The ODMG Standard for Object Databases

Francois Bancilhon and Guy Ferran
O_2Technology
2685 Marine Way-Suite 1220
Mountain View, California 94043

Research on object databases started at the beginning of the 1980's and became very active in the mid 1980's. At the end of the 1980's, a number of start-up companies were created. As a result, a wide variety of products are now commercially available. The products have quickly matured after several years of market presence. Production applications are being deployed in various areas: CAD, software engineering, geographic information systems, financial applications, medical applications, telecommunications, multimedia and MIS.

As more and more applications were being developed and deployed and as more vendors appeared on the market, the user community voiced a clear concern about the risk of divergence of the products and expressed their need for convergence and standards. The return from the vendor community was both implicit and explicit. Implicit, because as the vendors understood more and more the applications and the user needs, the systems architecture started to converge and the systems to look alike. Explicit, because the vendors got together to define and promote a standard.

The Object Database Management Group (ODMG) was created in 1991 by five object database vendors under the chairmanship of Rick Cattell.

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

It published an initial version of a standard interface in 1993, it was joined by 2 more vendors at the end of 1993, and produced a final revision in early 1994 [Catt94].

Thus, all object database vendors are active members of the group and totally committed to comply to the standard in future releases of their products. For instance, the current release of O_2 [O_2] fully complies with the ODMG OQL query language, and the next release will comply with the ODMG C++ binding.

This standard is a major event and a clear signal for the object database market. It is clearly as important for object databases as SQL was for relational databases.

The ODMG standard is a *portability standard*, i.e. it guarantees that a compliant application written on top of compliant system X can be easily ported on top of compliant system Y, as opposed to an *interoperability standard*, such as CORBA, that allows an application to run on top of compliant systems X and Y at the same time. Portability was chosen because it was the first users demand.

Because object databases cover more ground than relational databases, the standard covers a larger area than SQL does. An object database schema defines the data structure and types of the objects of the database, but also the "methods" associated to the objects. Therefore a programming language must be used to write these methods.

Instead of inventing "yet another programming language", ODMG strongly believes that it is more realistic and attractive to use existing object oriented programming languages. More-

over, during the last few years a huge effort has been carried out by the programming language community and by the OMG organization to define and adopt a commonly accepted "object model". The ODMG contribution started from this state and proposed an object model which is a simple extension of the OMG object model.

This paper gives an informal presentation of the ODMG standard by presenting the data model, the query language and the language bindings. It emphasizes the query language which we consider essential in the standard. The rest of this paper is organized as follows: Section 1 describes the data model, Section 2 introduces the C++ binding, and Section 3 presents OQL.

1 Defining an Object Database Schema

In ODMG-93, a database schema can either be defined using the Object Definition Language (ODL), a direct extension of the OMG Interface Definition Language (IDL), or using Smalltalk or C++. In this presentation, we use C++ as our object definition language. We first briefly recall the OMG object model, then describe the ODMG extensions.

1.1 The OMG object model

The ODMG object model, following the OMG object model supports the notion of class, of objects with attributes and methods, of inheritance and specialization. It also offers the classical types to deal with date, time and character strings. To illustrate this, let us first define the elementary objects of our schema without establishing connection between them.

```
class Person{
   String name;
   Date   birthdate;

// Methods:
   Person();
   // Constructor: a new Person is born
   int age();
   // Returns an atomic type
};

class Employee: Person{
   // A subclass of Person
   float salary;
};

class Student: Person{
   // A subclass of Person
   String grade;
};

class Address{
   int number;
   String street;
};

class Building{
   Address address;
   // A complex value embedded
   // in this object
};

class Apartment{
   int number;
};
```

Let us now turn to the extensions brought by ODMG to the OMG data model: *relationships* and *collections*

