Role-Specific Operators in the SBQL Query Language*

Andrzej Jodłowski[1], Jacek Płodzień[1,2], Ewa Stemposz[1,3], and Kazimierz Subieta[1,3]

[1] Institute of Computer Science PAS, Warsaw, Poland
[2] Warsaw School of Economics, Warsaw, Poland
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland
{andrzejj, jpl, ewag, subieta}@ipipan.waw.pl

Abstract. In the paper we present a new version of the SBQL query language which has been extended to cover the concept of dynamic object roles in an object-oriented data model. In the main part of the paper we discuss special operators which are necessary/useful for managing objects and roles. Those operators enable one for instance to test presence of a given role within an object/role, to retrieve direct roles that are currently present within an object/role, etc. The discussion is illustrated with examples and figures.

1 Introduction

Today's object-oriented data models offer several concepts that make them a very useful tool in the fields of databases and information systems. One of such concepts is the concept of dynamic object roles which expresses the idea that an object (a so-called "owner-object" or "owner") can be associated with other objects (so-called "role-objects" or "roles") modeling its roles; moreover, an object can acquire and abandon roles dynamically (see for example [7]). Roles are treated as objects with some additional special features such as: a role cannot exist without its owner; deleting an owner implies deleting all of its roles; roles can exist simultaneously and independently. While being an object, a role can have its own additional attributes, behavior, etc.

A data model with dynamic object roles involves two kinds of inheritance: static and dynamic. Static inheritance is the inheritance between classes in the traditional sense, where the properties of a class are imported by its subclasses at compile time. The mechanism of dynamic inheritance is similar with the following difference: it concerns objects whose values are imported by their roles at run time.

Dynamic roles are believed to make an important paradigm for object-oriented databases and their query languages. Therefore we have decided to introduce and develop the concept in our object-oriented data model (called the *Stack-Based*

* This work was partially supported by the European Commission project ICONS; project no. IST-2001-32429.

Approach; SBA) and a special query language (called SBQL) for object-oriented database systems based on SBA. Some ideas are presented in [1–4]. At the beginning of the paper we discuss a general framework of our proposal[1], and in the main part we focus on special operators for managing objects and roles. Those operators can be applied to test presence of given roles within an object/role, to retrieve direct roles that are currently present within an object/role, etc.

The remainder of the paper is structured as follows. Section 2 covers general ideas of SBQL with regard to the concept of dynamic object roles. In Section 3 we discuss the role-specific operators that we define to manage objects and roles through our query language. Section 4 summarizes the paper.

2 SBQL with Dynamic Object Roles – General Ideas

In our discussion we assume a problem domain in which *Person* objects can possess *Employee* and/or *Student* roles, and *Employee* roles can possess *Designer* roles. Fig. 1 presents an example object store constructed in accordance with this problem domain and with our data model involving dynamic roles:

- Each store object is constructed of the following elements: an identifier, a name, and a value (which can be a literal, a link represented by an object identifier, or a set of objects). For example, one of the objects in the store is named *Person*; the object's identifier is i_7, and its value is two objects with identifiers i_8 and i_9.
- There are four objects storing the invariant properties of *Person*, *Employee*, *Designer* and *Student* objects; they are called *PersonClass*, *EmployeeClass*, *DesignerClass* and *StudentClass*, respectively.
- Each object has access to the invariant properties of its class. This is denoted as a thick arrow. For example, an *Employee* role is connected to its *EmployeeClass* owner.
- Role-objects dynamically inherit the properties of their owner-objects. This is denoted as a line with a white-and-black diamond end; e.g., *Employee* roles dynamically inherit from their *Person* owners.

2.1 Environment Stack

In programming languages a special data structure called an *environment stack* (ES) is responsible for scope control and name binding. A new section of volatile objects is pushed onto the stack when a new procedure/block is started, and the section is popped when the procedure/block is terminated (for a procedure such a section contains volatile variables (objects) declared within this procedure along with its actual parameters, return address, and possibly other data). Binding names follows the "search from the top" rule, that is, the last added section

[1] Nevertheless, due to space limit we are unable to present SBA and SBQL in detail. Therefore in this paper we assume at least a general familiarity with SBA and SBQL; the reader is referred to [2, 5].

is the first one visited during the binding, and due to so-called *static scoping* objects from some sections remain invisible for the binding.

