ODB. In this case, we may need to identify and then make use of the method to retrieve the desired *localAttrList* information.

Another limitation arises when the metadata query is tightly coupled to the implementation details of the schema repository. While the developers of an application will have complete knowledge of the application schema since they designed it themselves, they may have little knowledge of the schema repository's specific structure. This would require them to resort to the system documentation of the ODB system to gain complete knowledge of the structure of the schema repository. Furthermore, metadata queries involving the retrieval details not only lead to bloated codes in many cases, but also are not too easy to understand, impacting the codes' readability and thus software maintainability which are very important criteria in the software industry.

*Our Proposition* Hence there is a need for a general metadata access language to overcome these shortcomings. The language should be independent from the actual physical storage structure of the schema repository. We here propose a new language as an extension of OQL, called MetaOQL, that allows transparent system-independent navigation over the schema repository and hence provides uniform metadata access.

The advantages of MetaOQL are:

– MetaOQL is a natural extension to OQL, the de-facto standard of object query languages, in the sense that it is compatible with OQL

syntax and semantics, hence it is easy to use for the users who are familiar with OQL;
- MetaOQL provides generic metadata queries portable across different ODMG compliant ODB systems;
- MetaOQL supports transparent navigation over the metadata thus providing better usability to users in that the users need not be aware of low-level details of the schema repository in order to get access to it;
- By hiding the internal metadata retrieval details from the users, metadata query statements can be drastically simplified and thus are more easy to understand, providing better readability to users;
- MetaOQL can be implemented in a non-intrusive manner on any current ODB system that supports OQL without imposing restrictions on ODBs.

To summarize, the contributions of this work include:
- We identify the limitation of the current object query language OQL with respect to metadata queries in terms of portability and usability;
- We propose a language that solves the problem of limited metadata query portability across different ODB systems while at the same time improving its usability;
- We demonstrate the advantages of MetaOQL via extensive examples;
- We propose a one-pass translation approach from MetaOQL into OQL which is non-intrusive and highly efficient to implement MetaOQL on existing OQL engines.

The following outlines the remainder of the paper. Section 2 presents the syntax of MetaOQL while Section 3 illustrates the features of MetaOQL via examples. Section 4 describes the implementation strategy of MetaOQL. Section 5 discusses related research. We conclude with a summary and a discussion of future work in Section 6.

## 2 MetaOQL Syntax Extension to OQL

### 2.1 OQL Reviewed

OQL is an object query language proposed for the ODMG Object Model [Cea97]. It is similar to SQL92 with object-oriented extensions like complex objects, object identity, path expressions, operation invocation etc. And OQL is an expression language. A query expression is built from typed operands composed recursively by operators. Our MetaOQL extension to OQL introduces new syntax for representing special set expressions. So below we will quickly review the related parts of the OQL syntax while full details can be found elsewhere [Cea97].

OQL provides high-level primitives to deal with the *collection* construct. Collection objects are composed of distinct elements of the same type. *Set* type is one of the collection types supported by the ODMG Object Model. A set object is an unordered collection of elements, with no duplicates allowed. Set construct supports some operations that other collection types do not support.

The *collection expressions* include *universal quantification*, *existential quantification*, *membership testing* and *aggregate operators.* The *binary set expressions* include *union*, *intersection*, *difference* and *inclusion.* Typical syntax related to collection and set expressions as used later in our examples is listed below:

1. Universal quantification: *for all* $x$ *in* $e_1$*:*$p$
   If $x$ is a variable name, $e_1$ denotes a *collection* and $p$ is an predicate expression of type *boolean*, then "*for all* $x$ *in* $e_1$*:* $p$" returns true if all the elements of collection $e_1$ satisfy $p$ and false otherwise.
2. Member testing: $x$ in $e_1$
   If $e_1$ denotes a *collection*, $x$ is an object or literal having the same type or a subtype as the elements of $e_1$, then "$x$ in $e_1$" returns true if element $x$ belongs to collection $e_1$ and false otherwise.
3. Binary set expression: If $e_1$ and $e_2$ are *sets*, <op> is an operator from *union*, *except*, *intersect*, $e_1$ <op> $e_2$ computes set theoretic operations, union, difference, and intersection on $e_1$ and $e_2$.
4. Select from where clause: select f from $x_1$ in $e_1$, $x_2$ in $e_2$, ..., $x_n$ in $e_n$ [where p]
   The $e_i$'s have to be of type *collection*, p has to be of type *boolean.* $x_i$ is a variable that ranges over the collection $e_i$. The result of the query will be a collection of $t$, where $t$ is the type of the result of function $f$.