1.2 One-to-one relationships

An object refers to another object through a *Ref*. A Ref behaves as a C++ pointer, but with more semantics. Firstly, a Ref is a persistent pointer. Then, referential integrity can be expressed in the schema and maintained by the system. This is done by declaring the relationship as symmetric. For instance, we can say that a Person lives in an Apartment, and that this Apartment is used by this Person, in the following way:

```
class Person{
   Ref<Apartment> lives_in
```

```
   inverse is_used_by;
};

class Apartment{
   Ref<Person>    is_used_by
   inverse lives_in;
};
```

The keyword **inverse** is the only ODMG extension to the standard C++ class definitions. It is, of course, optional. It ensures the referential integrity constraint: if a Person moves to another Apartment, the attribute "is_used_by" is automatically reset to NULL until a new Person takes this apartment again. Moreover, if an Apartment object is deleted, the corresponding "lives_in" attribute is automatically reset to NULL, thereby avoiding dangling references.

1.3 Collections

The efficient management of very large collections of data is a fundamental database feature. Thus, ODMG-93 introduces a set of predefined generic classes for this purpose: Set<T>, Bag<T> (a multi-set, i.e., a set allowing duplicates), Varray<T> (variable size array), List<T> (variable size and insertable array).

A collection is a container of elements of the same class. As usual, polymorphism is obtained through the class hierarchy. For instance a Set<Ref<Person>> may contain Persons as well as Employees, if the class Employee is a subclass of the class Person.

In a database schema, collections may be used to record extents of classes. In the ODMG-93 C++ binding, the extent of a class is not automatically maintained, and the application itself creates and maintains explicitly a collection, whenever an extent is really needed. This management can be easily encapsulated in creation methods, and furthermore, the application can define as many collections as needed to group objects which match some logical property. For instance, the following collections can be defined:

```
Set<  Ref<Person> >     Persons;
  // The Person class extent.
Set<  Ref<Apartment> > Apartments;
  // The Apartment class extent.
Set<  Ref<Apartment> > Vacancy;
  // The set of vacant apartments.
List< Ref<Apartment> > Directory;
  // The list of apartments.
  // ordered by their number of rooms.
```

1.4 Multiple Relationships

Very often, an object is related with more than one object through a relationship. Therefore, the notion of 1-1 relationship defined previously has to be extended to 1-n and n-m relationships, with the same guarantee of referential integrity.

For example, a Person has two Parents and possibly several Children; in a Building, there are many Apartments.

```
class Person{
   Set  < Ref<Person> > parents
     inverse children; // 2 parents.
   List < Ref<Person> > children
     inverse parents;
     // Ordered by birthday
};

class Building{
   List< <Ref<Apartment> > apartments
     inverse building;
     // Ordered by apartment number
};

class Apartment{
   int number;
   Ref<Building> building
     inverse apartments;
};
```

1.5 Naming

ODMG-93 enables explicit names to be given to any object or collection. From a name, an application can directly retrieve the named object and then operate on it or navigate to other objects following the relationship links.

A name in the schema plays the role of a variable in a program. Names are entry points in the database. From these entry points, other objects (in most cases unnamed objects) can be reached through associative queries or navigation. In general, explicit extents of classes are named.

1.6 The sample schema

Let us now define our example schema completely.

```
class Person{
   String name;
   Date   birthdate;
   Set  < Ref<Person> > parents
     inverse children;
   List < Ref<Person> > children
     inverse parents;

   Ref<Apartment> lives_in
     inverse is_used_by;

// Methods:
   Person();
   // Constructor: a new Person is born
   int age();
   // Returns an atomic type
   void marriage( Ref<Person> spouse);
   // This Person gets a spouse
   void birth( Ref<Person> child);
   // This Person gets a child
   Set< Ref<Person> > ancestors;
   // Set of ancestors of this Person
   virtual Set<String> activities();
   // A redefinable method
};

class Employee: Person{
   // A subclass of Person
   float salary;
// Method
   virtual Set<String> activities();
   // The method is redefined
};

class Student: Person{
   // A subclass of Person
   String grade;
// Method
   virtual Set<String> activities();
   // The method is redefined
};

class Address{
   int number;
   String street;
};

class Building{
   Address address;
   // A complex value
   // embedded in this object
   List< <Ref<Apartment> > apartments
     inverse building;
// Method
   Ref<Apartment> less_expensive();
};

class Apartment{
   int number;
   Ref<Building> building;
   Ref<Person> is_used_by
     inverse lives_in;
};


Set<  Ref<Person> >    Persons;
  // All persons and employees
Set<  Ref<Apartment> > Apartments;
  // The Apartment class extent
Set<  Ref<Apartment> > Vacancy;
  // The set of vacant apartments
List< Ref<Apartment> > Directory;
  // The list of apartments
  // ordered by their number of rooms
```

