ODRA Variables – an overview

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

Introduction

ODRA is an object-oriented database management system that implements a powerful query/programming language SBQL (Stack Based Query Language). SBQL code is used write programs as well as declaring and creating database objects that includes: modules (ODRA compilation units and object containers); variables (objects representing data), procedures, classes, views, etc.

Code vs. database structure

The structure of the ODRA source code is related to the structure of the objects in the database. It is worth mention that all the declaration elements are mapped to the ODRA objects in particular: modules, classes, views, procedures, etc.

ODRA Modules

The root for the declaration is a module that represents a compilation unit as well as an object container in the database. From the point of view of the code each module can contain declaration of other elements (variables, procedures, classes, types, views, etc.). From the database point of view module contains module data and metadata used during compile-time processes (type checking, optimization, etc.).

Modules interior

ODRA objects properties

As described before, module represents container for the objects as well as the code compilation unit. In programming languages where objects are volatile and constraint to the executing application making objects persistent or sharing them between applications requires special mechanisms and treatment. The process of reuse is based on the program code not the objects itself. In object-oriented database environments like ODRA reuse is base on shared objects representing programs (procedures, methods, classes) as well as data (variables). The objects availability and persistence property is orthogonal to programming constructs and does not require additional explicit treatment.

In the context of database systems the main property that is pulled out to the front is persistence. It results from the historical distinction of databases (with mechanisms to persist data) and programs executing instruction and using data from database. ODRA system takes a bit different point of view. The main property is availability thus the objects are divided into:

1. Shared – objects that are available to many users. They are stored in the space that ensures common availability (according to security rules) and persistence (between user sessions). Thus shared objects can be perceived as persistent.

2. Private - objects that are available to the owner. They are stored in the volatile space and are not shared between user sessions. The examples of such a space are procedure local environment and user session environment.

Variable declaration

To create and manage objects in the database the ODRA system requires the information about them. This knowledge has to be introduced with use of a variable declaration. At first, as in case of popular programming languages, variable declaration introduces the name and type of declared objects. The variable name represents the name of defined objects and type informs the type system about the structure of them. The basic syntax is following:

```
<name>:<typename>;
```

The variable name can be any proper identifier. The variable type name can be the name of one of the basic or user defined types. It can also be a name of the other variable (when a variable declaration defines pointer objects) or a class (when a variable defines class instances).

Defining objects collections

Because ODRA is a database system it usually works on collection of objects the basic declaration is extended with the information about the size of the object collection represented by the variable. This information can be introduced with use of cardinality. The cardinality describes minimal and maximal number of objects in collection defined by the variable. If the minimal cardinality is equals 0 the collection can be empty, if is greater than 0, it obligatory requires at least so many objects as the minimal cardinality value. The maximal cardinality (which must be greater or equals 1) defines the upper limit of objects in the collection being declared. It is also possible to define (logically) unlimited size of a collection. The basic variable declaration syntax extended with cardinality definition is presented below:

```
<name>:<typename>[<minimal cardinality>..<maximal cardinality>];
```

The basic variable declaration (without information about cardinality) implicitly defines a collection with lower and upper limit set to 1 (thus is equivalent to explicit [1..1] form).

Below some examples of variable declaration are presented:

```
counter:integer[1..1]; //declaration of a single integer object named
```

```
                        //counter

counter:integer; //same as above

address:string[1..10];//declaration of a collection of string type objects
                      //named 'address'. The collection contains at least
                      //one element. The upper limit is equals 10

Car: record {
             model:string;
             manufacturer:string;
             production_date:date;
            } [0..*]; //declaration of a collection of objects named 'Car'
                      // that type is defined by the record consisting of
                     // three fields. The collection can be empty and has
                     //no upper limit
```

Defining objects availability

As benefits of database management system ODRA objects defined by the variable declaration are available to many users, and can be accessed concurrently by many user sessions. Mainly the ODRA objects availability is divided into two levels: shared and private. A shared object is available (according to the security constraint) to all users and it durability exceeds the user session boundary. A private object is available only to one user within the single session. The private objects are further divided into session and local. Session object life line spreads from the moment of creation to the end of user session. Local object is limited to the block of code it was declared in.