### 2.2 Syntax of MetaOQL

MetaOQL extends OQL in that it introduces new set expressions for representing and manipulating metadata information. The definitions of these *MetaOQL-specific set expressions* are given below. We get some inspiration for some of the expressions, i.e. expression 1 (a), 1 (b) from ShemaOQL [LSS96]. Figure 3 shows an example of the ODL definition for a schema that will be used as our running example.

```
class Person
{
  attribute string name;
  attribute Address address;
}

class Graduate: Student
{
  attribute short graduateYear;
  attribute string street;
}
```

```
Class Student: Person
{
  attribute short id;
}

class Address{
  attribute string city;
  attribute string street;
}
```

**Fig. 3.** ODL Definition of a Database Schema

**Definition 1 MetaOQL-Specific Set Expression**

A *MetaOQL-specific set expression* is one of the following expressions, where *cl* is a *class expression* defined in Definition 2.

(a) ->

-> denotes a set of names of all classes that are defined in a given scope. Currently, many ODB systems [Obj93] only support one single schema, i.e., all classes defined in a database are within the same schema scope (name space).

(b) *cl*->

*cl*-> denotes a set of *full path attribute expressions* of the local attributes defined in the class *cl*. The type of *full path attribute expression* is a new type we introduce into the MetaOQL type system.

(c) *cl*->*

*cl*->* denotes a set of *full path attribute expressions* of the attributes, local and inherited, of a given class *cl*.

(d) *cl*+

*cl*+ denotes a set of names of the immediate superclasses of a given class *cl*. For an object model that does not support multiple-inheritance, it returns a singleton set composed of one single element.

(e) *cl*++

*cl*++ denotes a set of names of all the superclasses of a given class *cl*.

(f) *cl*-

*cl*- denotes a set of names of the immediate subclasses of a given class *cl*.

(g) *cl*- -

*cl*- - denotes a set of names of all the subclasses of a given class *cl*.

*Example 1.* In the schema defined in Figure 3, we have:
-> denotes a set of class names: {“Person”, “Student”, “Graduate”, “Address”}.
Person-> denotes a set of full path attribute expressions of all local attributes defined in class Person: {Person.name, Person.address}.
Student->* denotes the set of full path attribute expressions of all local and inherited attributes defined in class Person: {Student.name, Student.address, Student.id}.
Graduate+ denotes the set of all immediate superclass names: {“Student”}
Graduate++ denotes the set of all superclass names: {“Student”, “Person”}.
Person- denotes the set of the immediate subclass names: {“Student”}.
Person- - denotes the set of all subclass names: {“Student”, “Graduate”}.

**Definition 2 Class Expression**

A *class expression cl* is defined by one of the following forms:

- a constant, i.e., a fixed class name.
- an expression of applying a *get-type operator* (defined in Definition 4) to an *attribute expression* (defined in Definition 3).

**Definition 3 Attribute Expression**

A *attribute expression* is defined by one of the following forms:

- *a*, where *a* denotes a variable that ranges over the metadata set defined in Definition 1 (b) or 1 (c), i.e., *cl*-> or *cl*->* for some *cl*.

– $C.A_1.A_2...A_n$, where $C$ denotes a fixed class name, $A_1$, $A_2$,..., $A_n$ denote fixed attribute names. The expression is a concatenation of a class name and a sequence of attribute names.

**Definition 4 Get-Type Operator**

@$e_1$, where @ is referred as *get-type* operator, $e_1$ is an *attribute expression*, returns attribute $e_1$'s type name.

*Example 2.* @(Person.address) = "Address"

**Definition 5 Get-Name Operator**

$e_1$, where | | is referred as *get-name* operator, $e_1$ is an *attribute expression*, returns the attribute $e_1$'s name.

*Example 3.* If *attr* is a variable that ranges over Person-> and the current value of *attr* over the iteration is *Person.address*, then $|attr|$ = "address".