2 C++ Binding

To implement the schema defined above, we write the body of each method. These bodies can be easily written using C++. In fact, because Ref<T> is equivalent to a pointer (T*), manip-

ulating persistent objects through Refs is done in exactly the same way as through normal pointers.

To run applications on a database instantiating such a schema, ODMG-93 provides classes to deal with Databases (with open and close methods) and Transactions (with start, commit and abort methods). When an application creates an object, it can create a transient object which will disappear at the end of the program, or a persistent object which will survive when the program ends and can be shared by many other programs possibly running at the same time. Here is an example of a program to create a new persistent apartment and let "john" move into it.

```
Transaction move;
move.begin();
   Ref< Apartment > home =
     new(database) Apartment;
   Ref< Person >    john =
     database->lookup_object(''john'');
   // Retrieve a  named object
   Apartments.insert_element(home);
   // Put this new apartment
   // in the class extent
   john->lives_in = home;
   // Persistent objects are handled
   // as standard C++ objects
move.commit();
```

3 Object Query Language, OQL

ODMG-93 introduces a query language, OQL. OQL is an SQL style language that allows easy access to objects. We just presented an object definition language (using C++) and a C++ binding. We strongly believe that these two languages are not sufficient for writing database applications and that many situations require a query language:

- Interactive ad hoc queries

  A database user should not be forced to write, compile, link edit and debug a C++ program just to get the answer to simple queries. OQL can be used directly as a stand alone query interpreter. Its syntax is simple and flexible. For someone familiar with SQL, OQL can be learned in a few hours.

- Simplify programming by embedded queries

  Embedded in a programming language like C++, OQL dramatically reduces the amount of C++ code to be written. OQL is powerful enough to express in one statement a long C++ program.

  Besides, OQL directly supports the ODMG model. Therefore, OQL has the same type system as C++ and is able to query objects and collections computed by C++ and passed to OQL as parameters. OQL then delivers a result which is put directly into a C++ variable with no conversion. This definitively solves the well known "impedance mismatch" which makes embedded SQL so difficult to use, since SQL deals with "tables" not supported by the programming language type system.

- Let the system optimize your queries

  Well known optimization techniques inspired from relational technology and extended to the object case can be used to evaluate an OQL query by virtue of its declarative style. For instance, the OQL optimizer of O_2 finds the most appropriate indexes to reduce the amount of data to be filtered. It factorizes the common subexpressions, finds out the expressions which can be computed once outside an iteration, pushes up the selections before starting an inner iteration.

- Logical/Physical independence

  OQL differs from standard programming languages in that the execution of an OQL query can be dramatically improved without modifying the query itself, simply by declaring new physical data structures or new indexing or clustering strategies. The

optimizer can benefit from these changes and then reduce the response time.

Doing such a change in a purely imperative language like C++ requires an algorithm to be completely rewritten because it cannot avoid making explicit use of physical structures.

- Higher level constructs

  OQL, like SQL provides very high level operators which enables the user to sort, group or aggregate objects or to do statistics, all of which would require a lot of C++ tedious programming.

