Implementation of bi-directional association in ODRA system with use of reverse pointers

This work is supported by the EC 6-th FP, Project VIDE, IST 033606 STP

Table of content

1. Introduction 3

2. Reverse pointers – the implementation of bi-directional association in OODBMS ODRA........ 4

3. Reverse pointers implementation 7

3.1 Reverse pointers support in the object store 7

3.2 Reverse pointers support in the ODRA query execution engine 8

3.2.1 Creating reverse pointers 9

3.2.2 Updating reverse pointers 9

4. Summary 10

1. Introduction

One of the important features of UML static model are the associations between classes. As class defines object (a class instances), the association defines links (an association instances) connecting class instances. Or more precisely: "An "association" in UML is defined as a kind of relationship between classes, which represents the semantic relationship between two or more classes that involves connections (links) among their instances". The association represents strong (structural) relationship between the class instances. Description of the association involves a name, multiplicity, roles, and direction.

Consider the example of an association named **worksIn** that connects class Department and class Employee shown on Figure 1.

Figure 1 Sample binary association

The association defines that each Employee class instance is connected with obligatory link *worksIn* with a Department class instance. From the opposite side each Department class instance is optionally connected with arbitrary number of Employee class instances. The association multiplicity is able to define if we deal with obligatory (minimal cardinality greater than 0) or optional (minimal cardinality equals 0) connection. The second important information is the direction of the association that can bi uni- or bi-directional. The direction property defines the sides of the associations that "knows" about the opposite one. This knowledge means that one can be able to navigate from the one instance to another with use of link defined by the association. It also defines the range of the association implementation and maintenance. Bi-directional association requires implementation expenditure for both classes. Moreover, if we think about operations like creating or updating an association instance, bi-directional one requires an action affecting both association ends. For the MDA tools like VIDE, generating code from visual models it is very important to have an execution engine that does manage the associations automatically. This can greatly simplified the process of mapping the visual model onto the execution one. In the following section such a solution, implemented in the OODBMS ODRA is presented.

2. Reverse pointers – the implementation of bi-directional association in OODBMS ODRA

The implementation of the association in the object database and the query language has to take broader than only the "conceptual" view on this concept. In ODRA, Uni-directional associations are mapped to pointer objects in the appropriate class. For the purpose of bi-directional associations ODRA introduces special kind of pointer objects called reverse pointers that allow for automatic update of pointers pairs in both classes participating in the given association. The multiplicities are mapped directly to ODRA object cardinalities.

The definition of the model from Figure 1 in ODRA SBQL syntax has the following form:

```
class EmployeeC {
      instance Employee : {
            name:string;
            salary:integer;
            workplace:ref DepartmentC reverse employs; }
/*methods definition*/
}
class DepartmentC {
      instance Department : {
            name:string;
            employs:ref EmployeeC[0..*] reverse workplace; }
/*methods definition*/
}
//class variable declarations
Employee:EmployeeC[0..*];
Department:DepartmentC[0..*];
```

The above code defines classes *EmployeeC* and *DepartmentC.* The class definition consists of two parts: instance definition and methods definition. The instance definition defines the class instance type and (optionally) instance name. The method definition (omitted in the above example) specify

behavior of class instances. It is worth mention that from the database point of view the definition of the class is not enough because a class does not, itself, define the data structure. This is obvious for the object-oriented programming languages where the class definition is not equals to definition of a variable of given class type. Thus, the last part of the example declaration defines two class variables *Employee* and *Department*. In contrary to the object-oriented programming languages the variable name usually represents a bag of objects and the single object is only a special case.

The implementation of bi-directional association *worksIn* is achieved with use of reverse pointers (in the sample declaration the association roles where used as pointer names). Each reverse pointer has its counterpart in the target object. The definition of the pointers cardinality implements the multiplicities of association ends (no cardinality declaration means by default [1..1]).