OQL is a purely functional language that allows its operators to be freely composed, as long as the operands respect the type system. This is a consequence of the fact that the result of any query has a type that belongs to the ODMG type model and thus can be queried again. As a natural extension to OQL, the new set expressions introduced by MetaOQL are also typed. Besides the set expressions *cl*-> and *cl*->* defined in Definition 1 (b) and 1 (c), all the other set expressions are of type *set*<*String*>. *Class Expression* is of type *String* with a constraint that the string must be a class name defined in the schema. For the expressions *cl*->, *cl*->* and *attribute expression*, we introduce a new type *full path attribute expression type*. A expression of this type is different from the concept of a *path expression* in OQL. A *path expression* is a way to navigate from a complex object to other object instances using object references to reach the desired data. It models a navigation over object instances. A *full path attribute expression* is a navigation over the class hierarchy to reach a desired attribute. For example, we have a class `Person` with the ODL definition shown in Figure 3 and an instance `p` of class `Person`, `p.name` is a *path expression* while `Person.name` is a *full path attribute expression*. Hence *cl*-> and *cl*->* are of type *set*<*full path attribute expression type*>. Example 5 shows the free composition of operators.

*Example 4.* (@(Person.address))-> denotes a set: {Address.city, Address.street}
**Explanation:** @(Person.address) will return the type name of attribute *address* defined in class *Person* which is *Address*. Then (@(Person.address))-> will evaluate as Address-> which will return the full path attribute expressions of all attributes defined in type Address, i.e., Address.city and Address.street.

## 3 Portability and Usability Improved by MetaOQL

In this section, we illustrate some of the benefits of using MetaOQL in the place of writing OQL queries against the metadata repository of the respective ODB systems. We base our case study on the SERF system

[CJR98a,CR99b,CR99a]. In its first cut, the SERF framework allows the users to describe complex schema transformations using OQL. These transformations are principally as general and hence applicable to any tool that is based on OQL requiring metadata access. However, ad-hoc transformations suffer from the fact that they specify the transformation for one particular application schema only. To improve the re-usability of the transformations hence avoiding to rewrite each such transformation from the scratch, the notion of SERF templates [CJR98b,RCL+99], has been introduced. A template, a generalized named transformation that includes input and output parameters, can be instantiated and then applied to different systems and different application schemas based on the provided input parameters. Besides the significant advantages of the re-usability brought by SERF templates, the aim for these templates is also to make them portable over all ODMG compliant ODB systems in the form of libraries. However, the encapsulation and generalization of the transformation templates requires extensive metadata access based on the input parameters such as the given class and attribute names. While depending on OQL alone, the transformation templates are tightly coupled to a specific schema repository of an ODB system. Hence for SERF, MetaOQL brings forth the major advantage of portability. SERF templates becomes portable and reusable across different ODMG compliant ODBs as libraries.

Figure 4 illustrates an instantiated *inline* template to perform the same transformation shown in Figure 2. The template *Inline(String className, String refAttrName)* is a generalized transformation that inlines the type of an attribute specified by the parameter *refAttrName* into the class specified by the parameter *className*. In the example shown in Figure 4, the parameter *className* is instantiated to "*Person*" and parameter *refAttrName* to "*address*".

```
Inline("Person","address")
{
  for all attr in (@(Person.address))-> :
    add_attribute ("Person", |attr|, @attr);

  delete_attribute ("Person", "address");
}
```

**Fig. 4.** Inline Transformation from Figure 2 Written in MetaOQL

Figure 5 depicts two other complex schema evolution transformations called *merge_difference* and *merge_union*. In the *merge_difference* transformation, the structure of the new class *JournalPaper* is defined by the difference of the attributes of the two source classes *Author* and *Pa-*

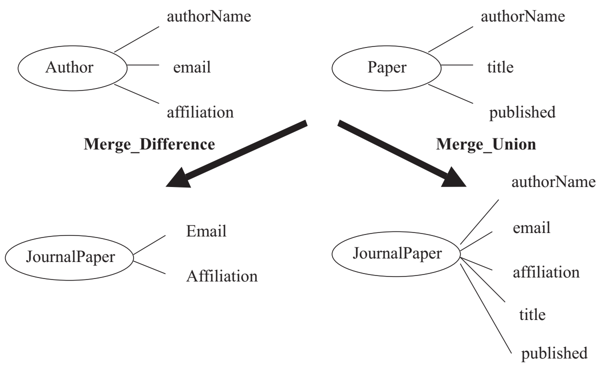


**Fig. 5.** Merge_Difference and Merge_Union Transformation

*per* while in *merge_union* transformation, *JournalPaper* is defined as the union of the attributes of *Author* and *Paper*.

Figures 6 and 7 show the instantiated *merge_difference* transformation template [3] written in OQL and MetaOQL respectively. Figures 8 and 9 show the instantiated *merge_union* transformation template [4] written in OQL and MetaOQL respectively. We can see that in the transformations written in OQL, named queries over the metadata are tightly coupled to the internal structure of schema repository and hence not so easy for the programmers to write and for the readers to understand. However in the transformation written in MetaOQL, all the named queries over metadata in OQL are now replaced by simple and clear MetaOQL-specific set expressions. From the comparison of the same transformation written in OQL and MetaOQL, it is obvious that the transformation can be drastically simplified if written in MetaOQL while at the same time being elegant and easy to read. And the uniform access to metadata also makes the transformation portable across different ODB systems.

## 4 The MetaOQL Translator

In the view of the popularity of the de-facto standard object query language OQL, we aim at an efficient yet non-intrusive implementation of MetaOQL. Our approach is thus to add a translator above the existing

[3] *merge_difference(String className1, String className2, String newClassName)* defines the transformation of merging difference attributes in class *className1* and class *className2* into new class *newClassName*.

[4] *merge_union(String className1, String className2, String newClassName)* defines the transformation of merging union attributes in class *className1* and class *className2* into new class *newClassName*.

```
Merge_Difference("Author","Paper","JournalPaper")
{
 //Retrieve local attributes of class cName
 define localAttrs(cName) as
   element(select c.localAttrList
           from MetaClass c
           where c.className = cName);

//Retrieve the difference attributes of class1 an class2
 define diffAttrs(class1,class2) as
   select attr1
   from localAttrs(class1) attr
   where not
     (attr1.attrName in
            select attr2.attrName
            from localAttrs(class2) attr2);

 //call schema evolution primitive to create a class JournalPaper
 create_class ("JournalPaper");

 //call schema evolution primitive to add difference attributes
 //of class Author and Paper into new class JournalPaper
 for all attr in unionAttrs("Author","Paper") :
   add_attribute(JournalPaper,attr.attrName,attr.attrType);
}
```

**Fig. 6.** Merge_Difference Transformation Written in OQL

OQL engine instead of developing a new query processor. The translator will do a one-pass translation from MetaOQL statements to OQL statements. The later can then be processed by any of the existing OQL engines, exploiting their existing techniques for query optimization. The translator, which is a middleware, can be developed either by the ODB vendors or some other parties as long as the vendors provide the corresponding APIs we'll address later. In this way, we alleviate the additional requirement that are imposed on the ODB vendors. An additional advantage of adopting this translator approach is that not only the application written in the MetaOQL language but also the MetaOQL processor itself is portable to any OODB system.

### 4.1 Overview of the ODMG Schema Access Class Hierarchy

In the following, we expect the underlying ODB schema repository to be ODMG compliant. We briefly give an overview of the ODMG schema access class hierarchy as described in [Cea97].

– *d_Scope*

  The *d_Scope* instances are used to form a hierarchy of meta-objects. A *d_Scope* instance contains a list of *d_Meta_Object* instances representing the elements defined in the schema. Operations to manage the list, e.g., finding a *d_Meta_Object* by its name, are provided.

```
Merge_Difference("Author","Paper","JournalPaper")
{
  //Retrieve the difference attributes of class1 and class2
  define diffAttrs(class1,class2) as
     select attr1
     from class1-> attr1
     where not
       (|attr1| in select |attr2| from class2-> attr2);

  create_class ("JournalPaper");

  for all attr in diffAttrs("Author","Paper") :
     add_attribute ("JournalPaper",|attr|,@attr);
}
```

**Fig. 7.** Merge_Difference Transformation Written in MetaOQL

- *d_Meta_Object*

  Instances of *d_Meta_Object* are used to describe elements of the schema. In particular, we have:
  - *d_Type*

    *d_Type* is an abstract class for all type descriptions.
    * *d_Class*
      A *d_Class* instance is used to describe an application-defined class. All persistent-capable classes are described by a *d_Class* instance.
    * *d_Primitive_Type*
      A *d_Primitive_Type* represents all built-in types, e.g., short, float, boolean etc.
  - *d_Attribute*
    A *d_Attribute* instance describes an attribute of an object or structure.