- Dynamicity

  C++ is a compiled programming language which requires heavy compiling and link edition. This precludes having a function dynamically generated and executed by an application at runtime. OQL does not suffer from this constraint since a query can be dynamically computed immediatly.

- Object server architecture

  The new generation of Object Database Systems have a very efficient architecture. For instance, O_2 has a page server which minimizes the bottleneck in multi-user environment and draws all the benefits of the CPU and Memory available on the client side which holds visited objects in its own memory.

  This architecture suits local area network applications very well. For wide area network and/or more loosely coupled database applications this architecture can be supplemented by an OQL server, where the client sends a query to the server which is completely executed on the server side. Without a query language, this architecture is impossible.

- SQL like

  Object Database Systems must propose the equivalent of relational systems, i.e., a query language like SQL. Whenever possible, OQL looks like SQL. This facilitates the learning of OQL and facilitates its acceptance.

- Support for advanced features (views, integrity constraints, triggers)

  Finally, without OQL it would be impossible to offer advanced services in object database systems. Features such as views, triggers and integrity constraints need a declarative language.

Let us now turn to an example based presentation of OQL. We use the database described in the previous section, and instead of trying to be exhaustive, we give an overview of the most relevant features.

3.1 Path expressions

As explained above, one can enter a database through a named object, but more generally as soon as one gets an object (which comes, for instance, from a C++ expression), one needs a way to "navigate" from it and reach the right data one needs. To do this in OQL, we use the "." (or indifferently "->") notation which enables us to go inside complex objects, as well as to follow simple relationships. For instance, if we have a Person "p" and we want to know the name of the street where this person lives, the OQL query is:

```
p.lives_in.building.adddress.street
```

This query starts from a Person, traverses an Apartment, arrives in a Building and goes inside the complex attribute of type Address to get the street name.

This example treated 1-1 relationship, let us now look at n-p relationships. Assume we want the names of the children of the person p. We cannot write: p.children.name because "children" is a List of references, so the interpretation of the result of this query would be undefined. Intuitively, the result should be a

collection of names, but we need an unambiguous notation to traverse such a multiple relationship and we use the select-from-where clause to handle collections just as in SQL.

```
select c.name
from c in p.children
```

The result of this query is a value of type Bag<String>. If we want to get a Set, we simply drop duplicates, like in SQL by using the "distinct" keyword.

```
select distinct c.name
from c in p.children
```

Now we have a means to navigate from an object towards any object following any relationship and entering any complex subvalues of an object.

For instance, we want the set of addresses of the children of each Person of the database. We know the collection named "Persons" contains all the persons of the database. We have now to traverse two collections: Persons and Person::children. Like in SQL, the select-from operator allows us to query more than one collection. These collections then appear in the "from" part. In OQL, a collection in the "from" part can be derived from a previous one by following a path which starts from it, and the answer is:

```
select c.lives_in.building.address
from p in Persons,
     c in p.children
```

This query inspects all children of all persons. Its result is a value whose type is Bag<Address>.

Predicate

Of course, the "where" clause can be used to define any predicate which then serves to select only the data matching the predicate. For instance, we want to restrict the previous query to the people living on Main Street, and having at least two children. Moreover we are only interested in the addresses of the children who do not live in the same apartment as their parents. And the query is:

```
select c.lives_in.building.address
from p in Persons,
     c in p.children
where p.lives_in.building.address.street
      = ''Main Street'' and
      count(p.children) >= 2 and
      c.lives_in != p.lives_in
```

Join

In the "from" clause, collections which are not directly related can also be declared. As in SQL, this allows us to compute "joins" between these collections. For instance, to get the people living in a street and have the same name as this street, we do the following: the Building extent is not defined in the schema, so we have to compute it from the Apartments extent. To compute this intermediate result, we need a select-from operator again. This shows that in a query where a collection is expected, this can be computed recursively by a select-from-where operator, without any restriction. So the join is done as follows:

```
select p
from p in Persons,
     b in (select distinct a.building
     from a in Apartments)
where p.name = b.address.street
```

This query highlights the need for an optimizer. In this case, the inner select subquery must be computed once and not for each person!