The availability of objects is (with one exception) defined implicitly through the place of a variable declaration. If a variable declaration is placed directly in the module definition, the objects being defined with use of the declaration are shared by default. If the variable is declared within the procedure definition body, the defined objects are private and local to the given code block. Only definition of session objects requires syntactic distinction. The declaration of a session variable can only be placed directly in the module definition and has to be preceded with a **session** keyword.

Types of ODRA objects

Up to now it has been said that the declaration of a variable in the ODRA directly defines an object in the store. Each object has a unique identifier and a (non unique) name. Now it is time to take a closer

look on the object value. An ODRA objects are divided into three types according to a value kind: simple, complex and pointer:

- simple – an object value is one of the basic ODRA types. For a variable declaration it means that the declared type name is a name of a basic type (see: **Błąd! Nie można odnaleźć źródła odwołania.**), e.g.:

```
i:integer;
```

- complex – an object value is a set of sub-objects. For a variable declaration it means that the declared type is an unnamed record or a name of user defined complex type or a name of a class, e.g.:

```
Person : record { name:string;
                  age:integer; }[0..*];
```

- pointer – an object value is an identifier of the other object in the store. For a variable declaration it means that the declared type name is the name of the other variable precede with **ref** keyword, e.g. the declaration of a variable pointing at a 'Person' variable (defined previously) can be defined as:

```
adult : ref Person[0..*];
```

Pointer object in ODRA has properties that require additional explanation to be introduced in the next section.

Creating objects in ODRA

Declaring a variable in languages like Java does not mean that a referenced object is automatically created. The variable has to be set to point on either newly created object or already existing one (previously referenced by other variable). In ODRA the process is slightly different. As was pointed before, variable represents a name for objects in the object store. The variable cardinality defines the number of objects that can exists it the store and are referred with a variable name. Thus the process of creating objects is divided into two stages: compile/initialization time and runtime.

Compile/initialization time automatic object creation

Compile/initialization time object creation depends on the declared variable cardinality. Variable minimal cardinality value greater than 0 informs ODRA that the collection cannot be empty. It forces the system to automatically create required minimal number of objects. Additionally there will be no possibility to delete all the objects from the collection (because minimal cardinality requirement is

enforced by the system). The exact time while such “default” objects will be created depends on the objects availability property:

- If the object is shared (i.e. declared directly in the module without *session* keyword specified) it will be automatically created during module compilation process.
- If the object is private it will be automatically created at the moment of:
    - User session initialization - for session objects.
    - Local store initialization - for local objects. It can be the moment of procedure call or the moment when control enters local block where a variable was declared.

Almost all object types can be automatically created by the system. One exception is pointer objects that cannot be automatically created. It results from the assumption that there in ODRA all existing pointers have to refer proper objects and there is no “dangling pointers” (NULL value does not exist in ODRA). As for ordinal (not pointer) objects the system can set the default value, for pointer objects it has no chance to do that[1]. This could be changed with the extension for the variable declaration that gives user a chance to set some user defined initialization values. Unfortunately the current version of the system gives very limited abilities to that. The default object initialization is limited only to local variable declaration with cardinality set to [1..1]. An object that is automatically created according to such a declaration can be user-initialized. What more if the declaration represents pointer object the initialization can be postponed. The only constraint is that it must be set to proper object reference before first reading (dereference).

To sum up, the process of automatic object creation is only an ODRA functionality supplement designed to fulfill cardinality requirements. The main creation process is played at the programs execution in response to user orders.