### 4.2 System Requirements

As we mentioned before, the responsibility of metadata retrieval is shifted from the users to the translator. The new expressions introduced in MetaOQL for referring to meta objects and their properties are effective shortcuts to retrieving metadata information. Users directly use these MetaOQL-specific expressions to represent sets of desired metadata without having to be concerned with how to retrieve such information from the system dictionary. The details of retrieval are left to the MetaOQL processor.

Since the translator itself is tied to the low-level implementation details of the underlying schema repository, one may expect that we have to develop a specific translator for each ODMG compliant ODB. This is obviously not an attractive idea. Here instead we put forth that we can develop a generic translator tool that can translate the MetaOQL

```
Merge_Union("Author","Paper","JournalPaper")
{
 //Retrieve local attributes of class cName
 define localAttrs(cName) as
   element(select c.localAttrList
           from MetaClass c
           where c.className = cName);

//Retrieve the difference attributes of class1 an class2
 define unionAttrs(class1,class2) as
   localAttrs(class1) union localAttrs(class2);

 //call schema evolution primitive to create a class JournalPaper
 create_class ("JournalPaper");

 //call schema evolution primitive to add union attributes
 //of class Author and Paper into new class JournalPaper
 for all attr in unionAttrs("Author","Paper") :
   add_attribute("JournalPaper",attr.attrName,attr.attrType);
}
```

**Fig. 8.** Merge_Union Transformation Written in OQL

```
Merge_Union("Author","Paper","JournalPaper")
{
  //Retrieve union attributes of class1 and class2
  define unionAttrs(class1,class2) as
   class1-> union class2->

  create_class ("JournalPaper");

  for all attr in unionAttrs("Author","Paper") :
   add_attribute ("JournalPaper",|attr|,@attr);
}
```

**Fig. 9.** Merge_Union Transformation Written in MetaOQL

metadata query statements into OQL statements specific to any schema repository. However for the uniform translator approach, we impose some system requirements on the ODMG compliant ODBs. There is a tradeoff between the uniformity of the translators built above different ODBs and the additional system requirements on the ODBs.

The ODMG 2.0 standard defines interfaces for accessing the schema of an ODMG database. The interfaces define an iterator protocol which supports several methods including retrieving the current element of the iterator, advancing the iterator to the next element, and so on. The class *MetaScope* defines an iterator that iterates over all instances of *MetaObject* defined in the schema. The class *MetaClass* defines an iterator that iterates over instances of *MetaAttribute* representing all local attributes defined in the class. The *MetaClass* also defines iterators that can iterate over the instances of *MetaClass* representing the subclasses and superclasses of the class itself represents.

Although we can retrieve all the meta-information via iterators, we instead expect the underlying ODBs to provide a set of schema access APIs that will make the translation process uniform. These schema access APIs can be implemented via the methods that should be provided by all ODMG compliant ODB schema repositories to accord with the interface specifications. So both the ODB vendors and any other programmers can implement these APIs easily as long as the underlying schema repository is ODMG compliant. Thus the uniform translator approach requires wrappers to be built by each ODB system. Hence for each ODB, there should be one such mapping table in which each MetaOQL-specific expression has an entry indicating the corresponding ODB API available to realize the desired functionality. Table 1 shows the mapping table we have built on the PSE Pro's ODMG compliant schema repository, the system that we are targeting for implementing the MetaOQL processor. In addition, we also require system APIs that can retrieve certain common properties of the metadata objects. These methods are listed in Table 2.

| **Expression** | **System API** |
|---|---|
| -> | MetaScope.getClasses |
| cl-> | MetaClass.getLocalAttrs |
| cl->* | MetaClass.getAllAttrs |
| cl+ | MetaClass.getSupers |
| cl++ | MetaClass.getAllSupers |
| cl- | MetaClass.getSubs |
| cl - - | MetaClass.getAllSubs |

**Table 1.** Schema Access APIs for PSE Pro

| **Desired Property** | **System API** |
|---|---|
| className | MetaClass.getName |
| primitiveTypeName | MetaPrimitiveType.getName |
| attributeName | MetaAttribute.getAttrName |
| attributeType | MetaAttribute.getAttrType |

