Reinhard Karge

Object-oriented views

May 2004

March 2007 (revised)

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, October 2012

Content

1 Introduction

ODABA2

ODABA2 is an object-oriented database system that allows storing objects and methods as well as causalities. As an object-oriented database, ODABA2 supports complex objects (user-defined data types), which are built on application relevant concepts.

ODABA2-applications are characterised by a high flexibility that is achieved by supporting in addition to object (concept) hierarchy, multifarious relations between objects (master and detail relations, relations between independent objects and others). This way conditions and behaviour of objects in the real world can be represented considerably better than in relational systems.

ODABA2-applications cannot only be drawn up as event-driven applications within the field of the graphical surface but also at the database level. This is one more way in which the application design is very close to the problem.

This makes ODABA2-applications a favourite possibility to solve highly complex jobs as come up in administrative and knowledge areas.

Platforms

ODABA2 supports windows platforms (Windows95/98/Me, Windows NT and Windows 2000) as well as UNIX platforms (Linux, Solaris).

You can build local applications or client server applications with a network of servers and clients.

Interfaces

ODABA2 supports several technical interfaces:

- C++, COM as application program interface (this allows e.g. using ODABA2 in VB scripts and applications)
- ODBC (for data exchange with relational databases)
- XML (as document interface as well as for data exchange)

User Interfaces

ODABA2 provides special COM-Controls that easily allow building applications in Visual Basic. On the other hand ODABA2 provides a special ODABA2 GUI builder.

2 Overview

The ODABA Scrip Interface (OSI) defines an expression syntax that is based on the OMG standard of the database management group ODMG ’2003 (OQL). It can be used as a query language as well as an expression language for defining expressions or functions for ODABA object classes.

OSI allows defining SQL like queries for an ODABA database. In addition, OSI provides a JAVA like expression language, which allows providing complex functionality. Thus, OSI consists of two parts, a common expression language and the SQL syntax that may refer to the expression language.

Expressions defined by means of OSI are always creating a result, i.e. they will not produce output like tables. The result, however, can be an elementary value, a complex instance, a set of values or a set of instances.

OSI supports to essential types of queries, OQL expressions and access paths. Expressions may contain access paths as well as an access path may contain expressions in different places.

Access Path

An access path is an extension of the OQL query language considering directives of standard query languages (as select, from where …) as operations. This allows calling different operations in any order passing the result from each operation to the next operation.

OQL Expression

An OSI expression usually consists of a set of statements, which are executed sequentially. As well as an access path an expression returns a value, which might be atomic or complex. Within an OSI expression you may process statements conditionally (if, switch)) or in loops (while). The expression syntax is Java or C++ like and thus, familiar to many developers.

Document generation

Beside basic expressions that can be defined in OSI, an extended expression syntax is provided that allows the specification of document expressions or document templates. Document templates can be used to create documents (ASCII, RTF or HTML documents) by means of expressions.

Running OSI

OSI can be executed in several ways. Practically, all the possibilities are equivalent, i.e. you can express the same with each of this features. The difference is mainly the environment, where those expressions can be used.

C++ program

In a C++ program OSI expressions can be imbedded as access paths (OpenAccessPath) or by calling appropriate expression functions (ProvideExprDecl, Execute). Both ways return a property handle that allows accessing an elementary value or a collection of returned instances.

OShell Utility

OShell allows running any database function from a command line environment. Thus, OShell can be used to open access paths as well as defining and executing expressions like in a C++ program.

View definition

View definitions or predefined views are another way of defining queries, which may refer to expressions and access paths. Pre-defined views are part of the database model and can be accessed like normal instance collections.

Examples

The document will refer to examples based on a simple structure.

Ex1: We consider **Persons** and **Accounts**. An account might be owned by one or more persons. A person may have any number of children and each person has an income.

This can be expressed in ODL as follows:

```
class Person (extent persons)
{
  attribute string id;
  attribute float salery;
  relationship set<Account> accounts
               inverse Account::owner;
  relationship set<Person> children
               inverse Person::parents;
  relationship set<Person> parents
               inverse Person::children;
}
```

```
class Account (extent accounts)
{
  attribute string acc_no;
  attribute float saldo;
  relationship set<Person> owner
               inverse Person::account;
}
```

The relational definition would look like:

```
create table Person (
        id int IDENTITY (1, 1),
        name varchar(50),
        salary float(53),
        );
create table BankAccount (
        id int IDENTITY (1, 1),
        [value] float(53),
        number int
        );
create table ParentChild (
        id int IDENTITY (1, 1),
        child int,
        parent int
        );
create table PersonBankAccount (
        id int IDENTITY (1, 1),
        person int,
        account int
        );
ALTER TABLE dbo.Person
  ADD         CONSTRAINT         FK_Person_Partner
FOREIGN KEY  (  partner  )  REFERENCES  dbo.Person  (
id )
ALTER TABLE dbo.BankAccount
  ADD        CONSTRAINT        FK_BankAccount_Owner
FOREIGN KEY  (  owner  )    REFERENCES  dbo.Person  (
id )
ALTER TABLE dbo.ParentChild
  ADD       CONSTRAINT       FK_ParentChild_Parent
FOREIGN KEY  (  parent  )   REFERENCES  dbo.Person  (
id )
ALTER TABLE dbo.ParentChild
  ADD        CONSTRAINT        FK_ParentChild_Child
FOREIGN KEY  (  child  )    REFERENCES  dbo.Person  (
id )
ALTER TABLE dbo.PersonBankAccount
  ADD    CONSTRAINT    FK_PersonBankAccount_Person
FOREIGN KEY  (  person  )   REFERENCES  dbo.Person  (
id )
ALTER TABLE dbo.PersonBankAccount
  ADD   CONSTRAINT   FK_PersonBankAccount_Account
FOREIGN    KEY     (     account     )     REFERENCES
dbo.BankAccount ( id )
```

Because of the M:N relationship between persons and accounts and persons and children we have to introduce appropriate intermediate tables.

Sample queries

There is a list of three queries, which we are going to compare. We will start with a simple example as getting the total income from children, going to a more difficult query as getting the total amount of money on a persons account, considering, that the money for each person is the sum of money on person's accounts supposing that common accounts are owned by equal parts for all account owners. The third query is simply combining the first two.

SQL

Appropriate SQL statements would look as follows:

```
select
    pers_id,
    chld_income: sum(select income
                     from p.children)
From persons p
```

```
select
    pers_id,
    tot_saldo: sum {
       select
           saldo: a.saldo/count(a.owners)
       from p.accounts a )
From persons p
```

```
select
    id: p.id,
    chld_income: sum(select salary: c.salary
                     from p.children as c)
    tot_saldo: sum {
       select
           psaldo: a.saldo/count(a.owners)
       from p.accounts as a )
from persons as p; //  (147 characters)
```

OSI

Using OSI with standard OQL features, the three queries could be defined as follows:

```
Select  pers_id,
    children{chld_income = Sum(income);}
From persons;
```

```
Select  pers_id,
    accounts{
       tot_saldo = Sum(saldo/owners.GetCount};}
From persons;
```

```
Select  pers_id,
    children{chld_income = Sum(income);},
    accounts{
       tot_saldo=Sum(saldo)/owners.GetCount};}
From persons;
```

Obviously, the last two cases are more difficult to define and to interpret. This was the reason to consider a more sophisticated way of using property paths with an extended meaning.

Enhanced OSI

Enhanced OSI uses some specific extensions for access paths, which allow defining the query as path expression.

```
persons.select {
  id,
  chld_income = Children->sum(income)
};
```

```
persons.select (
  id,
  tot_saldo   = accounts.select(
                part:saldo/owners->count)->
                      sum(part)
);
```

```
person.select {
  id,
  chld_income = children->sum(income)
  tot_saldo   = accounts.select(
                  psaldo:saldo/owners.count)->
                      sum(psaldo)
}; // (119 characters)
```

Obviously, enhanced OSI provides shortest way of expressing the query. But it seems also, that the idea of the query is easier to grasp in the last case, which makes programs easier to understand. Moreover, enhanced OSI paths can be used within other OSI expressions.

Since traditional queries and access paths will produce the same result, you may use the one or the other way.

3 Property path extensions

There is no doubt about the usefulness of property paths. The problem, however, is that property paths can be interpreted in different ways. Here we will introduce a syntax that provides a unique interpretation for property paths and leads to extreme simplifications in query expressions.

Path definition

According to the OMG standard [1] a path is a sequence of property names according to the type definitions. This allows addressing complex properties as well as traversing references. In [3] we have shown that each property path corresponds to a schema, which includes an extensional and an intentional (type) definition. Moreover, it has been shown, that combining property paths with operations leads to a more general definition for a path.

Each property path has an origin, which might be either an object identifier or an extent.

```
Persons().accounts().owners
```

Moreover, it has a terminal property, which determines the result schema. The expression above refers to a collection (bag) of all owners for the accounts for all persons. This, obviously defines a person collection. According to the OMG standard [1] we can also combine property paths with operations which syntactically may lead to expressions as:

```
persons.IncomeGroup()
```

Operations may get parameters. Since operations are schemata and schemata are operations [3], we can say in general, that a path is defined as:

```
path          := element {.element} |
                 element {->element}
element       := path | [path] | name (parm_list)
name          := prop_name | oper_name
parm_list     := parameter {, parameter}
```

We will not focus here on what parameters means in detail. The important thing is that each element returns a type, which allows checking, whether the succeeding element name is defined as member of the preceding element type.

There are two possibilities to modify the schema defined by an element in the property path. We can modify the extension of the result as well as the intention. Hence, we added two additional options to the definition of an element in a path.

Sub-paths

There are always two ways interpreting a path. Considering the path **`persons().accounts`**, we can interpret this as a collection (bag) of all accounts owned by persons, but also as an inner join between persons and accounts, which results in a person/account instance. In OSI this difference could be expressed as:

```
Select a from persons.accounts as a
```

```
Select p, a from persons as p, p.accounts as a
```

Within a property path we can use sub-paths to differ between inner joins and "last property type". We use the [] operator to indicate inner joins and interpret paths not enclosed in [] as returning a type according to the last referenced property:

```
Persons().accounts      type: Account
```

```
[persons().accounts]    type: Person, Account
```

The type (implicitly defined class) for an inner join is always based on the types involved in the path, i.e. in the example the result type is based on **`Person`** and **`Account`**. This allows referring to methods defined in the referenced base classes.

Considering the path **`[person().children]`**, which is based on **`Person`**, **`Person`**, names become ambiguous. This ambiguity can be avoided by prefixing referenced names always with the sub-path addressing the appropriate level in the path (e.g. **`person.name`** or **`children.name`**)

Practically, paths may include references (relationships) as well as attributes within complex types. Theoretically, this makes, however, no difference. It just not makes much sense to define a path like **`address.city`** as inner join **`[address.city]`** although this would be possible.

Sub-paths in a path can be nested within a path.

Join path

Usually, a path begins with a property or method valid in the given context. Since OSI may refer to sources, which cannot be expressed as simple paths, a generic operation is required, which allows combining independent paths. Therefore, we can use the ‘from’ operator in the same sense as defined in OQL.

```
Join_path       := from( path_ref {,path_ref} )
Path_ref        := [name:] path
```

The **from** operator can be used for simple paths as well, but it is not required. The advantage using the **from** operator instead of the **from** clause in the OQL statement is, that the from operator defines an operation, which can be used in a path. This allows reading an expression “from left to right” instead “from inner to outer” as in an OQL statement.

The output for a **from** operation is the Cartesian product of all paths passed as parameters to the operation. It creates a type (class), which is based on the types (classes) of all paths involved. Names can be given to paths to avoid ambiguity.

Instance filter

Instance filters are expressed in an OQL statement in **where** and **having** clauses. Within a path the position of the filter defines its scope and we need not to differ between **where** and **having**. Instead, we will always use the operation **where**, which selects the instances from the preceding path expression.

Filters allow changing the extension of a collection, but do not change the structure of the instances. Usually, a filter defines a condition, which is true for all instances in the result schema. Instances, which return false (or empty) for the condition, are not considered as instances of the result collection. Because filters can be used for any element in the path, they are easier to use in complex queries than where clauses.

The expression for an instance filter must be defined in the scope of the schema, which results from the preceding element or sub-path.

```
persons.where(income>2000).
        children.where(age>10)
```

This expression returns all children older than 10 years, which have a parent getting more than 2000 EURO per month. This is a simple way of expressing the OQL-query:

```
select c form persons p, p.children c
         where p.income > 2000 and c.age > 10
```

Using filter in a path expression does not only improve the readability of an expression but automatically leads to an expression optimization.

A filter is a special **element** in an OSI path with the following definition:

```
filt_elmt      := op_name( condition )
op_name        := where | having
condition      := valid OSI expression
```

For compatibility reasons, we will use **where** and **having** in a path expression. The filter condition (expression) must be a valid expression in the context of the type defined by the preceding element (sub-path).

Result type

Each element or sub-path in a path expression implicitly or explicitly defines type. The type for the result depends on the property or operation in the path. For generic operations as filter or **from** the rules for constructing the result type have been defined. User defined operations usually have an defined output type.

It is a general requirement, that each element or sub-path in a path returns a defined type. Sometimes, the type can be determined at runtime, only (e.g. when the collection is weak typed or untyped). In OQL this feature is called late binding. The consequence is, that the expression cannot be validated in advance. When running the expression, however, each instance returned as result must be associated with a defined type.

In some cases the data type of the result needs to be defined explicitly in the OQL statement (**select** clause).

A type operation is a special **element** in an OSI path with the following definition:

```
type_elmt      := select( type_def )
type_def       := prop_list                  |
                  type_name [ (prop_list) ] |
                  * [ , prop_list ]
prop_list      := prop_spec [ ,prop_spec ]
prop_spec      := prop_name [ = src_expr ]
src_expr       := valid OSI expression
```

For compatibility reasons we will use **`select`**, instead of type, although **`type`** would be more appropriate. The source expressions for properties (srce_expr) must be a valid expression in the context of the type defined by the preceding element (sub-path).

This includes two ways of defining target structures. One is a typed target instance, which is defined by referring to a type name. The type name is a pre-defined type in the database or OSI expression. When a type name is passed with no property list, property values are assigned by name, i.e. all properties in the result instance of the element are assigned to properties with the same name in the target instance. When using typed target with a property list, the property list may contain property names defined in the target type, only.

When not defining a type name, but a property list only, the target is considered as untyped in a sense, that all instances in the target collection have the same structure, but there is no name given to the structure. In this case the property list defines the properties for the target. When the target structure should contain all properties from the source an * can be passed instead of the names for all properties of the source.

```
persons.select{*, age = (Date()-birth).Year()}
```

This expression will return all person properties plus the age calculated from the birth. This expression corresponds to the standard OQL expression

```
select
    p, age = ( Date()-p.birth).Year() )
from Persons p
```

The type operator creates internal unnamed types, which may act as input to generic operations and expressions, but which cannot have predefined operations.

Order

There are different ways ordering the result of an element or sub-path. The order can be defined by means of expressions or attribute references, but also by means of sort keys, when the system supports sort keys.

Now, we can define the order operation as generic operation within a property path as:

```
order_elmt     := order_by( order_def )
order_def      := key_name [ (asc | desc) ] |
                  sort_field [ , sort_field ]
sort_field     := sort_expr [ (asc | desc) ]
sort_expr      := valid OQL expression
```

Order_by is a generic operation, which returns the same collection, just re-ordered, i.e. type and extent of the collection remain unchanged.

Grouping instances

Grouping operations in OQL are expressed as **`group by`** clause. Since the object model supports ad hoc groupings by definition and may support classifications as conceptual groupings [3], grouping operations are required only for defining groups which are not defined in the data model.

The OQL standard [1] allows grouping by value but also by derived values. Thus, a grouping may define the extension of the output implicitly or explicitly.

```
group_elmt     := group_by( group_def )
group_def      := prop_expr | val_spec [ ,val_spec ]
prop_expr      := prop_name | expression
val_spec       := val_name [ : condition ]
condition      := valid OSI expression with type boolean
```

A grouping definition is based either on an attribute value (each attribute value creates a separate group) or on an explicit list of grouping values. Defining an explicit value list for a grouping requires associated Boolean expressions for associating an instance with a grouping value. The instance is associated with the first value, for which the expression returns true.

Defining the last value without expression will associate all instances, which do not any of the previous expressions, with the last (default) value. When no default value is defined, instances not matching any expression are skipped.

The extension of the result of a grouping operation is determined by the attribute values (when referring to an attribute) or by the values in the list.

The type of instances in a grouping result is constructed from the grouping attribute (or string in case of a grouping list) and a relationship **`partition`**.

```
persons.group_by(
    low: income < 1000,
    medium: income >= 1000 and income < 5000,
    high )
```

The example above creates an internal result type as:

```
struct(
    attribute value string,
    relationship bag<Person> partition )
```

When grouping instances by an attribute (which might be a complex attribute as well) the implicitly generated result structure of the operation contains the attribute instead of the standard attribute **`value`**.

```
persons.group_by( age )
```

Results in:

```
struct(
    attribute age long,
    relationship bag<Person> partition )
```

The type for the attribute is taken over from the original attribute definition in the input collection.

Aggregation operations

OQL and OSI allow referring to instance or object operations within a property path. Moreover, it provides some general aggregation expressions as **`sum`** or **`count`**. A more general solution, however, would be to differ between instance and aggregation operations. While an instance operation applies to any instance in a collection, an aggregation operation applies to the whole collection and creates one single instance for each collection.

OSI allows defining an expression as instance or collection expression. Also build-in functions are precisely defined as instance or collection operations, but in some cases, user-defined methods will not provide this information. In such cases, we need a language extension, which allows distinguishing between instance and collection operations.

OSI differs between collection and instance operations by using the () operator. Using the () operator will path each instance of the collection to the following operation. Not using the () operator will pass the collection to the operation. The operator always applies to the nearest preceding element or sub-path.

```
Persons().children.sum(income)
[persons.children]->sum(children.income)
```

The first expression returns an income value for each person (sum of income of all children). The second query returns one income value, which contains the sum of incomes of all children of all persons.

Combining the collection operator with the type operator allows defining a number of aggregation operations for one collection.

```
persons.children->select(
    tot_inc: sum(income),
    avg_inc: avg(income),
    number_of_children: count
)
```

In this case, each expression in the type operation should be a valid collection operation. Any operation, which refers to an instance expression, will return the value evaluated for the last instance passed to the type operator.

Other operations

Other operations known in OQL as **`intersect`**, **`union`**, **`difference`**, can be easily expressed as operations and can be added to a path whenever required, as well as different types of join operations or user defined operations. Note, that these operations are set operations, but not aggregation operations. The semantics of those operations, however, need some more explanation.

4 Summary

The main difference with the path approach is, that is turns the clauses of an OQL select statement into generic operations. This allows combining any sequence of operations in a query. Since each operation provides the same features, OSI path expressions are at least as powerful as OQL statements.

On the other hand, it can easily be shown, that there is an OQL statement for any sequence of operations, which can be defined with a path. Thus, we can say, that OQL and OSI path expressions are equivalent. The missing feature of using user-defined collection operations must be considered as temporary and not as a significant difference between the two approaches.

The advantage of path expressions is mainly readability and flexibility. Since operations become independent it is a matter of knowing the semantics of each operation, which can be combined freely.

When considering path expressions as sequence of **`from`**, **`select`**, **`where`**, **`order_by`** and **`group_by`** operations combined with other generic or user-defined operations, it becomes possible combining the advantage of the well known SQL syntax with the short and precise way of expressing queries in paths.

5 References

[1] ODMG: *The Object Data Standard ODMG 3.0,* Academic Press, 2000

[2] SQL

[3] R. Karge, Unified Database Theory, run-software, 2003