User object creation at runtime

In ODRA we have to look for the object creating process from the data store perspective. For better understand of this point of view it would be more suitable to think about this process as about the way a given object can appear at the chosen level of the object store. By the term level we understand the particular object interior. It can be the root level which in ODRA means module object, it can be a sub-object of a module root object and a sub-object o this sub-object, as the level

[1] It will be changed it the further versions of ODRA system

of nesting is unlimited and constraint only by the declaration. There are following methods used to put an object at the given nesting level:

- Explicit object creation with use of *create* operator. The environment where the newly created objects appear depends on the execution context.
- Inserting an object with use of one of insert operators' family. Depending on the operator type:
    - the existing object (or objects) can be moved from one level in the object store to another – it can be done with use of insert operator
    - the existing object (or objects) can be moved from one level in the object store to another but the name of object/objects change – this can be done with use of "rename and insert" operator
- Creating an object in the selected environment. This operator is called "create and insert" and semantically is equivalent to creating new object and immediate inserting it into a given environment.

Each of above methods cause the object appears in the new place of an object store. All those methods are controlled by the compiler with use of variable declarations. In other words, an object can appear at the given store level if it was previously declared there and its structure fits the declaration. Additionally, declaration cardinality must allows for placing new objects (it cannot be [1..1]). In the following sections the details of each method will be explained.

Create operator use

Create operator allows for explicit creation of a new object in the store. As an argument it requires the name and the value of the object being created. From the point of a SBQL language <value> is a result of a sub-query. For simple object the value must be typologically compatible with the declared variable type. For complex object the value means, at least names and default values for all the sub-objects that are required from the point of view of declared cardinality. For pointer object the value means a reference to an object that type is compatible with declared pointer type.

Newly created object property like: availability (shared or private) as well as the exact place for the object is determined by the context the create instruction is executed against. The context means the variable declaration that is bounded with a <name> parameter of a create instruction in current execution environment. Examples in the following sections brings closer look at this aspect.

Create operator belongs to a group of set operators. It means that they allow the sub-query calculating value being parameter returns a bag of result. In other words, create operator can create zero or more objects. The number depends on the multiplicity of the <value>.

Creating simple objects

Creating simple objects is straightforward. The name of the object being created should be followed with the object value of type compatible with the context declaration. Consider the following code:

```
module testCreateSimpleObjects {
   varA:string;
   varB:integer[0..1];
   session varC:boolean[0..*];

  createSomeObjects() {
     varA:string[0..*];
     varD:real[0..1];

     create varA("test");
     create varB(12 + 3);
     create varC(true union true union 1 <> 1);
     create varD(12.4);
  }
}
```

The example module *testCreateSimpleObjects* defines two shared variables (namely: 'varA' and 'varB') and one private session variable varC. Additionally procedure *createSomeObjects* declares two local variables 'varA' and 'varD'. According to scoping rules local variable 'varA' obscure shared variable with the same name defined in the module. Instructions in the *createSomeObjects* procedure create following objects:

- `create varA("test");` - creates local object named 'varA' in the procedure local store. The value of the object is string "test".
- `create varB(12 + 3);` - creates shared object named 'varB' inside the *testCreateSimpleObjects* module. The value of the object is a result of 12+3 query.
- `create varC(true union true union 1 <> 1);` - creates session objects named 'varC' in the *testCreateSimpleObjects* module private store. The number of objects and its values are determined by the result of a parameter query (`true union true union 1`

<> 1). Because the query returns a three elements bag (bag{true, true, false}) three objects will be created.

- `create varD(12.4);` - creates local object named 'varD' with real value equals 12.5.

Calling the *createSomeObjects* procedure from within the user session cause creation of objects according to the procedure code definition.

Creating complex objects

To create complex object the parameter query has to return a result that reassemble the structure of a complex object fields. Consider the following code:

```
module testCreateComplexObjects {
   type AddressType is record {
                          city:string;
                          street:string;
                          zip:string[0..1]; }

   type PersonType is record {
                          fName:string;
                          lName:string;
                          age:integer;
                          address:AddressType[1..*];
   }
  Person:PersonType[0..*];


}
```

Definition of module *testCreateComplexObjects* introduces named record types: 'AddressType' and 'PersonType'.' AddressType' consists of two required (namely 'city' and 'street') and one optional field ('zip'). 'PersonType' includes three simple type fields ('fName', 'lName' and 'age') and one field of a complex 'AddressType' type that is also declared as collection with at least one element. Additionally it declares variable representing a collection of 'PersonType' objects named 'Person'. To create a 'Person' object (or object) it's necessary to define a query that result will be structurally compatible to a 'PersonType'. Thus the query should result in the structure consisting of binders with names equals to names of the 'PersonType' record and values of types compatible to declared types for individual fields. The query creating sample person object is presented below:

```
create Person("John" as fName, "Smith" as lName, 26 as age, ("New York" as
city, "5'th avenue" as street) as address);
```

The definition of a ‘PersonType’ declares that address field is a collection that can have more than one object. Subsequent address objects can be added with use of an insert operator or (what will be shown below) created with a Person object. Below example creates second ‘Person’ object, this time with two address sub-objects:

```
create Person("Michael" as fName, "Pears" as lName, 34 as age, (("New York" as city, "5'th avenue" as street) union ("Warsaw" as city, "Koszykowa" as street, "02-008" as zip)) groupas address);
```

Notice that sub-query defining ‘address’ objects:

```
 ((("New York" as city, "5'th avenue" as street) union ("Warsaw" as city, "Koszykowa" as street, "02-008" as zip)) groupas address)
```

uses *groupas* operator to avoid creating two Person object instead of one with two address sub-objects.

Creating pointer objects

The process of creating pointer objects is similar to the one described for simple objects. The difference lays in the object value. For pointer objects the value must be another object reference. Variable defining pointer has to declare type that name represents another variable name. Assume that a sample model from Creating complex objects section is extended with a declaration of a pointer variable:

```
module testCreatePointerObjects {
/*   AddressType declaration */

/* PersonType declaration */
   2
  Person:PersonType[0..*];
  adultPerson: ref Person[0..*];
}
```

The ’adultPerson’ variable defines a collection of pointer objects that value has to be reference to an objects defined by a ‘Person’ variable. Assuming that a collection of ‘Person’ object is not empty we can create an adult pointers by selecting those Persons that value of an ‘age’ sub-object is grater or equal 18:

```
create adultPerson( ref (Person where age >= 18) );
```

The *ref* operator is required here to avoid automatic dereference of a result being a parameter for create operator. The number of created pointer objects depends directly on the number of results returned by a parameter query.

Insert operator use

Previous sections described the process of explicit object creation with use of a *create* operator as a way to introduce object into given level of ODRA object store. This section concentrates on the second kind of performing this task – the insert operator usage. An insert operator occurs in ODRA in two variations developed for slightly different situations and needs. This diversity has its source in the ODRA type system and in the type-checking rules the program code must conform to. The main property that must be taken into consideration is the requirement saying that an object can appear at the given level of an object sore only if is declared for that place. The declaration can be expressed as a variable directly (in module root or in procedure) or as a record type field declaring interior of a complex object.

Insert operator - variable equivalence based objects insertion

First insert operator (denotes as ':<' sign) move the object from one place in the store to another without any change. The syntax is following:

```
<objref>[1..1] :< <objref>[0..*]
```

Insert operator L-value is a query returning single object reference. R-value is a query returning a bag of references of objects that are type and name compatible with a declaration of objects inside the L-value referenced object. In other words insert operator assumes that inside the environment referenced by L-value exists a variable that name is equal and the type is compatible to the name and type of moving object. Additionally the variable has to declare appropriate cardinality that allows for changing the size of an object collection. It means that declared cardinality range must include r-value cardinality range.

Consider the following sample code:

```
module testInsertObjects {

   type EmployeeType is record {
         fName:string;
         lName:string;
         salary:integer;
   }

   Employee:EmployeeType[0..*];

 type DepartmentType is record {
         name:string;
```

```
        employs:ref Employee[0..*];
 }
 Department:DepartmentType[0..*];

 addEmployee(fName:string; lName:string; dName:string) {
   employs: ref Employee;
   employs := create Employee(fName as fName, lName as lName; 2000 as
                                                              salary);
   (Department where name = dName) :< employs;
 }
}
```

The above module definition introduces two user defined types: one for an employee objects (EmployeeType) and second for Departments (DepertmentType). 'DepartmentType' definition declares 'employs' field defining pointer object collection for storing references to Employee objects (notice that declaration of a pointer object requires 'Employee' variable not an 'EmployeeType' type). The usage of an insert operator is shown in the code of *addEmployee* procedure. The procedure takes three parameters: first and last name of an employee to be added to the system and the name of the department – new employee work place. The procedure body declares single local pointer object 'employs' and initialize it with a reference to newly created 'Employee' object (notice that local reference store a reference of shared (global) object). The last procedure instruction use the insert operator to move 'employs' pointer object from local store to the interior of Department object referenced by a result of a left operand query. The compiler allows for this operation because source and target object declarations are equivalent.

"Rename and insert" operator

Default insert operator requires the existence of equivalent declaration in source and target environments. This is fairly strong requirements and can, in some situations, makes the coding difficult. The "rename and insert operator" still requires type compatibility but lessen the constraints by allowing the source and target objects declaration possessing different names. The syntax is similar to an ordinal insert operator but right operand query has to return single binder:

```
<objref>[1..1] :< <binder>[1..1]
```

The binder name refers to a name of the inserted objects in the target environment and the binder value is a bag of references to objects being inserted (which names are insignificant). The insert process during moving objects from source to target, change their names to fit the required name in the target environment. Below sample code is a modification of an *addEmployee* procedure from previous example that use rename and insert instead of ordinary insert operation:

```
addEmployee(fName:string; lName:string; dName:string) {
   employee: ref Employee;
   employee := create Employee(fName as fName, lName as lName; 2000 as
                                                              salary);
   (Department where name = dName) :< employee groupas employs;
 }
```

The local pointer object name is different from the one needed inside the department object. Thus right operand of a query inserting pointer into 'Department' object returns a binder created with use of *groupas* operator. The usage of *groupas* operator assure that the result has [1..1] cardinality as required in the operator definition.

"Create and insert" operator use

The "create and insert" operator does not move the existing object from one environment to the other. Its semantics is based on the creation of a new object and inserting it into a given (as operator parameter) environment. This is de facto a create operator. The difference from an original create operator is connected with the way the target environment is exposed to the creation process. The ordinal create operator has the only argument – specification of created object, and the place for the object is calculated basing on the usage context (see description in section "Create operator use"). The "create and insert" operator has two arguments. First, similarly to insert operator, is the query that result is a reference to the environment in which the object should be placed. Second, is the same as in the create operator case – the new object specification:

```
<objref>[1..1] :<< <name>(<value>[0..*])
```

This operator introduces the bridge between create and insert operators and can be used when it is a need to create a new object only to immediately insert it into other place. Instead of declaring an object in the execution environment, create it and, then insert it into given object, the 'create and insert' operator can be used. As the example of use of this operator the model defined in section: "Creating complex objects" will be used:

First a new Person object will be created with use of *create* operator:

```
create ("John" as fName, "Smith" as lName, 26 as age, ("New York" as city,
"5'th avenue" as street) as address);
```

Next new address object will be introduced into the 'Person' object (the 'PersonType' defines field 'address' with [1..*] cardinality):

```
 (Person where lName = "Smith") :<< address("Warsaw" as city, "Koszykowa"
as street);
```

Finally because ‘address’ posses optional field named ‘zip’ that was not initially created, the following query add it for the ‘address’ in “Warsaw”:

```
 (Person where lName = "Smith").(address where city = "Warsaw") :<<
zip("02-008");
```

Summary

ODRA system introduces fully functional programming environment. One of its features is a possibility to define objects, representing data, with use of a variable declaration. In contrary to the meaning of the variable concept in popular programming languages ODRA introduce new sense to it. The variable expresses the place in the object store, identified by a variable name, where objects of particular type are stored. The permissible number of objects is expressed with use of a cardinality concept. The place of a variable definition expresses the availability of defined objects.