**Table 2.** Methods to Retrieve Properties of Meta-Object

For each metadata set expression, our MetaOQL wrapper for the PSE Pro schema repository provides a system API to retrieve the desired metadata. For example, the class *MetaScope* provides a method *getClasses* to retrieve the names of all classes defined in the schema scope. Another ODB may not provide APIs of exactly the same format as PSE Pro. Wrappers for ObjectStore [Obj93] for example provides a system API *os_database_schema.get_Classes* to realize the same functionality. Thus the implementation of our translator includes a graphic user interface (GUI) for the mapping table definition. The translator will translate MetaOQL metadata queries to OQL queries on the given proprietary schema repository using the APIs provided by the MetaOQL wrapper of the ODB system. And then the OQL statements will be processed by the existing OQL engine to query and manipulate the metadata. The procedure of processing MetaOQL by the generic MetaOQL processor is shown in Figure 10.

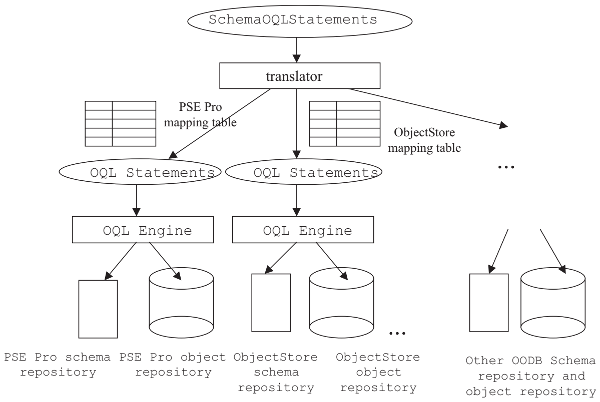


**Fig. 10.** One-Pass General MetaOQL Translator Approach

### 4.3 Translation Strategy

The translation process consists of two steps:
Step 1: Generate all named queries according to the mapping table and put all these system-specific named queries at the head of OQL statements;
Step 2: Scan the MetaOQL statements, replacing all MetaOQL-specific set expressions and operators with their corresponding defined named queries as further explained below.
It can easily be seen that our translation from MetaOQL to OQL is a one-pass process without any intermediate execution. In the following, we will discuss the generating the named queries basing on the mapping tables of Table 1 and 2 provided by the wrappers for PSE Pro schema repository.

**Translation of MetaOQL-Specific Set Expression** For each MetaOQL-specific set expression, we define a named query. We use each name query to replace a MetaOQL-specific set expression during translation process. In OQL, a named query "*define [query] id*($x_1$,$x_2$,...,$x_n$) *as e*($x_1$,$x_2$,...,$x_n$)" records the definition of the function with name *id* in the database schema. Once the definition has been made, each time when the OQL query engine evaluates a query and encounters such a function expression, if it cannot be directly evaluated or bound to a function or method, the OQL query engine replaces the query name *id* by the expression *e*. We use the schema access APIs exposed in the MetaOQL wrapper of the specific ODB in these named queries to access the metadata. Due to the space limitation, we only list the named queries for the metadata set of class names, the full path attribute expression of local attributes (similar to the metadata set of full path attribute expression of local and inherited attributes) and the immediate superclass names (similar to the metadata set of all superclass names, immediate subclass names and all subclass names). All the bold words in the following named queries vary with different wrapper APIs.

1. -> metadata set of class names defined in the schema

*define getClasses as element(select s.***getClasses** *from* ***MetaScopes*** [5] *s);*

2. *cl*-> metadata set of full path attribute expressions of local attributes defined in a class

```
define getLocalAttrs(CName) as
  element(select c
          from MetaClasses [6] c
          where c.getClassName = CName).getLocalAttrs;
```

It should be mentioned here that the new type *full path attribute expression type* is introduced here. We mentioned before that set expressions *cl*-> and *cl*->* are of type *set<full path attribute expression type>*. For users who write in MetaOQL, it is a new type that we introduced into the MetaOQL type system. However, at the implementation level, it is in fact a system dictionary defined type *MetaAttribute*. The reason that we do not introduce it to the users as type *MetaAttribute* is that we do not want to expose the concept of *meta-object* to the users in order to hide the low-level implementation details of the schema repository. The users do not need to be aware of the internal details of meta-objects at all. MetaOQL supplies an easy mechanism to refer to *MetaAttribute* objects by *full path attribute expression*. The new type is more intuitive to the users and easier to understand than the concept of *meta-object*.