The idea of reverse pointers in ODRA assumes that the initialization and maintenance of defined links is automatic, i.e. there is no special requirements for the programmer to initialize both sides of the associations. For example if one create an Employee class instance and initialize the *workplace* pointer there is no additional action required to create reverse twin pointer in the Department object. Below the sample creation of a Department and Employee object is presented below:

```
//S1. create ‘IT’ department object

create Department(“IT” as name);


// S2. create ‘PR’ department

create Department(“PR” as name);


// S3. create employee ‘Doe’ and set him as IT department worker

create Employee(“Doe” as name, 2000 as salary, (ref Department where
name=”IT”) as workplace);


//S4. create employee ‘Poe’ and set him as IT department worker

create Employee(“Poe” as name, 2000 as salary, (ref Department where
name=”IT”) as workplace);
```

After executing the above statements assumed database state is shown on Figure 2.

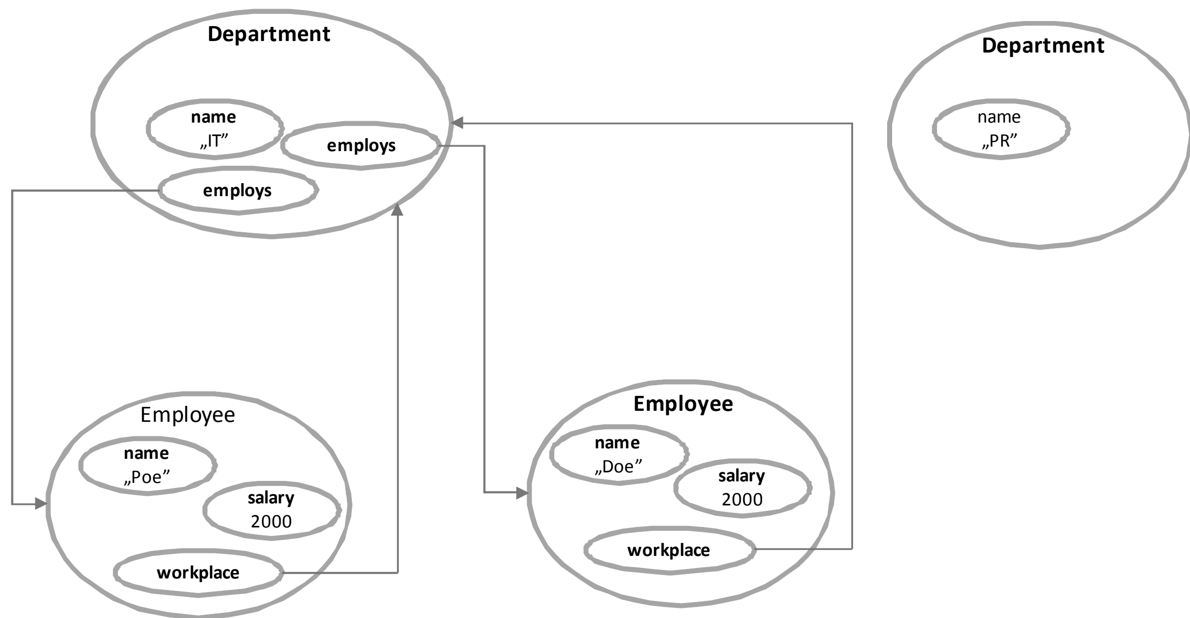

Figure 2 Sample database state

Creation of Employee objects with *workplace* pointers caused automatic creation of reverse pointers (*employs*) in the target Department object. Further management is also done automatically. For example moving Employee "Doe" from "IT" to "PR" department is performed on query level by updating workplace pointer value, cause deletion of corresponding reverse pointer in "IT" Department and creation of a new one in object representing "PR" department. The required query is presented below and new database state is depicted on Figure 3.

```
//S5. move 'Poe' to 'PR' department
(Employee where name="Doe").workplace := ref (Department where name="PR");
```

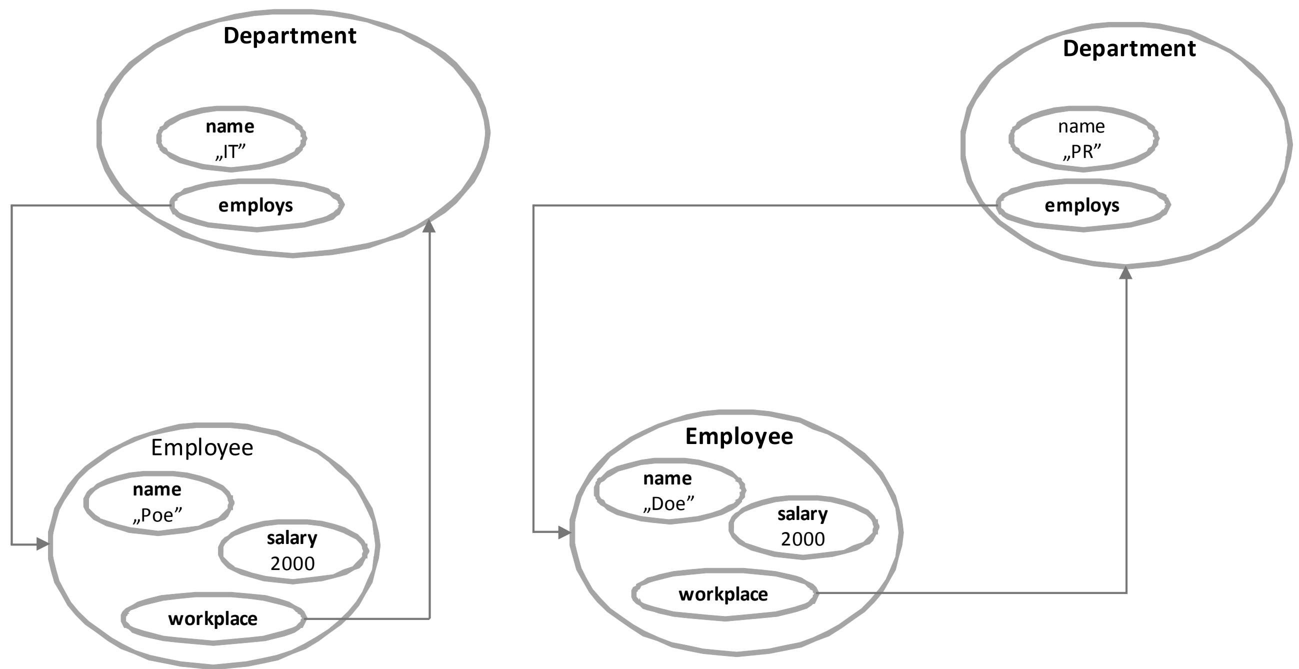

Figure 3 Database state after moving 'Doe' to 'PR' department

The declared cardinality can constraint operations that can be performed on the reverse pointer. Most of them can be signaled at the query compilation level by the type-checker. For example the deletion of employs pointer from inside of the department object is not possible because it force the deletion of corresponding workplace pointer in the Employee object which cardinality is [1..1] (currently ODRA does not support mechanism that could allows for deletion of the entire Employee object in response to deleting employs pointer). ODRA compile forbids the execution of such a query. This constraint leads to another one. It is not possible to delete Department object form the database if it posses any employs pointer (although this situation has to checked during runtime because of minimal cardinality equals 0).

3. Reverse pointers implementation

The implementation of reverse pointers involves following elements of the system: object store, and query compilation engine (supported by the information stored in the meta-base) and query execution engine.

3.1 Reverse pointers support in the object store

In the ODRA object store an object is a triple <i,n,v> where i denotes unique internal identifier, n denotes an external object name and v denotes the object value. On the object store level objects are divided into three types (depending on the value):

- Simple objects – the value of the object belong to one of the following type: integer, real, string or date.
- Complex objects – the value of the object is a collection of (nested) sub-objects of arbitral type. ODRA introduces a special purpose complex object called aggregate object that is used to aggregate object with the same name and type on the given nesting level. An aggregate object is often referred as collection representation.
- Reference objects – the value of the is a identifier of the other object in the store.

Each reference object is potentially able to have a reverse reference. It means that from the ODRA object store point of view an ordinal pointer object can become a reverse pointer without changing its identity (and vice versa). The store interface allows for the following operations to be performed on the pointer objects (that are connected with reverse references).

1. boolean isReverseReferenceObject() – return true if the object has reverse reference
2. setReversePointer(OID target) – set reverse reference value equals to target pointer object identifier (OID).
3. OID getReversePointer() – returns reverse pointer object reference

Figure 4 shows the pointers object connected with reverse references.

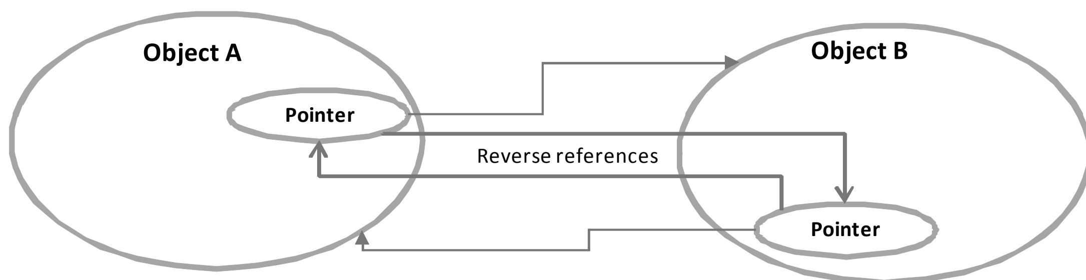

Figure 4 Reverse references

If the reverse pointer is being set to a new value the existing reverse counterpart is automatically deleted and the reverse reference is invalidated, although it is not set to a new value. The reverse pointer become an "ordinary" one. This process of setting new reverse reference must be supported by the query engine as described in the following section.

3.2 Reverse pointers support in the ODRA query execution engine

Because the store level introduces only base reverse pointer management it is supported by the query execution engine.

ODRA execution engine supports the process of automatic reverse pointers management during creation and update of reverse references:

3.2.1 Creating reverse pointers

Rely upon the meta-information code-generator (store at the meta-base level) generates byte-code responsible for creating twin pointers and connect them with reverse references. For the sample create query (presented below), the process consists of the following steps:

```
// S3. create employee 'Doe' and set him as IT department worker

create Employee("Doe" as name, 2000 as salary, (ref Department where name="IT") as workplace);
```

1. The *workplace* pointer object is created inside new Employee object and initialized with the result of the sub-query `ref Department where name="IT"` (which is the "IT" department object reference). The process is the same as for ordinal pointers.

2. The *employs* pointer object is created inside the "IT" department object and initialized with new Employee object reference.

3. New *workplace* and *employs* object are inter-connected with reverse references. From now on they are reverse pointers.

3.2.2 Updating reverse pointers

As in case of creating reverse pointer the process if updating requires support from the execution engine. For the example update query (presented below) the generated byte-code perform the following tasks:

```
//S5. move 'Poe' to 'PR' department

(Employee where name="Doe").workplace := ref (Department where name="PR");
```

1. The *workplace* pointer is set to new value (reference to "PR" department object reference). Because the pointer pointed on the "IT" department object and in "IT" department object corresponding reverse pointer exists it will be automatically deleted. This deletion is possible only if the reverse pointer minimal cardinality will be consistent after this operation (violation to this constraint can cause runtime- or compile-time error).

2. Inside "PR" department object new *employs* pointer object is created and its value is set to employee object reference (a parent object of *workplace* pointer).

3. Updated *workplace* and new *employs* objects are inter-connected with reverse references.

4. Summary

The ODRA reverse pointers give the ability to implement UML bi-directional associations with the automatic update of both association sides. The association ends are represented by the reverse pointers that extends ordinal pointers with reverse references interconnecting pointers itself. The multiplicities of the association ends are implemented as reverse pointers cardinalities. The functionality of the object store supported by the execution engine manages the process of automatic update of one association end in response to an update of an opposite one.