Using schema information for querying databases

Iztok Savnik[a], Zahir Tari[b], Tomaž Mohorič[c]

[a]Computer Systems Department, Jožef Stefan Institute, Slovenia
[b]School of Information Systems, Queensland University of Technology, Australia
[c]Faculty of Electrical Eng. and Computer Science, University of Ljubljana, Slovenia

Abstract

We propose the set of operations for querying the conceptual schema of an object-oriented database. The operations form the basis of an algebra for objects called OVAL. They are defined using the constructs introduced for our formalization of the object-oriented database model. The operations allow a user to query: (i) associations among individual objects, (ii) relationships between individual objects and class objects, and (iii) relationships among class objects themselves.

1 Introduction

Object-oriented database model provides a rich set of modeling constructs that make the conceptual schema of an object-oriented database more expressive than relational schemas. We observe that, comparing a relational database to an object-oriented one, some information about the modeling environment has been moved from the *data* part to the *schema* part of the database. Hence, some aspects of the modeling environment can be, using an object-oriented database model, represented and stored in a database by means of a database schema. Consequently, the *schema* part of an object-oriented database should be treated in a similar manner as the *data* part of the database: it is, like ordinary data, the subject of the user's inquiry and modification.

In general, there are two types of queries which relate to the conceptual schema. Firstly, the user should be able to query the relationships between the instances and the conceptual schema of a database. Secondly, due to the frequently very complex conceptual schema, a user should be able to query it in order to obtain a precise mental image of the structure and the behavior of stored information [10].

In this paper we present the operations of the *algebra for objects* called OVAL, which are used for querying conceptual schema. The following section briefly overviews the work related to OVAL. Next, the basic constructs used for the formalization of the OVAL's data model are defined in Section 3. The basic operations of the algebra OVAL are presented in Section 4. Finally, the concluding remarks are given in Section 5.

2 Related work

The constructs that recent query languages provide for querying database schema are briefly presented in this section. To our knowledge, recently proposed database algebras (e.g., Query algebra [13], Excess [14] or Complex Object Algebra [1]) do not include such facilities.

Firstly, most recent query languages (e.g., query languages of ORION [9] or O_2 DBMS [4]) provide the constructs for using the class extensions [5] in queries. In [5] Bertino proposes the use of operator $CLASS\_OF$, which returns the class of an object at run-time. The resulting class can be further used in a query. Next, ORION [9] provides a set of operations for modifying database schema at different levels: modification of inheritance, class properties, methods and inheritance hierarchy of classes.

In [8] Kifer and Lausen propose a declarative language based on logic, called F-Logic, which includes the capabilities for querying database schema. The relationships between instances and classes, which are based on the *isa* hierarchy of database objects, can be in F-Logic queried using the predefined predicates for testing class membership and subclass relationship. Further, F-Logic provides the capabilities to explore the properties of individual and class objects by treating attributes and methods as objects that can be manipulated in a similar manner to other database objects. In this way, some types of non-trivial relationships among objects such as the analogy and the similarity relationships can be expressed in F-Logic.

Next, the query language XSQL [7] includes a set of constructs for querying database schema. XSQL queries can include variables that range over class objects. Therefore, classes can be queried on the basis of their properties and the properties of their instances. The XSQL operation $subclassOf$ can be in this context used to inquire about the relationships among classes which are based on the inheritance hierarchy of classes. In a similar manner to F-Logic, XSQL also treats attributes and methods as objects that can be queried; hence, a user can inquire about the properties of individual and class objects.

Finally, in [10] Papazoglou suggests a set of high-level operations for expressing *intensional* queries which aid a user to understand the meaning of stored data. The proposed operations can express the following types of queries: relate individual objects to classes, browse the *isa* hierarchy of classes, inquire about the class properties described using attributes, compute associations among classes which are not related by *isa* relationship, locate objects on the similarity basis and inquire about the dynamic evolution of objects represented by *roles*.

3 Data Model of OVAL

The algebra for objects OVAL is tightly related to its data model which provides, in addition to the basic constructs of the object-oriented database model [3], an uniform view of the database by treating classes as abstract objects.

This section overviews the basic features of our formalization of the object-oriented database model which serve as the platform for the development of the algebra OVAL. More details about the formalization can be found in [11].

3.1 Objects and Classes

An *object* is defined as a couple $< i, v >$, where i is the object identifier and v its corresponding value. An *object identifier* (oid) is a reference to an object, and an *object value* represents the state of the object, called an *o-value* [2]. The o-value is either: (i) a constant, (ii) an oid, (iii) a set of objects $\{o_1, \ldots, o_n\}$, where o_i-s represent o-values, or (iv) a tuple object defined as $[A_1 : o_1, \ldots, A_n : o_n]$, where o_i-s represent o-values and A_i-s are attribute names.

The data model supports two types of objects: *class objects* and *individual objects*. The class object represents an abstract concept and acts as a representation of a set of objects which share similar static structure and behavior. The interpretation of a class object is the set of objects that are called the *members* of a given class object. The interpretation of class c is denoted by $I(c)$. Furthermore, the interpretations of two classes are non-overlapping sets of object identifiers. Therefore, an individual object has exactly one parent class object.

The set of classes from a given database is organized according to the partial ordering relationship *is_a_subclass*, which we denote $\preceq_i$. The partially ordered set of classes is extended to include individual objects. The member of a given class is related to this class by the relationship $\preceq_i$. Formally, $o \in I(c) \Longrightarrow o \preceq_i c$, where o represents an individual object and c is a class object.

The *inherited interpretation* [2, 14] of class c, denoted by $I^*(c)$, includes all instances of class c, i.e. the members of class c and the members of its subclasses. Formally, $I^*(C) = \bigcup_{C_j \preceq_i C \wedge C_j \in \mathcal{V}_C} I(C_j)$, where $\mathcal{V}_C$ denotes the set of all classes from a given database.

3.2 O-Values and Types

A *type* is a pair in the form of $(S; P)$, where S represents the structure of a set of objects and P describes their behavior. This sub-section includes the description of the structural part of a type, which we call *static type*. The behavioral part of a type is not presented in this paper; its description can be found in [11]. The static type can be: (i) a primitive type, (ii) a reference type, (iii) a set-structured type and (iv) a tuple-structured type.

The primitive types are: *int*, *real* and *string*. A reference type is specified by a class object. The object identifier of the class *person*, for instance, denotes a reference type whose instances are references, i.e. object identifiers that are the elements of the class *person* interpretation. A set-structured type is defined as $S = \{S_1\}$, where S_1 is again a static type. A tuple-structured type is in the form of $S = [a_1 : S_1, \ldots, a_n : S_n]$, where a_i-s represent attribute names and S_i-s are again static types.

The interpretation of a static type is the set of o-values, the structure of which is defined by a given type. The interpretation of the primitive type is the set of constants of that type. The interpretation of a reference type is defined using the inherited class interpretation. The interpretation of a tuple structured type is $I([a_1 : T_1, \ldots, a : n : T_n]) = \{[a_1 : o_1, \ldots, a_n : o_n]; o_i \in I(T_i), i \in [1..n]\}$. Finally, the interpretation of a set-structured type is $I(\{S\}) = \{s; s \subseteq I(S)\}$.

In addition to the partial ordering relationship $\preceq_i$, a partial ordering relationship among o-values, denoted by $\preceq_o$, is defined. We call it the relationship *more_specific*. First, the partial ordering of static types is introduced. The partial ordering relationship defined among types is usually called a *subtype* relationship [14]. Intuitively, if type S is the subtype of type T, then the type S is more specific than (or refines) the type T. The reference type T_1 is the subtype of T_2 whenever there exists the *subclass* relationship between T_1 and T_2, i.e. $T_1 \preceq_i T_2$. Next, the type $\{S_1\}$ is the subtype of $\{S_2\}$, if S_1 is the subtype of S_2. Finally, $[A_1 : T_1 \ldots, A_k : T_k]$ is the subtype of $[A_1 : S_1 \ldots, A_n : S_n]$, if $k \geq n$ and T_i is the subtype of S_i, where $i \in [1..n]$. Again, as with the partially ordered set of oids, the partially ordered set of types is extended to include the instances of types. Formally, $v \in I(T) \Longrightarrow v \preceq_o T$, where T is a static type and v is an o-value. The obtained partially ordered set includes all o-values from a given database.

In a similar way to the inherited interpretation of classes, we define the *inherited interpretation* of types. Given the type T, the inherited interpretation of the type T includes the union of interpretations of the type T and all its subtypes. Formally, $I^*(T) = \bigcup_{T_j \preceq_o T \wedge T_j \in \mathcal{V}_T} I(T_j)$, where $\mathcal{V}_T$ denotes the set of all types from a given database.

Finally, the *extended interpretation* of structural types is defined. The extended interpretation of the type T, denoted by $I^\diamond(T)$, includes all o-values that are more specific than T. Formally, $I^\diamond(T) = \{o; o \preceq_o T\}$. The extended interpretation is used to define the semantics of OVAL variables.

4 Algebra for Objects

The algebra OVAL includes two types of operations: *model-based* and *declarative* operations. The former are used for the manipulation of object properties that are defined by the use of the database model constructs. The later are defined for expressing declarative queries on a database.

4.1 Model-based operations

The model-based operations are closely related to the constructs of the previously presented formalization of the object-oriented database model. They are intended to inquire about: (i) associations among individual objects, (ii) relationships between individual objects and class objects, and (iii) relationships among class object themselves. The use of operations is described by examples that are expressed in a predicate calculus notation.

Valuation operator

Given an object, the information describing the static properties of this object can be derived by means of the valuation operator *val*. Let us present an example of using the valuation operator. In the following expression the operator *val* is used to obtain the static properties of the class object *student*.

$$student.val = [name : string, age : int, \\ friends : \{person\}, \\ lives\_at : string, attends : \{course\}] \tag{1}$$

If the valuation function is followed by the attribute name, the expression can be abbreviated using the operator denoted by "->" as it is common in procedural programming languages.

Extension operators

Two types of extension operators are defined. Firstly, the extension operator denoted by *ext* corresponds to the ordinary class interpretation which maps a class to the set of its members. Secondly, the extension operator denoted by *exts* realizes the inherited interpretation of the class that maps a class to the set of its instances.

The following query illustrates the use of extension operators. It computes the set of persons who are either employees younger than 22, or student assistants.

$$\{o; o \in person.exts \ \wedge \ (o-> age < 22 \ \wedge \\ o \in employee.ext \ \vee \ o \in student\_assistant.ext)\} \tag{2}$$

Poset comparison operations

The simplest and most natural way to express object properties that relate to the partial ordering of objects and o-values is by using the partial ordering relationship $\preceq_o$ which is introduced in Sub-section 3.1. The comparison operations that are related to the relationship $\preceq_o$ are $\prec_o, \succ_o, \succeq_o$. Their semantics is defined in a usual manner, e.g. $a \prec_o b \iff a \preceq_o b \wedge a \neq b$. We call these operations *poset comparison operations*.

Before illustrating the use of the poset comparison operations by some examples, the function that maps an instance object to its parent class object is defined. We name this function *class_of*. It is defined as follows: $x.class\_of = c \iff x \in I(c)$, where $I(c)$ denotes class c interpretation. Note that an instance belongs to the exactly one class interpretation.

The following query specifies objects that are more specific than the class *lecturer* and, in the same time, the elements of either the class *student_assistant* or some more general class.

$$\{o; o \in person.exts \ \wedge \ \ o \prec_o lecturer \ \wedge \\ student\_assistant \preceq_o o.class\_of\} \tag{3}$$

In the above example the poset comparison operations are used to relate objects. In the following example we present the use of poset comparison operations for relating o-values. The query (4) filters the values of objects of the class *person*. The selected tuples have to include the values for the attributes *manager*, *friends* and *lives_at*. The value of the attribute *lives_at* has to be "Brisbane", and the value of attribute *manager* is required to be more specific than the class *lecturer*. In the similar way, the value of the attribute *friends* is required to be more specific than the type $\{student\}$. The query is formulated as follows.

$$\{v; o \in person.exts \ \wedge \ \ v = o.val \ \wedge \\ v \prec_o [manager : lecturer, \\ friends : \{student\}, \\ lives\_at : "Brisbane"]\} \tag{4}$$

Closure operations

The closure operations *subcl* and *supcl* are defined on class objects. Given an argument class c, the operation *subcl* returns all subclasses of c including the class c itself. The operation *supcl* returns all superclasses of the argument class c including the class c.

The closure operations can express similar relationships among objects to those that can be expressed using the poset comparison operations. The expression $x \preceq_o y$, where x and y are classes, for instance, can be expressed as $x \in y.subcl$. The query (3) can be restated as follows.

$$\{o; o \in person.exts \ \wedge \\ o.class\_of \in lecturer.subcl \ \wedge \\ o.class\_of \neq lecturer \ \wedge \\ o.class\_of \in student\_assistant.supcl\} \tag{5}$$

While the poset comparison operations can serve only for expressing relationships among objects, the result of the closure operation is a set of classes that can be further queried.