3. *cl*+ immediate superclass name set

```
define immediatesuperClass(CName)
  element(select c
          from MetaClasses c
          where c.getClassName = CName).getSupers;
```

[5] *MetaScopes* is the extent of class *MetaScope*. This is a naming convention in ODB.

[6] *MetaClasses* is the extent of class *MetaClass*.

**Translation of *get-type* Operator** For translating *get-type* operator, we define two named queries for the *get-type* operator. Both named queries return the type name of a given attribute. The first query definition *getTypeFrName* takes two parameters *CName* and *AttrName*. The parameter *CName* is a class name and the parameter *AttrName* is the name of an attribute defined in the class specified by the parameter *CName*.

*define getTypeFrName(CName,AttrName) as*
*element(select attr.**getAttrType**.**getName***
*from getLocalAttrs(CName) attr*
*where attr.**getAttrName** = AttrName);*

The second query definition *getTypeFrMeta* only takes one parameter *AttrMeta*. The parameter *AttrMeta* will be bound to a full path attribute expression.

*define getTypeFrMeta(AttrMeta) as MetaAttr.**getAttrType**.**getName***;

Then, when the translation processor encounters a *get-type* operator,

- if the operand is of the form $a$ where $a$ is a variable that ranges over the metadata set defined in Definition 1 (b) or 1 (c), replace @$a$ with *getTypeFrMeta(*a*)*.
- if the operand is of the form $C.A_1.A_2...A_n$ where $C$ is a fixed class name and $A_1$, $A_2$, ..., $A_n$ are fixed attribute names, replace @(C.$A_1.A_2...A_n$) with the following expression:
`getTypeFrName(getTypeFrName(...getTypeFrName(getTypeFrName(`$C$`,`$A_1$`),`$A_2$`)...),`$A_n$`).`

*Example 5.* @(Person.address.city) is translated into the following expression:

`getTypeFrName(getTypeFrName(''Person'', ''address''), ''city'');`

`getTypeFrName(''Person'', ''address'')` will return the type name of the attribute `address` defined in class `Person`, i.e., "Address", and then `getFrName(''Address'', ''city'')` will return the type name of attribute `city` defined in class `Address`, i.e., "String".

**Translation of *get-name* Operator** For translating the *get-name* operation, we define the following named query *getAttrName* with a parameter *AttrMeta*.

`define getAttrName(MetaAttr) as AttrMeta.`***getAttrName***;

Then the processor translates $|attr|$ where *attr* is an variable of type *MetaAttribute* into *getAttrName(attr)*.

## 5 Related Work

XSQL [KKS92] is a language that is capable of querying and restructuring ODBs. It has an SQL like syntax and can express sophisticated queries in a concise way. This is achieved via extended path expressions which may have variables that range over classes, attributes, and methods. Unlike OQL, it is possible to query data without complete knowledge of the schema. However, the complex nature of XSQL raises concerns about effective and efficient implementation, a concern not addressed in their work.

Also in the relational world, several papers [KLK88,CKW89,KLK91,LSS93,KLJ95] have appeared in the literature that address the meta-data dependency problem. The solutions proposed in [KLK88,CKW89,KLK91] augment the query language with mechanisms that allow it to query both meta-data and ordinary data. These solutions are embedded in very powerful obeject-oriented query languages. Following the work [LS93], schemaSQL [LSS96,LSS99] is a recently proposed extension to SQL designed for multi-database interoperability. SchemaSQL retains the flavor of SQL while supporting the manipulation of data and metadata. SchemaSQL permits four additional types of variables in the `from` clause: db-name, rel-name, attr-name and domain variables besides the tuple variables SQL already supports. The db-name variable ranges over a set of database names in a multi-database federation. The rel-name variable ranges over a set of relation names in a database. The att-name variable ranges over a set of names of attributes in a relation. The domain variable ranges over a set of values appearing in a column in a relation.

MetaOQL differs from SchemaOQL in the following aspects:

- SchemaOQL only queries the metadata of schema labels, i.e., relationa labels and field labels. It does not query other meta-information for example a field's type. Only querying the those counterpart in ODB is not sufficient for supporting schema evolution. MetaOQL supports access to richer meta-information that are needed due to the object-oriented nature of the underlying data model such as an attribute's type besides its name;
- SchemaSQL focuses on the issue of interoperability between heterogeneous databases while MetaOQL focuses on improving portability and usability of the object query language. Hence the implementation of SchemaOQL focuses on conversion between metadata and data while that of MetaOQL focuses on translating general meta-query to system-specific query.

# 6 Conclusions and Future Work

**Summary.** In this paper we proposed a new language, MetaOQL, that addresses the limitations of portability and usability of metadata querying in ODBs. We introduce new set expressions for meta-information. These expressions are more than syntactic sugaring Our proposal of MetaOQL removes the tight coupling between metadata retrieval and the actual physical storage structure of the schema repository. Metadata queries are no longer tied to any proprietary schema repository. And the users do not need to be aware of low-level implementation details of the schema repository. They can write more concise, simple and transparent queries on metadata. Moreover, MetaOQL is a natural extension to OQL and can be implemented in a non-intrusive manner by adding a preprocessor (translator) above any existing OQL engine. Our proposed translator approach is:

– Efficient: The translation can be done as a one-pass process in a preprocessing phase;
– Non-intrusive: No extension of standard OQL processor is needed;
– Generic: Simply by providing a mapping table, the translator and hence MetaOQL would work for any ODMG compliant ODB without requiring any software development.

With the development of ODBs, more and more schema features are adopted by the ODMG standard. For instance, key as an integrity constraint has been supported in ODMG 2.0. Before that there is no ODB vendors providing schema evolution primitives supporting the transformations such as adding and dropping keys (users have to hardcode to ensure the integrity constraint while performing the schema evolution). Therefore, with new in-coming features embraced by the standard, we need to explore what other operators we may need to support the corresponding schema evolution.

## References


[Cea97] R. G. G. Cattell and D. Barry et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.

[CJR98a] K. T. Claypool, J. Jin, and E. A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[CJR98b] K. T. Claypool, J. Jin, and E. A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. Technical Report WPI-CS-TR-98-9, Worcester Polytechnic Institute, May 1998.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. Hilog as a platform for database languages. In *DBPL*, pages 37–44, 1989.

[CR99a] K. T. Claypool and E. A. Rundensteiner. Flexible Database Transformations: The SERF Approach . In *IEEE Data Engineering Bulletin 22(1)*, pages 19–24, 1999.

[CR99b] K. T. Claypool and E. A. Rundensteiner. SERF: An ODMG Based Extensible Schema Transformation System. Technical Report WPI-CS-TR-99-21, Worcester Polytechnic Institute, July 1999.

[Gem99] GemStone. http://www.gemstone.com/, 1999.

[Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. *SIGMOD*, pages 393–402, 1992.

[KLJ95] M. Kifer, G. Lausen, and Wu. J. Logical foundations of object-oriented and frame-based languages. In *JACM*, pages 741–843, 1995.

[KLK88] R. Krishnamurthy, W. Litwin, and W. Kent. Towards a real horn clause language. In *VLDB*, 1988.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Languages features for interoperability of databases with schematic discrepancies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 40–49, 1991.

[LS93] K. W. Liu and D. Spooner. Object-oriented database views for supporting multidisciplinary concurrent engineering. In *IEEE Computer Software and Applications Conference*, pages 19–25, November 1993.

[LSS93] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database system. In *DOOD*, pages 81–100, 1993.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *VLDB*, pages 239–250, 1996.

[LSS99] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On Efficiently Implementing SchemaSQL on a SQL Database System. In *VLDB*, pages 471–482, 1999.

[O'B97] P. O'Brien. Making Java Objects Persistent. *Java Report*, 1(1):49–60, 1997.

[Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., 1993.

[Obj99a] ObjectDomain. http://www.objectdomain.com/, 1999.

[Obj99b] ObjectStore. http://www.odi.com/objectstore/, 1999.

[RCL+99] E. A. Rundensteiner, K. T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, pages 568–570, 1999.

[Rea00] J. F. Roddick and L. Al-Jadir et al. Evolution and Change in Data Management - Issues and Directions. In *SIGMOD Record*, pages 21–25, March 2000.

[Tec94] $O_2$ Technology. *$O_2$ Reference Manual, Version 4.5, Release November 1994.* $O_2$ Technology, Versailles, France, 1994.