3.2 Complex data manipulation

A major difference between OQL and SQL is that object query languages must manipulate complex values. OQL can therefore create any complex value as a final result, or inside the query as intermediate calculation.

To build a complex value, OQL uses the constructors `struct`, `set`, `bag`, `list` and `array`. For example, to obtain the addresses of the children of each person, along with the address of this person, we use the following query:

```
select struct(
  me: p.name,
  my_address:
    p.lives_in.building.address,
  my_children:
   (select struct(
     name: c.name,
     address:
       c.lives_in.building.address)
     from c in p.children))
from p in Persons
```

This gives for each person the name, the address, and the name and address of each child and the type of the resulting value is:

```
struct result_type{
   String me;
   Address my_address;
   Bag<struct{String name;
              Address address}>
    my_children;
}
```

OQL can also create complex objects. For this purpose, it uses the name of a class as a constructor. Attributes of the object of this class can be initialized explicitly by any valid expression.

For instance, to create a new building with 2 apartments, if there is a type name in the schema, called List_apart, defined by: tydedef List<<Ref<Apartment> > List_apart; the query is:

```
Building
  (address: struct(
              number: 10,
              street: ''Main street''),
   apartments:
     List_apart(Apartment(number: 1),
                Apartment(number: 2)))
```

3.3 Method invoking

OQL allows us to call a method with or without parameters anywhere the result type of the method matches the expected type in the query. The notation for calling a method is exactly the same as for accessing an attribute or traversing a relationship, in the case where the method has no parameter. If it has parameters, these are given between parenthesis.

This flexible syntax frees the user from knowing whether the property is stored (an attribute) or computed (a method). For instance, to get the age of the oldest child of the person "Paul", we write the following query:

```
select max(select c.age
           from c in p.children)
from p in Persons,
where p.name = ''Paul''
```

Of course, a method can return a complex object or a collection and then its call can be embedded in a complex path expression. For instance, inside a building b, we want to know who inhabits those least expensive apartment. The following path expression gives the answer:

```
b.less_expensive.is_used_by.name
```

Although "less_expensive" is a method we "traverse" it as if it were a relationship.

3.4 Polymorphism

A major contribution of object orientation is the possibility of manipulating polymorphic collections, and thanks to the "late binding" mechanism, to carry out generic actions on the elements of these collections.

For instance, the set "Persons" contains objects of class Person, Employee and Student. So far, all the queries against the Persons extent dealt with the three possible "objects" of the collection. If one wants to restrict a query on a subclass of Person, either the schema provides an extent for this subclass which can then be queried directly, or else the super class extent can be filtered to select only the objects of the

subclass, as shown in the example below with the "class indicator".

A query is an expression whose operators operate on typed operands. A query is correct if the type of operands matches those required by the operators. In this sense, OQL is a typed query language. This is a necessary condition for an efficient query optimizer.

When a polymorphic collection is filtered (for instance Persons), its elements are statically known to be of that class (for instance Person). This means that a property of a subclass (attribute or method) cannot be applied to such an element, except in two important cases: late binding to a method, or explicit class indication.

Late binding

Give the activities of each person.

```
select p.activities
from p in Persons
```

"activities" is a method which has 3 incarnations. Depending on the kind of person of the current "p", the right incarnation is called.

Class indicator

To go down the class hierarchy, a user may explicitly declare the class of an object that cannot be inferred statically. The interpreter then has to check at runtime, that this object actually belongs to the indicated class (or one of its subclasses).

For example, assuming we know that only "students" spend their time in following a course of study, we can select those persons and get their grade. We explicitly indicate in the query that these persons are students:

```
select (Student)p. grade
from   p in Persons
where  ''course of study''
       in p.activities
```

3.5 Operator composition

OQL is a purely functional language: all operators can be composed freely as soon as the type system is respected. This is why the language is so simple and its manual so short.