SBA uses ES – the general idea of the stack-based semantics for object query languages is that some query operators (called *non-algebraic*) act on ES in a similar way as invocations of program blocks. For instance, in the query[2]

Employee **where** *Salary* < 2000 **and** *Age* > 40 (*)

the part

Salary < 2000 **and** *Age* > 40

is a block evaluated in a new environment, which is determined by the currently processed *Employee* object. Thus, for the evaluation of this subquery ES is augmented with a new section containing information about the internal properties of the object. After the evaluation this section is popped.

ES consists of *sections*, which are sets of *binders*. A binder is a pair (n, x), where n is an external name, and x is some value, in particular, a reference to an object; such a pair is written as $n(x)$.

In general, binders serve name binding occurring in queries. For instance, if binder $n(x)$ is on ES and we want to bind the name n, then the result of the binding is x. The "search from the top" rule means that when n is being bound, we are looking for the binder $n(x)$ that is closest to the stack's top. To cover bulk data structures of the store model, we assume that binding is multi-valued: if the relevant section contains more binders whose names are n: $n(x_1)$, $n(x_2)$, $n(x_3)$,..., then all of them form the result of the binding. In such a case binding n returns the collection $\{x_1, x_2, x_3, \ldots\}$.

2.2 Opening New Sections on ES and Binding

The rules for opening a new section on ES by a non-algebraic operator for the object model with roles are a natural modification of the rules for the SBA model without roles. The most important differences for the model with roles are the following: first, there are new sections for the properties of the roles (and possibly their owners); second, the database sections contain binders to root roles.

In contrast, the binding rules for the data model with roles are exactly the same as for the classical (i.e. without dynamic object roles) model. All roles' names can be bound in a database section of ES. If the model with roles is used for programming of applications, then roles' names can also be bound in the section of the current user session and in sections containing local environments of procedures and methods. The rules for auxiliary naming (the **as** operator known from OQL) are the same as for the classical model.

In Fig. 2 we present example states of ES during evaluation of query (*) for the object store in Fig. 1. Sections irrelevant for this example are not shown. The figure shows the idea of *thin* and *thick* sections: when a new scope is opened on

[2] In OQL, this query can be defined as: **select** *e* **from** *Employee* **as** *e* **where** *e.Salary* < 2000 **and** *e.Age()* > 40.

Fig. 1. An example object store

ES, then a new thick section, which consists of possibly several thin sections, is created. We mark thick sections with thick lines, and thin sections with thin ones. (These concepts are introduced only for better comprehension of the environment stack mechanism in our approach. In particular, the division has no influence on the binding rules.)

The first ES state consists only of the database section containing binders for owners and their roles, where the name *Employee* is bound (returning $\{i_{13}, i_{16}\}$). The second state presents the situation when **where** is processing the object whose identifier is i_{16}. The name *Salary* is bound at the top (returning i_{17}) and the name *Age* is bound in the *PersonClass* section, the fourth from the top (returning i_{41}). The next state concerns executing the *Age* method after pushing the method's local environment onto the top of ES. During execution of the method, the section of the *Employee* role (the second from the top) and the section of *EmployeeClass* (the third from the top) are invisible due to static scoping. The final state, after executing the query, is the same as the beginning state.

