Query Processing in an Object Data Model with Dynamic Roles

JACEK PŁODZIEN™ and KAZIMIERZ SUBIETA‡

™Institute of Computer Science, Polish Academy of Sciences
ul. Ordona 21, 01-237 Warsaw, POLAND
‡Warsaw School of Economics
Al. Niepodległości 164, 02-554 Warsaw, POLAND
™Polish-Japanese Institute of Information Technology
ul. Koszykowa 86, 02-008 Warsaw, POLAND

http://www.ipipan.waw.pl/~{jpl, subieta}

Abstract: The paper presents the general framework of a query language for a database based on an object-oriented data model with dynamic object roles. A number of questions both for the language and its data model (the so-called stack-based approach) are covered, among others, managing an environment stack, binding names, and casting. The article also considers the issue of query optimization for the language. The discussion is illustrated with examples and figures.

Key-Words: object-orientation, dynamic object roles, databases, query processing

1 Introduction

The object-oriented data model is a generalization of the relational one and is believed to eliminate many of its flaws through incorporating modern concepts. One of them is dynamic object roles with the following characteristics [10]: a role has its own properties and behavior; an object can acquire and abandon roles dynamically without changing its identity; an object can play different roles simultaneously; an object can play the same role several times.

The current object-oriented data models can express static properties, e.g., the fact that an employee “is a” person. However, it is more precise to say that a person “becomes” an employee for a period of time and later he/she terminates the employee role. Moreover, a person can be an employee two or more times.

In general, the concept of dynamic object roles assumes that an object (a so-called “owner-object” or “owner”) can be associated with other objects (so-called “role-objects” or “roles”), which model its roles. 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. As an object, a role can have its own additional attributes, behavior, etc. Moreover, roles can be further specialized as subroles, sub-subroles, etc, e.g., specialization of an Employee role can be a Superior role.

Dynamic roles can significantly support conceptual modeling and, in comparison to the classical object models, do not lead to the anomalies and limitations of multiple, repeating, and multi-aspect inheritance. For instance, two roles (of the same object) can contain attributes and methods with the same names without implying any conflict. This is a fundamental difference in comparison to the concept of multiple inheritance.

Associations between objects can connect not only owners with owners, but also owners with roles and roles with roles. For example, a works_in association can connect an Employee role with a Company owner-object.

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.

To a big degree dynamic roles can be an important paradigm for object-oriented databases and their query languages constructed e.g. in the spirit of the ODMG standard [3]. The concept is already present (under another name and with

1 This work was partially supported by the European Commission project ICONS; project no. IST-2001-32429.
specific semantics) in the standard SQL3 (abandoned) and its successor SQL1999 [9].

In spite of many proposals, the problem of object-oriented query languages is still considered open. The ODMG standard with its OQL (Object Query Language) is criticized by some specialists due to its inconsistencies, lack of precise specification, etc [1]. Therefore in our research on dynamic object roles we use another proposal – the Stack-Based Approach (SBA) along with its query language SBQL. In our opinion it does not have the disadvantages of its competitors. (Due to space limit we are unable to present SBA and SBQL. Therefore in this paper we assume at least a general familiarity with these concepts; the reader is referred to [4, 7].)

In our previous work [2] we have incorporated dynamic object roles into SBA. In this paper we discuss how this concept can be incorporated into SBQL.

An early version of the roles concept has been implemented in the prototype system Loqis [8]. Currently we are working on a prototype of a content management system where we intend to implement the ideas presented here.

The remainder of the paper is structured as follows. In Section 2 we discuss incorporating dynamic object roles into SBQL. Section 3 covers the issue of query optimization for the query language. Section 4 summarizes the paper.

2 SBQL with Dynamic Object Roles

In our discussion we assume that Person objects can possess Employee and Student roles. Fig. 1 presents an example object store built in accordance with our data model with 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, or a set of objects); e.g., one of the objects is named Person, its identifier is \(i_1\), and its value is two objects with identifiers \(i_2\) and \(i_3\).

- There are three objects storing the invariant properties of Person, Employee, and Student objects; they are PersonClass, EmployeeClass, and StudentClass, respectively.

- Each object has access to the invariant properties of its class. This is denoted as a thick gray arrow; e.g., the Student role is connected to the StudentClass object.

- Roles dynamically inherit the properties of their owner-objects. This is denoted as a double-line with a diamond end; e.g., Employee roles dynamically inherit from their Person owner.

2.1 Environment Stack

In programming languages a special data structure called an environment stack (ES) is responsible for scope control and binding names. A new section of volatile objects (a so-called activation record) is pushed onto the stack when a procedure/block is started, and the section is popped when the procedure/block is terminated. An activation record for a procedure invocation contains volatile variables (objects) declared within this procedure, its actual parameters, its return address, and other data. Binding follows the “search from the top” rule. The last added section is the first one visited during the binding, and objects from some sections remain invisible for the binding (for so-called static scoping).