Operations *lub-set* and *glb-set*

The algebra OVAL includes the operations for computing the nearest common more general and more specific objects for a given set of objects with respect to the relationship $\preceq_i$. Let consider first the use of the operation which computes the nearest common more general objects. As an example, the nearest common more general object of the set of classes $\{professor, student\_assistant\}$ is the class $lecturer$. The class *lecturer* includes all properties which are common to the class objects from the argument set.

The operator that computes the nearest common more general objects of a set of objects with respect to the relationship $\preceq_i$ is called *lub-set*. Next, the operation *glb-set* is defined to compute the set of nearest common more specific objects for a given set of objects. Resulting objects include the properties which relate to *all* objects from the argument set.

The use of operation *lub-set* is presented in the following example. The presented expression first determines the nearest common more general objects of objects referenced by object identifiers: *peter*, *student_assistant* and *jim*. The members of the resulted classes are selected by the query. Note that *peter* and *jim* are individual objects, while the oid *student_assistant* refers to the class object.

$$\{o;\ c \in \{peter, student\_assistant, jim\}.lub\text{-}set\ \wedge\ o \in c.ext\} \tag{6}$$

Equality

The algebra OVAL provides two types of equality operations which reflect the features of the underlying data model. The first operation is the identity equality [13] denoted by the symbol "==". Two instances are *identical* if they have equal object identifiers. The second equality operation is the *value equality*. It compares objects on the basis of their values. We distinguish between two types of value equality: *complete equality* and *local equality*.

The complete equality compares two instances by comparing the values of all operand components. The operator is denoted by the symbol "=". The local equality allows the comparison of instances on the basis of the properties that pertain to the particular class. This operation is denoted by "=/class". To be able to compare two instances on the basis of the properties of the class, say C, these instances should inherit from the class C. This, of course, does not imply that they have the same parent classes. Let us present the use of local value equality by an example.

Assume that we want to compare two instances $(i1, [name{:}tone, age{:}40, works\_at{:}ijs, salary{:}10000])$ and $(i2, [name{:}vanja, age{:}24, works\_at{:}ijs, salary{:}10000, cour{:}\{c1, c2\}])$. The first instance is derived from the class *employee*, whereas the second one is the member of the class *student_assistant* which is a subclass of *student* and *employee*. These two instances are not value equal if all properties are considered. However, they are value equal if the local properties of the class *employee* are considered, i.e. *works_at* and *salary*.

4.2 Declarative operations

The algebra OVAL includes a set of *declarative* operations which are intended for querying a database. This set includes operations for: applying a query to the set of objects, set filtering, object restructuring, applying a query to the arbitrary nested component of object and computing transitive closure of a set of objects. The operations can be combined using the composition operator and the higher-order operations to form more complex queries.

In the following sub-sections we present some of the basic declarative operations of OVAL. The examples of using these operations for querying database schema are given.

The types of variables in queries are defined similarly to C++ variable definitions. For instance, the expression "$T\ v$;" defines the variable v of type T. The semantics of variables is defined using the extended interpretation of types $I^\diamond$.

Apply

The operation $apply(f)$ is used to evaluate a parameter function f on the elements of the argument set. The parameter function f can be an attribute, an operation or a query.

Let us present an example of using the operation *apply*. The query described below maps a set of students into a set of student names. The *identity* function *id* is used to identify the elements of the set *studs* which is an argument of the operation *apply*.

```
{student} studs;                                    (7)
{string} str;

str = studs.apply( id->name );
```

Selection

The operation *select*(p) is used for filtering an argument set of o-values using a parameter predicate. The parameter predicate p specifies the properties of selected o-values. It can be composed of o-values and variables related by arithmetic operations, previously presented model-based operations and boolean operations. Let us illustrate the use of operation *select* for querying database schema using some examples.

The queries (3) and (4) are restated in the following two examples to illustrate the use of poset comparison operations in the context of OVAL declarative operations.

```
{person} ps;                                        (8)

ps = person.exts.
     select( id < lecturer and
             student_assistant =< id.class_of );
```

```
{person.val} pvs;                                   (9)

pvs = person.exts.
      apply( id->val ).
      select( id < [ manager:lecturer,
                     friends:{student},
                     lives_at:"Brisbane" ]);
```

The following query illustrates the use of the operation *lub-set*. The set of instances of the class *employee* is filtered by selecting the employees who work for the Computer Systems Department and are younger than 25. The operation *lub-set* than computes the closest common more general classes of the selected set of objects.

```
{employee} s;                                       (10)

s = employee.exts.
    select( id->works_at = csd and
            id->age < 25 ).
    lub-set;
```

The use of local equality is illustrated by the query (11) which selects student assistants that have the properties that relate to their role of being employees equal to the properties of an employee referenced by the variable *peter*.

```
{student_assistant} s;                              (11)
employee peter;

s = student_assistant.exts.
    select( id.val =/employee peter.val );
```

Tuple

The operation $tuple(a_1 : f_1, \ldots, a_n : f_n)$ is a generalization of the relational *projection*. Given a set of objects as an argument of the operation, a tuple is generated for each object from the argument set. Each component of the newly created tuple is specified by the corresponding *tuple* parameter which includes the attribute name a_i and the parameter query f_i.

The query in the following example constructs the tuple for every subclass of the class *person*. Each tuple is composed of the class object identifier and the value of the class object.

```
{[ pclass: person;                              (12)
   ptype: person.val ]} ptypes;

ptypes = person.subcl.
         tuple( pclass: id,
                ptype: id.val );
```

The tuple constructed for the class *student*, for instance, is $[pclass{:}student, ptype{:}[name{:}string, age{:} int, attends{:}\{course\}]]$. Note that the role of operator *subcl* in the above query is similar to the role of extension operator.

Group

The operation $group(a : f, b : g)$ is used for grouping of o-values resulted from the query g evaluation with respect to the result of the "key" query f. Therefore, the result of evaluating the operation $group(a : f, b : g)$ on a set of o-values is a two column table, where the first column, labeled a, stores the distinct values of the query f evaluation, and the second column, labeled b, includes the corresponding values of the query g evaluation.

In the following example the operation *group* is used for grouping the instances of the class *employee* with respect to their parent classes.

```
{[ class: employee,                             (13)
   emps: { employee }]} EmpGroups;

EmpGroups = employee.exts.
            group( class:id->class_of,
                   emps:id );
```

5 Concluding remarks

The operations of the algebra for objects called OVAL, which are intended for querying database conceptual schema, are presented in this paper. These operations are called *model-based operations* since they are based on the concepts introduced for our formalization of the object-oriented database model. As the consequence, a tight correlation between the database model and the algebra for objects is established. Such correlation allows the algebra to support *all* aspects of the underlying database model.

References

[1] S.Abiteboul, C.Beeri, *On the Power of the Languages For the Manipulation of Complex Objects*, Verso Report No.4, INRIA, 1993

[2] S. Abiteboul, P.C. Kanellakis, *Object Identity as Query Language Primitive*, ACM SIGMOD 1989

[3] M. Atkinson et al. *The Object-Oriented Database Systems Manifesto*, Proc. First Int'l Conf Deductive and Object-Oriented Databases, Elsevier Science Publisher B. V., Amsterdam, 1989, pp. 40-57.

[4] F.Banchilion, S.Cluet, C.Deobel, *A Query Language for the O_2 Object-Oriented Database System*, Proc. 2nd Workshop on Database Programming Languages, 1989

[5] E.Bertino et al, Object-Oriented Query Languages: The Notion and Issues, IEEE TKDE, vol.4, No.3, June 1992

[6] P.Buneman, R.E.Frankel, *FQL- A Functional Query Language*, ACM SIGMOD, 1979

[7] M.Kifer et al, *Querying Object-Oriented Databases*, ACM SIGMOD 1992

[8] M.Kifer, G.Lausen, J.Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Technical Report 93/06, Dept. of Computer Science, SUNY at Stony Brook

[9] W.Kim, et al, *Features of the ORION Object-Oriented Database System*, 11th Chapter in *Object-Oriented Concepts, Databases and Applications*, editor W.Kim

[10] M.P. Papazoglou, *Unraveling the Semantics of Conceptual Schemas*, to appear in Comm. of ACM

[11] I.Savnik, *A Query Language for Complex Database Objects*, Ph.D. thesis, IJS-DP Technical Report, Ljubljana 1995

[12] I.Savnik, Z.Tari, T.Mohorič, *A Database Algebra for Objects*, Submitted for publication, 1995

[13] G.M.Shaw, S.B.Zdonik, *A Query Algebra for Object-Oriented Databases*, Proc. of Data Eng., IEEE, 1990

[14] S.L.Vandenberg, *Algebras for Object-Oriented Query Languages*, Ph.D. thesis, Technical Report No. 1161, University of Wisconsin, 1993