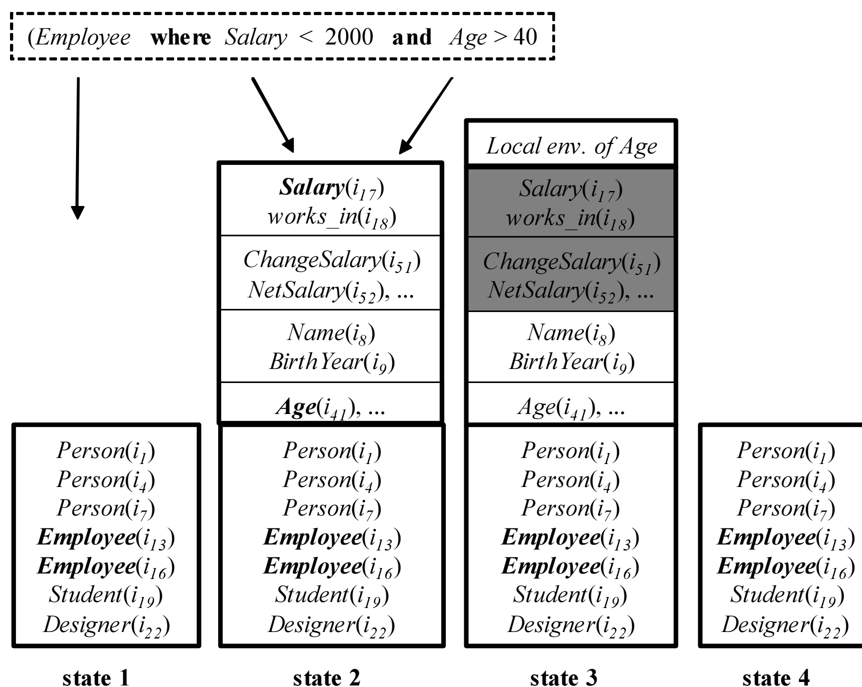

Fig. 2. States of ES during processing query (*)

2.3 Polymorphism and Overriding

The discussed above stack-based semantics supports polymorphism due to the fact that each role and its class are encapsulated. A designer is allowed to use the same name for different methods defined in an owner's class and classes of its roles. The choice of method depends on the class membership.

Overriding is naturally supported by the scoping rules as well. In particular, it is possible to override a method defined for a role by a method defined for its sub-role. The overriding mechanism is extended: it is possible to override an attribute defined for a role by a method defined for its sub-role (and vice versa). Such a feature can be useful e.g. when in a specialized role one wants to replace an attribute with a virtual attribute.

3 SBQL with Dynamic Object Roles

The SBQL with dynamic object roles is an extension of the early SBQL query language[3] implemented in the LOQIS System [6] with an adapted semantics and some new constructs. The main changes with regard to the early SBQL are:

[3] Its denotational semantics is presented in [10].

- The semantics of the operators has been extended to deal with objects and roles. In our approach objects and their roles possess different identifiers.
- Naming, scoping, and binding rules have been changed. Each role has its own name, which can be used to bind the role from a program or a query. The objects presented in Fig. 1 can be bound either through name *Person*, or name *Employee,* or name *Student*, or name *Designer*. Each binding returns the identifier of a proper role (or the identifiers of proper roles in the case of multi-valued bindings). Because roles inherit dynamically properties from their owner-objects/roles, a new *nested* function has been defined, and the scoping/binding rules have been extended as well.
- Because roles cannot exist without their owners, the operators for creating and deleting object and roles have been changed.
- Some new role-specific operators have been introduced in order to act on objects with roles in an effective and precise manner.

3.1 Creating and Deleting Roles

Creating and deleting objects and roles require introducing some changes to the semantics of **create** and **delete** operators. In this section we briefly discuss the issue. However, we do not consider how to incorporate constraints[4], which can be imposed on objects with roles, and expressed in some data definition languages (e.g. in a class schema) [8, 9].

Creating Roles Due to various constraints that can be imposed on object and roles, the *create* operator should be able to create not only single objects, but objects with roles too. We define the grammar for creating objects and roles (in a modified BNF[5] notation) as follows[6]:

(1) <create_def > ::= { <create_expr> }
(2) <create_expr> ::= **create** <create_kind> [**as** <auxName>]
[”(” <attribute_list> ”)”] [”{” <with_role_list> ”}”] ”;”
(3) <create_kind> ::= <Class> | **role** <RoleClass> **of** <identifier>
(4) <attribute_list> ::= <attribute> [{ ”,” <attribute> }]
(5) <attribute> ::= <attribute_name> ”=” <value>
(6) <value> ::= <literal> | <collection_value>
(7) <literal> ::= <integer> | <float> | <string> | <name> | **null**
(8) <collection_value> ::= ”{” <value_list> ”}”
(9) <value_list> ::= <value> [{ ”,” <value> }]
(10) <with_role_list> ::= <with_role> [{ ”,” <with_role> }]

[4] A role can be mandatory, optional, and/or multiple. A mandatory role means that its owner-object/role must possess it. Usually roles are optional. A multiple role may have many instances within an object. Some roles and objects can be mutually exclusive or must occur all along.

[5] BNF – *Backus Naur Form.*

[6] The following metasymbols are used: {S} – S may be repeated (S must occur at least once); [S] – S may occur at most once.

(11) <with_role> ::= **with role** <RoleClass> [**as** <auxName>]
["(" attribute_list ")"] ["{" <with_role_list> "}"]

Apart from the keyword and symbols, the following terminals are used:

- name – denotes any name;
- Class – denotes a class name;
- RoleClass – denotes a role class name;
- auxName – denotes an auxiliary name;
- identifier – denotes an object identifier;
- integer, float, string – denote an integer, real number and string, respectively.

Below we present a few examples showing how to create objects:

create *Company* **as** C (*Name* = "IPI");
create *Person* (*BirthYear* = 1948, *name* = "Doe");
create *Person* (*BirthYear* = 1951, *name* = "Smith") {
with role *Student* (*StudentNo* = 223344, *Faculty* = "biology"),
with role *Employee* (*Salary* = 1500, *works_in* = "ABC") };
create *Person* (*BirthYear* = 1975, *Name* = "Brown") {
with role *Employee* (*Salary* = 2500, *works_in* = "XYZ") {
with role *Designer* (*Bonus* = 1000)} };

Deleting Roles Removal of an object/role implies automatically removal of all its roles, but obviously, removal of a role does not imply removal of its owner-object/role. The role concept involves the cascade deleting semantics, applied in many database systems.

For example, the query

delete *Person* **as** *p* **where** *p.Name* = "Brown"

deletes all persons whose name is *Brown* with all their roles, while the instruction

delete *Employee* **as** *e* **where** *e.Salary* > 3000

deletes those employees (i.e. *Employee* roles) who earn more than 3000.

3.2 Role-Specific Operators

For some kinds of queries it may be useful to make an explicit conversion between different roles of an object. Such a feature is necessary e.g. for the query "get all employees which are students at the same time". Thus, it is necessary to provide a cast operator, which will convert the identifier of a role into the identifier of another role. Similarly, a Boolean operator can be introduced for testing presence of a given role within an object. Another operator may return identifiers of the roles that are currently present within an object. Such operators increase the applicability of generic programming.

The Cast Operator The well-known technique of casting can be applied in the SBQL with roles, where it enables one to make an explicit conversion between: a role and its owner-object, different roles of an owner-object, an owner-object and some of its roles. In contrast to the typical cast operators (e.g. in C++), our cast operator not only converts types, but it is a regular run-time operator mapping collections of identifiers into another collection of identifiers. Syntactically, the operator is written as:

$(n)\ q$

where n is an object/role name, and q is a query returning object/roles identifiers. The semantics of this operator is the following: it takes the identifier of an object/role returned by the q, then returns the identifier of an object/role with name n within the same owner-object/role. If the owner-object/role has no role with name n, then the result is empty. If the object has more than one role named n, then all the roles' identifiers are returned.

Using the operator we can express the following queries, for example, "select persons who are employees":

(*Person*) *Employee*

Evaluation of the query *Employee* returns the collection of identifiers of the roles *Employee* in all *Person* objects. Then the cast operator (here: (*Person*)) converts each of them into the identifier of the *Person* owner-object. The result is a bag of identifiers of *Person* objects.

We mentioned that the cast operator should act on different roles of an owner-object/role as well. For instance, we should be able to define the following query: "select persons who study and work":

(*Person*) ((*Employee*) *Student*)

The above query is almost equivalent to the query:

(*Person*) ((*Student*) *Employee*)

Both queries lead to the same results modulo duplicates.

The cast operator is a powerful operator which also acts on objects with nested roles. For instance, if we assume that a student can work as depicted in Fig. 1, than we are able to select persons who are students working as designers:

(*Person*) ((*Student*) *Designer*)

The hasrole Operator The **hasrole** operator tests presence of a given role within an object/role. Syntactically, the operator is written as:

q **hasrole** n

where n is a role's name, and q is a query returning objects/roles identifiers. The semantics of this operator is the following: it takes the identifier of an object/role returned by q, and then checks whether roles with name n exist

within this object/role. If the object/role has no n role, then the result is *false*. If the object/role has one or more than one role with name n, then *true* is returned.

Example: "Select persons who are working and are more than sixty years old":

((*Person* **where** *Age* > *60*) **as** *p*) **where** (*p* **hasrole** *Employee*)

The Roles Operator In this section we introduce another operator, called **roles**, which can be applied to access only direct[7] roles of a given object/role. The operator returns identifiers of certain direct roles that are currently present within an object/role. Syntactically, the operator is written as

roles [n] **of** q

where n is a role's name and q is a query returning identifiers of objects/roles. The semantics of this operator is defined as follows:

- If n is specified, the operator takes the identifier of the object/role returned by q, then checks whether direct n roles exist within the object/role. If the object/role has no such roles, then the result is empty. If the object/role has one or more than direct n role, then all their identifiers are returned.
- If n is not specified, the operator takes the identifier of the object/role returned by q, then checks whether any direct roles exist within the object/role. If the object/role has no direct roles, then the result is empty. If the object/role has one or more direct roles, then all their identifiers are retrieved.

Example: "Retrieve the names of all roles acquired by *Smith*".

unique (**nameof** (((**roles of** (*Person* **where** *name* = "Smith")) **as** *r*) **close by** ((**roles of** *r*) **as** *r*)))

In the above example **nameof** is an operator that returns the name of an object/role, **unique** is an operator for removing duplicates, and **close by** is an operator for transitive closure[8].

For the example object store shown in Fig. 1, the query returns the following 3-element set of names: {"*Employee*", "*Designer*", "*Student*"}, while the query

nameof (**roles of** (*Person* **where** *name* = "Smith"))

returns a 2-element set {"*Employee*", "*Student*"}, because the *Designer* role is an indirect role of *Smith*.

[7] Each role is the direct role of some owner-object/role (and direct instance of some class) and the indirect role of the ancestors of the owner of the role within the role hierarchy.

[8] The idea of transitive closure is applied for instance to process recursive data structures and to encapsulate some non-trivial iterations. The implementation of transitive closure in SBQL is presented e.g. in [10].

4 Summary

In the paper we have presented general assumptions concerning an extension to the SBQL query language, which incorporates the concept of dynamic object roles. The concept is a powerful modeling tool that makes it possible to express that e.g. an object during its lifetime can acquire and lose roles without changing its identity.

The SBQL extension is based on the modified (that is, involving dynamic object roles) version of SBA. In particular, the modified version of the environment stack needs a new kind of sections to store binders to roles' properties. However, the general idea remains unchanged.

Our further research will concern embedding such a query language into imperative programming constructs e.g.: creating objects and roles, inserting and deleting roles, control statements, procedures and methods, views, etc.

References

1. A. Jodłowski, P. Habela, J. Płodzień, K. Subieta: Objects and Roles in the Stack-Based Approach. Proc. of DEXA, Springer LNCS 2453, pp. 514-523, 2002
2. J. Płodzień, A. Kraken: Object Query Optimization through Detecting Independent Subqueries. Information Systems 25(8), Elsevier Science, pp. 467-490, 2000
3. J. Płodzień, K. Subieta: Applying Low-Level Query Optimization Techniques by Rewriting. Proc. of DEXA, Springer LNCS 2113, pp. 867-876, 2001
4. J. Płodzień, K. Subieta: Static Analysis of Queries as a Tool for Static Optimization. Proc. of IDEAS, IEEE Computer Society, pp. 117-122, 2001
5. K. Subieta, Y. Kambayashi, J. Leszczyłowski: Procedures in Object-Oriented Query Languages. Proc. of VLDB, pp. 182-193, 1995
6. K. Subieta, M. Missala, K. Anacki: The LOQIS System. Description and Programmer Manual. Institute of Computer Science, Polish Academy of Sciences, Report 695, Warsaw, Poland, 1990
7. F. Steimann: On the Representation of Roles in Object-Oriented and Conceptual Modeling. Data & Knowledge Engineering 35(1), Elsevier Science, pp. 83-106, 2000
8. A. Jodłowski, P. Habela, J. Płodzień, K. Subieta: Dynamic Object Roles in Conceptual Modeling and Databases. Institute of Computer Science, Polish Academy of Sciences, Reports 932, Warsaw, Poland, 2001
9. A. Jodłowski, J. Płodzień, E. Stemposz, K. Subieta: Introducing Dynamic Object Roles into the UML Class Diagram. Proc. of IASTED International Conference on Software Engineering and Applications (SEA), ACTA Press, pp. 629-634, 2002
10. K. Subieta: Denotational Semantics of Query Languages. Information Systems 12(1), Elsevier Science, pp. 69-82, 1987