SBA involves 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

\[
\text{select } e \text{ from Employee as } e \text{ where } e.\text{Salary} < 2000 \text{ and } e.\text{Age}() > 40.
\]
**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 query, ES is augmented with a new section containing information about the internal properties of the object. After the evaluation this section is popped.

Figure 3. Sections pushed onto ES by a non-algebraic operator in SBA with roles

**2.2 Opening New Sections on ES**

Consider a query including some name:

**Employee** where ... **n** ...

(***)

In the classical SBA model (i.e., without dynamic object roles), while binding **n**, ES has the following sections going from the top: the internal environment of the currently processed object; the environment of its class (binders to **EmployeeClass**’s properties); the environment of the superclass (binders to **PersonClass**’s properties). A detailed description of the stack organization and its behavior for particular query operators can be found in our papers on SBA.

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 the discussion below we consider a query **q_1** \( \theta q_2 \), where \( \theta \) is a non-algebraic operator, **q_1** and **q_2** are sub-queries.

**2.2.1 The Model without Roles**

Let \( q_1 \) return the identifier of some object **O**. Let \( O \) be a member of **C1O** class, which inherits statically from **C2O**, which in turn inherits from **C3O**, etc. Let **O, C1O, C2O, C3O, ...** have identifiers \( i_{C1O}, i_{C2O}, i_{C3O}, ... \), respectively (in SBA, classes, methods, etc, are objects too). Then \( \theta \) pushes onto the top of ES the corresponding sections in the order shown in Fig. 2 (nested is a function returning binders to the internal properties of the object, whose identifier is the argument of the function).

Fig. 2. Sections pushed onto ES by a non-algebraic operator in SBA without roles

**2.2.2 The Model with Roles**

Let \( q_1 \) return the identifier of an **R1** role. Let **R1** inherit dynamically from **R2**, which in turn inherits dynamically from **R3**, etc. Let **Ri** \( (i = 1, 2, ...) \) be a member of **C1Ri** class, which inherits statically from **C2Ri**, which in turn inherits from **C3Ri**, etc. The corresponding identifiers are: \( i_{R1}, i_{R2}, i_{R3}, ..., i_{C1R1}, i_{C2R1}, i_{C3R1}, ..., i_{C1R2}, etc \). In such a case \( \theta \) pushes onto the top of ES the corresponding sections in the order shown in Fig. 3.

Fig. 3. Sections pushed onto ES by a non-algebraic operator in SBA with roles

All the opened sections are removed after processing the **R1** role. Other rules concerning opening/closing ES sections by a non-algebraic operator remain unchanged.
For simplicity, Fig. 2 and Fig. 3 neglect encapsulation that subdivides properties into public and private. However, the rule for the object model with roles remains the same as for the classical model, that is, private properties of an object of a given class are available only to the methods that are stored within this class.

Fig. 4 presents an example state of ES while processing Employee roles for the store from Fig. 1 in accordance with the ES manipulation idea shown in Fig. 3.

The search order during binding name n

| Properties of the currently processed Employee role r |
| Properties of EmployeeClass |
| Properties of the Person owner of r |
| Properties of PersonClass |
| ... other visible stack sections ... |
| ... invisible sections due to static scoping ... |
| Global properties of the current user session |
| Root database objects, views, database procedures, ... |
| Global library procedures, environment variables, ... |

**database sections**

**Fig. 4. An example for the stack in Fig. 3**

### 2.3 Binding

Binding rules for the data model with roles are exactly the same as for the classical model; see Fig. 4 for binding name n in query (**). 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. 5 and Fig. 6 we present example states of ES during evaluation of query (*) for the object store in Fig. 1. Sections inessential for this example are not shown. The first ES state contains only the database section containing binders for owners and their roles, where the name Employee is bound (returning {i13, i16}). The second state presents the situation when where is processing the object whose identifier is i16. The name Salary is bound at the top (returning i17) and the name Age is bound in the PersonClass section, the fourth from the top (returning i41). 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. 5. States of ES during processing query (*) (part 1)**

**Fig. 6. States of ES during processing query (*) (part 2)**

### 2.4 Polymorphism and Overriding

The discussed above stack-based semantics supports polymorphism due to the fact that each role and its class are encapsulated. Thus the designer can use the...
same name for different methods stored within different classes.

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.

2.5 Cast Operators

The well-known technique of casting can be applied in SBA 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 one of its roles.

Such a feature is necessary e.g. for the query “get all employees who are students”. In contrast to the typical cast operators (known e.g. from C++), our cast operators not only convert types, but they are also regular operators mapping a collection of identifiers into another collection of identifiers.