This philosophy is different from SQL, which is an ad-hoc language whose composition rules are not orthogonal. Adopting a complete orthogonality, allows us not to restrict the power of expression and makes the language easier to learn without losing the SQL style for the simplest queries.

Among the operators offered by OQL but not yet introduced, we can mention the set operators (union, intersect, except), the universal (for all) and existential quantifiers (exists), the sort and group by operators and the aggregative operators (count, sum, min, max and avg).

To illustrate this free composition of operators, let us write a rather complex query. We want to know the name of the street where employees live and have the smallest salary on average, compared to employees living in other streets. We proceed step by step and then do it all at once. We can use the "define" OQL instruction to evaluate temporary results.

1. Build the extent of class Employee (not supported directly by the schema)

   ```
   define Employees as
          select (Employee) p
          from p in Persons
          where ''has a job''
                in p.activities
   ```

2. Group the employees by street and compute the average salary in each street

   ```
   define salary_map as
   group  e in Employees
   by (street:
   e.lives_in.building.address.street)
   with (average_salary:
            avg(select x.salary
            from x in partition))
   ```

   The group by operator splits the employees into partitions, according to the criterion (the name of the street where this person lives). The "with" clause computes, in

each partition, the average of the salaries of the employees belonging to this partition.

The result of the query is of type:

```
Set<struct{String street;
          float average_salary;}>
```

3. Sort this set by salary

```
define sorted_salary_map as
       sort s in salary_map
            by s.average_salary
```

The result is now of type

```
List<struct{String street;
            float average_salary;}>
```

4. Now get the smallest salary (the first in the list) and take the corresponding street name. This is the final result.

```
sorted_salary_map[0].street
```

5. In a single query we could have written:

```
(sort s in(
 group  e in (select (Employee) p
              from p in Persons
              where ''has a job''
                in p.activities)
 by (street:
 e.lives_in.building.address.street)
 with (average_salary:
        avg(select x.salary
            from x in partition))
)by s.average_salary)[0].street
```

3.6 C++ embedding

An *oql* function is provided as part of the ODMG-93 C++ binding. This function allows to run any OQL query. Input parameters can be passed to the query. A parameter is any C++ expression. Inside the sentence, a parameter is referred to by the notation:

```
$<position><type>
```

where "position" gives the rank of the parameter and "type" is a tag indicating the kind of the parameter ("o" means object, "c" means collection, "i" integer, etc.). Let us now write as an example the code of the "ancestors" method of the class Person. This example shows how a recursive query can be easily written that gives OQL the power of a recursive query language.

Recursive query

```
Set < Ref<Person>>
   Person::ancestors(){
   Set < Ref<Person> > result;
   oql(result,
       "flatten
          (select distinct a->ancestors
           from a in $1c)
           union $1c", parents);
   return result;
};
```

"$1c" refers to the first parameter, i.e, the Set "parents". In the select clause, we compute the set of ancestors of each parent. We get therefore a set of sets. The "flatten" operator converts this set of sets into a simple set. Then, we take the union of this set with the parent set. The recursion stops when the parents set is empty. In this case, the select part is not executed and the result is the empty set.

4 Conclusion

ODMG-93 provides a complete framework within which one can design an object database, write portable applications in C++ or Smalltalk, and query the database with a simple and very powerful query language. Based on the OMG, SQL, C++ and Smalltalk standards, available today in industrial products such as O_2, it is supported by the major actors of the object database world.

ODMG-93 will of course be improved over time. New features will be added to the standard to make it more complete. The ODMG group is currently working on extensions of the

standard and on the convergence of OQL with SQL.

References

[Catt94] Rick Cattell and al. The Object Database Standard: ODMG-93, release 1.1. Morgan Kaufmann, 1994

[O_2] O_2Technology. The O_2 User Manual, release 4.5.