Syntactically, the operator is written as:

\[(\text{name}) \text{query}\]

where \text{name} is the name of an owner-object or a role, and \text{query} returns identifiers of owner-objects or roles. If \text{query} returns owner-objects’ identifiers, then the operator returns the identifiers of \text{name} roles within those objects. If \text{query} returns roles’ identifiers, then the operator returns the identifiers of their owner-objects (for \text{name} objects) or the identifiers of other roles within the same objects (for \text{name} roles). If an object has no role named \text{name}, then the result is empty (null).

Through the operator we can express, for instance, the queries below.

Example 1: “Get employees who are students.”

\text{Employee} \text{Student}

Evaluation of the query \text{Student} returns a collection of the identifiers of \text{Student} roles. Then the \text{Employee} cast operator converts each of them into the identifier of an \text{Employee} role or into null, if a given object has a \text{Student} role, but has no \text{Employee} role. The result is a collection of the identifiers of \text{Employee} roles; nulls are ignored.

Example 2: “Get persons named Jones who are students.”

\text{Person} (\text{Student where Name = “Jones”})

Evaluation of the query \text{Student where Name = “Jones”} returns a collection of the identifiers of \text{Student} roles in those objects for which the value of the \text{Name} attribute is “Jones” (note that this value can be accessed in \text{Student} roles through dynamic inheritance from their \text{Person} owner-objects). Then the \text{Person} operator converts each of them into the identifier of the corresponding \text{Person} owner-object. The result is a collection of those identifiers.

Example 3: “Suppose that students have a Scholarship attribute. For each person, get his/her name and income: the salary for employees, the scholarship for students and the sum of the salary and scholarship for working students. For a person, who is neither an employee nor a student, the income is 0.”

\text{Person as p} . (p . \text{Name}, \text{sum(0, ((Student) p) . Scholarship, ((Employee) p) . Salary)})

In the above example, \text{sum} is an aggregate function similar to the corresponding SQL function.

Similarly we can introduce a Boolean operator testing presence of a given role within an object. Another operator can return the names of those roles that are currently present within an object. Such operators increase the generic programming ability.

3 Query Optimization

Because the model with dynamic roles is based on the standard SBA, the general idea of static query optimization through static analysis, as presented e.g. in [4, 5, 6], remains generally the same. To cover the concept of roles, the database schema graph needs a new kind of nodes to store definitions of roles and a new kind of edges for the \text{is_role_of} relationship between roles.

Some modifications are needed for the query optimization techniques. (Due to space limit we avoid technical details and concentrate on general ideas and concepts illustrated by examples.) Among others, the stack’s sizes calculated in the method of independent subqueries [4] should concern the modified ES storing sections for roles as presented in Section 2.2.

In addition, optimization techniques, which have not been considered so far, can prove useful in the new model. For instance, the cast operators discussed in the previous section can lead to a situation when an auxiliary name cannot be eliminated but the query can still be rewritten to a more efficient form. The following illustrates the idea. Consider the query “get persons born after
1950 along with their companies” in SBQL with dynamic roles:

\[(\text{Person as } p) \bowtie ((\text{Employee}) p) . \text{works\_in} . \text{Company}) \text{ where } p.\text{BirthYear} > 1950\]

In this query the dependent join operator \(\bowtie\) joins each Person owner-object (named \(p\) in this query) with the Company object that the person \(\text{works\_in}\) iff the person has an Employee role, which is determined by the cast operator \((\text{Employee})\). The resulting pairs \((p^i_{\text{Person}}, i_{\text{Company}})\) (where \(i_{\text{Person}}\) and \(i_{\text{Company}}\) are references to Person and Company objects, correspondingly) are then filtered by \text{where}, which selects only those pairs, for which the person is born after 1950. According to the optimization rules presented in the papers referred to above, the selection predicate can be pushed before the \(\bowtie\) operator:

\[(((\text{Person as } p) \text{ where } p.\text{BirthYear} > 1950) \bowtie ((\text{Employee}) p) . \text{works\_in} . \text{Company})\]

but the auxiliary name \(p\) cannot be removed, because it is used after the join in casting. Nevertheless, we can perform the selection before introducing \(p\):

\[(((\text{Person where } \text{BirthYear} > 1950) \text{ as } p) \bowtie ((\text{Employee}) p) . \text{works\_in} . \text{Company})\]

Such a case can be especially common when the optimization concerns queries involving views, that is, when such a selection is applied to a view invocation, which is then macro-substituted. The example shows an ability to apply query-rewriting techniques to queries addressing the object model with roles.

Although optimization techniques for the model need further development, we do not expect that the model implies totally new query optimization problems.

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: