An Object Algebra for Object-Oriented Database Systems

Reda Alhajj and M. Erol Arkun
Bilkent University, Faculty of Engineering, Ankara, Turkey

Abstract

Although messages can be used to manipulate objects, a query language is still considered a required component of object-oriented database management systems. An object algebra is proposed, as a formal foundation for a query language, that can handle both the state and the behavior of objects. Creation of new objects and introduction of new relationships among objects/classes are also facilitated by the object algebra. The object algebra subsumes the five basic relational algebraic operations while providing several additional powerful ones. Each operand, as well as the result of a query, is a pair of sets: a set of objects and a set of message expressions formed as a sequence of messages accepted by the former set. The result of a query possesses the properties of an operand, thus providing the closure property for operations. Every class has a defined set of objects and derived set of message expressions, thus a class possesses the characteristics of an operand. The result of a query also possesses the characteristics of a class. The superclass/subclass relationships of a query result with the operands is established to provide persistency to the result as a class, and as a consequence algebraic equivalents of schema evolution functions are developed. Object algebra is as powerful as relational and nested relational algebra. In fact, due to message expressions that serve to handle both stored and derived values, object algebra provides greater computational power.

Keywords: Database system, object-oriented database, query language, object algebra, expressive power, schema evolution.

ACM Categories: D.3.3, F.4.3, I.1.3

Introduction

Requirements of new applications influence advances in database systems. Although the relational data model was suitable for handling conventional business applications, the first normal form restriction led to extensions to satisfy the needs of new application areas. Early extensions relaxed the first normal form by allowing set-valued attributes. A still more advanced extension is based on complex objects where sets and tuples are arbitrarily nested. To satisfy object sharing within complex objects, the notion of object identity was introduced. A more advanced step is the combination of object-oriented concepts with database technology leading to object-oriented databases. However, there is as yet no agreement on standardization within the realm of object orientation. Neither have the boundaries for the query model been set up nor an object-oriented query language well defined. This is one of the common complaints about object-oriented databases.

Explicitly speaking, object-oriented systems evolved to satisfy the needs of application areas where information about the domain is incomplete, becomes available incrementally, or is subject to change and requires flexibility in the database schema. Schema flexibility, then, should be one of the fundamental characteristics considered when judging the power of an object-oriented database system. Thus, W. Kim classifies schema changes among a number of architectural concepts developed for relational database systems that should be considered within the realm of object-oriented data models (1990). Also, extensibility is characteristic of object-oriented database systems (Atkinson, et al., 1989).

For example, on sending the message courses() to a student and the message grade() to the obtained result, the grades of the particular student are returned. Although this is handled through the implicit join present in object-oriented models, it corresponds to an explicit join in the relational model. The two messages, courses() and grade(), together form what we call a message expression. In general, a message expression for a class is defined as a valid sequence of messages $m_1...m_n$, with $n \geq 1$, accepted by the objects in that class.

While message expressions (see example above) prove object-oriented systems superior to the relational model, an object-oriented query language is still needed for more complex situations and to support associative access. The modeling power of an object-oriented database model will be unduly restricted unless new relationships can be introduced into the model through explicit joins. Allowing an explicit join raises the closure property problem. It is, therefore, necessary to have an object algebra that handles the introduction of new relationships while maintaining the closure property. The relational model satisfies the closure property with respect to the relational algebra operations; the result of any operation is a relation. Concerning object-oriented models, for the closure property to be satisfied, it should be possible to use the result of a query operation as an operand.

The Need for an Object Algebra

Two trends are being followed in providing query support for object-oriented data models. In the first case (Alashqur, et al., 1989; Banerjee, et al., 1988; Carey, et al., 1988; Maier & Stein, 1987; Straube & Ozsu, 1990) only retrieval operations are supported; in the other case (Cluet & Delobel, 1992; Dayal, 1989; Kim, 1989; Manola & Dayal, 1986; Shaw & Zdonik, 1990; Zdonik, 1988) it is also possible to create new objects. The proponents of retrieval operations argue that there is no need to create new objects by introducing new relationships into the model, as the required relationships are already defined at the modeling level. Proponents of object creation argue that it is not possible to have all the required relationships defined at the modeling level. The need for new relationships may arise while trying to satisfy changing application requirements. Hence, the power of the model should not be restricted by limiting the introduction of new relationships.

A major drawback of most query languages is that when new objects are allowed in the result, the closure property is not properly maintained. Nested relations are allowed as operands because the output of an operation is a nested relation. For instance, in O_2 values are introduced to maintain the closure property. The object algebra described in (Shaw & Zdonik, 1990; Zdonik, 1988) has its domain as sets of objects of the tuple type in which nesting of tuples is possible, and hence it is nothing more than the nested relational representation. We argue that nested relations do not form a proper logical representation of object associations.

Although in the query model of ORION the result of a query operation is a class, the improper placement of resulting classes in the lattice leads to duplication of class contents. Hence, ORION violates the reusability feature of object-oriented systems. Further, we argue that it is an overload to have a class as the output of a temporary query. In this article we describe the output of a query by the minimum requirements of an operand (given next), and from such characteristics we can derive the characteristics of a class when it is desired to make it persistent. In contrast, OSAM* operands in a query are the database itself, and all subdatabases are created from the original database by query operations (the result of a query is a subdatabase).

In this article, we describe an object algebra for object-oriented databases. We introduce the basic features of the data model on which the object algebra is based; we define a set of objects for each class and show how a set of message expressions for a class can be derived. Having a set of objects and a set of message expressions for a class is then shown to be an operand. The query algebra is described, demonstrating its power and showing how queries can handle schema changes.

An operand has a pair of sets: a set of objects and a set of message expressions defined by the elements of the first set. Message expressions preserve encapsulation and information hiding, in addition to providing full computational power via handling both derived and stored values. The output of any operation has these two sets defined and derived from the sets of its operand(s). In this way none of the object-oriented features are violated in maintaining the closure property.

Our object algebra is a superset of the relational algebra, but the semantics of the operations are different due to the object-oriented features. In addition to the relational operators, we define the nest and one level project operators. The nest operator introduces a desired relationship into the model. It is an explicit join that makes up for a missing implicit join. It is equivalent to the cross-product operation under certain conditions. The other operator — one level project — outputs the result of the evaluation of a set of message expressions against objects of the operand. The aim of this operator is to reduce the depth of nesting. The project operation, analog to the projection operation of the relational algebra, does not evaluate any message expressions but only allows access to specified parts of the objects in its operand. The algebraic operators described in this article enable the manipulation of existing objects, introduction of new relationships and creation of new objects. Operands, as well as the result of any query, possess the properties of a class and by maintaining the closure property, the output of a query can be placed in the lattice as a class thereby facilitating class-related schema changes. Thus, the operators of the algebra enable the specification of a wide variety of schema changes, eliminating the need for a separate, stand-alone language for this purpose.

In our approach the closure property is maintained without violating object-oriented concepts. An object-oriented model should be more powerful than the relational model at both the modeling and the manipulation phases. It is more powerful at the modeling phase due to the features of inheritance, encapsulation, identity and complex objects. It is more powerful at the manipulation phase due to the handling of both stored and derived values, which result in full computational power without any need to have an embedded query language and impedance mismatch.

Basic Features of the Data Model

The data model has classes that collect objects and methods. Classes are arranged in a lattice with the general class OBJECT at the root (i.e., a direct or indirect superclass of all other classes). Every object in a class, say c, should have values for all the instance variables (denoted $I_{variables}(c)$) defined within the context of class c. A class uses (inherits) instance variables and methods defined in classes higher in the lattice. Methods are executed to manipulate objects in their respective class. The following are example class definitions:

person<Ø,name:string,date-of-birth:date>
student<{person},year:integer,courses:course>
staff<{person},salary:integer>
research-assistant <{student,staff}>
course<Ø,code:string,credit:integer>

where any pair iv:d represents an instance variable defined such that iv is the instance variable name and d is the underlying domain. For example, year has an integer domain.

The first argument in a class definition is a set of classes from which inheritance is achieved. We say that person is a superclass of student and staff, while each of student and staff is a subclass of person. Any instance in student or staff is actually an instance in person but the reverse is not true. This is because, in general, a subclass may include additional instance variables and behavior definition.

To maintain the object-oriented features, it is important for the object algebra to equally handle the objects and the methods defined in a class. An object has an identity and a value. Identity distinguishes one object in the database from other existing objects and provides for object sharing. A value may be either a single value or a set of values drawn from a domain. A domain is either atomic or non-atomic; an atomic domain may be any of the conventional domains including integers, characters, etc. A non-atomic domain includes the set of objects of a class represented by their identities. The following are objects where o_i represents identity:

o_1<"John", 20-1-1972 >
o_4<"Brown", 28-6-1950 ,40K>
o_2<"Mary", 13-5-1974 >
o_5<"Lee", 1-10-1970 ,15K,5,{o_6,o_7}>
o_3<"Tom", 5-11-1965 ,5,{o_6,o_7}>
o_6<"CS530",3>
o_7<"CS565",4>

Related to an object we use value(o) and identity(o) to denote the value and the identity of object o, respectively. The identity function will be dropped and o will be used to represent identity(o) when it is clear from the context. Based on the notions of identity and value we define equality of objects.

Definition 1 — Equality of Objects

Two objects o_1 and o_2 are:

- identical ($o_1=o_2$) iff identity(o_1)=identity(o_2)
- shallow-equal ($o_1 \doteq o_2$) iff value(o_1)=value(o_2)
- deep-equal ($o_1 \cong o_2$) iff by recursively replacing every object o_i in value(o_1) or value(o_2) by value(o_i), equal values are obtained.

$$(o_1=o_2) \Rightarrow (o_1 \doteq o_2) \Rightarrow (o_1 \cong o_2)$$

identical ⇒ shallow-equal ⇒ deep-equal

We use $T_{instances}(c_i)$ to denote the set of total instances of class c_i, which includes objects in c_i and objects in all its direct and indirect subclasses:

$T_{instances}$(person)={o_1,o_2,o_3,o_4,o_5}
$T_{instances}$(staff)={o_4,o_5}
$T_{instances}$(student)={o_3,o_5}
$T_{instances}$(research-assistant) ={o_5}
$T_{instances}$(course)={o_6,o_7}

A class c has a set of messages, denoted messages(c), used to manipulate objects in $T_{instances}(c)$. For example, messages (person)={name(), date-of-birth(), age()}.

Some of the messages in messages(c) correspond to instance variables of class c to handle their values.

A message m_i in messages(c) invokes an underlying method t_i defined for the class c. The method t_i may be either locally defined in the class or inherited from one of its superclasses (directly or indirectly). The method t_i implements a specific function f_i:d_1 x d_2 x...x d_n → d_r, where d_1 is the domain of the function f_i, d_2, d3, ..., d_n are the domains from which the (n-1) arguments[1] of the function f_i are drawn, and d_r is the range of the function f_i. In other words, d_1 is the domain which has the receiver of the message m_i, d_2, d_3, ..., dn are the domains from which the arguments of the message m_i are drawn and d_r is the domain of the result of the application of the message m_i. More explicitly, given an object o_1 from d_1 (which is $T_{instances}(c)$) and some objects o_2, o_3, ..., o_n from the domains d_2, d_3, ..., d_n, respectively, o_1 $m_i(o_2,o_3,...,o_n)$ returns an object o_r from the domain d_r.[2]

It is possible to apply to the resulting object o_r any of the messages; say m_j which invokes a method t_j from the class of the object o_r. Thus, o_r m_j returns a value from the range of the function f_j implemented by the method t_j. The same value could also be obtained by applying the sequence of messages m_i m_j[3] to the object o_1. To illustrate this consider o_5 which is an object in the student class:

$o_5$courses() returns {o_6,o_7} and {o_6,o_7} code() returns {"CS530", "CS565"}

o_5 courses() code() returns {"CS530", "CS565"}.

The sequence of messages m_i m_j is called a message expression; it is an element of what we call the set of message expressions of the class c, denoted by $M_e(c)$ (given next in Definition 2). Accordingly, $M_e(c)$ is defined to include all the sequences of messages which could be applied to any object in $T_{instances}(c)$ to return a certain encapsulated value. Any such sequence of messages should be prefixed by a message in messages(c). It is important to indicate that messages(c) $\subseteq M_e(c)$ because any single message in $M_e(c)$ is actually a message in messages(c). The range of a message expression is chosen to be that of the last constituting message in the sequence. Having $T_{instances}(c_j)$ as the range of a message expression x leads to have every message expression x m_j to be in $M_e(c)$ for m_j in messages(c_j).

Definition 2—Message Expressions

The set of message expressions of the class c, $M_e(c)$ is defined to include:

Every message $m_i \in$ messages(c), i.e., messages(c) $\subseteq M_e(c)$.

In addition, $M_e(c)$ is extended to include any sequence of messages m_k m_l... m_p derived in the same way as the sequence m_i m_j.

According to Definition 2,

M_e(student)={name(), date-of-birth(), year(), courses(), courses() code(), courses() credit()}

= messages(student) ∪ courses() messages(course)

---

[1]Methods corresponding to instance variables have no arguments, i.e., they implement functions of the general form, f: d_2 → d_r

[2]Applying the message m_i to a subset of $T_{instances}(c)$ returns the set of related objects in the class of object o_r. The resulting set consists of the results of applying the message m_i to every object in the receiving subset.

[3]For brevity, we will not explicitly show the arguments of messages m_i and m_j, unless when necessary.

Function f_i, is executed as the result of invocation of the method t_i via the message m_i. The returned value may be either stored or derived. A stored value is an instance variable, while a derived value is drawn from the range of the executed function. It is derived in terms of some existing stored values. For example, date-of-birth is a stored value in objects of the person class. Age is derived in terms of the stored value date-of-birth. Thus, one can differentiate between message expressions that have derived values (like age()) and those that have stored values (like date-of-birth()). Hence, computational completeness is achieved without any need for an embedded query language having an impedance mismatch.

The Object Algebra

In this section we describe an object algebra that is more powerful than its relational and nested counterparts. It is necessary to emphasize the minimum requirements for operands in a query and the query result. These requirements are identified as the set of objects and the set of message expressions of a class. Although, an operand does have other characteristics, those characteristics are not considered at the query evaluation level. The only properties of interest are those of the already mentioned pair of sets. This is because values of objects are manipulated solely via the message expressions, regardless of the underlying structure. Since a class has a defined set of objects and a derived set of message expressions, a class can be an operand. The result of any query operation is also a pair of sets and can be made persistent in the lattice because it is possible to derive the state structure and behavior definition of the result of a query from those of the operand(s); hence, it is a class (Alhajj, 1992) .

Starting from a pair—a set of objects and a corresponding set of message expressions—it is possible to derive class characteristics. To recall, a class has a set of objects, a set of instance variables, a set of methods (each method invoked via a corresponding message) and a set of superclasses. A set of objects is given in the pair. So, finding a set of messages is equivalent to finding a set of methods, and since an instance variable has a corresponding method (and hence, a message) the set of instance variables is constructed by collecting those instance variables having a message in the calculated set of messages. The set of messages of a class is determined to include every message that appears as the first message in a sequence of messages that constitute an element of the set of message expressions of that class. Finally, the set of superclasses is determined according to the applied operation (as indicated next).

We differentiate between temporary and persistent evaluations of a query by using = for temporary and := for persistent evaluations, respectively. An assignment-free query is always evaluated on a temporary basis. While a temporary-based evaluation of a query ends by finding the pair of sets in the result, a persistent-based evaluation continues with the finding of class characteristics of the result pair. We manipulate objects depending on their being identical, shallow-equal or deep-equal, according to Definition 1. The classes introduced in the previous section will be used in all the examples presented in this section. In the rest of this section we will assume A and B to be either pairs (i.e., $<T_{instances}(A),M_e(A)>$ and $<T_{instances}(B),M_e(B)>$), or query expressions. A query expression is a sequence of one or more query operators applied on some operand to produce a pair of sets.

Selection

The selection operation presents a restriction on objects of the operand. In our object algebra, the selection has a single operand and produces an output consisting of a pair, where the included objects are those satisfying a given predicate expression (defined next). The set of message expressions of the resulting pair is the same as that of the operand. The result of the application of the selection operation on the pair $<T_{instances}(person), M_e(person)>$ which corresponds to the person class, to return persons who are older than "Tom", is the pair:$<\{o_4\}, M_e(person)>$. Notice that while $M_e(person)$ has been preserved in the result, the set of objects in the result is $\{o_4\} \subset T_{instances}(person)$. The selection oeration has the following general form:

$Select(A,p)=<\{o \mid o \in T_{instances}(A) \wedge p(o)\}, M_e(A)>$[4]

where p is a predicate expression. In a predicate expression, one variable is bound to objects of the operand and other variables are constants or bound to other queries.

Example: Find persons who are older than "Tom"

$S_1=Select(person\%p, \exists p_1 \in T_{instances}(person \wedge p_1\ name() = \text{"Tom"} \wedge p_1\ date\text{-}of\text{-}birth() > p\ date\text{-}of\text{-}birth())$

Since age() Œ messages(person), the same query could also be coded as:

$S_1=Select(person\%p, \exists p_1 \in T_{instances}(person) \wedge p_1\ name() = \text{"Tom"} \wedge p_1\ age() < p\ age())$

As seen from this example, a predicate expression is constructed using object variables that are bound to objects of an operand (by being prefixed by % as they are defined), message expressions, connectors and existential quantifiers.

Definition 3 — A Predicate Expression

The following are predicate expressions:

- T and F are predicate expressions representing true and false.
- Given two values, y_1 and y_2, having the same underlying domain such that at least y_1 or y_2 is of the form (o x), where o is an object variable bound to objects of an operand in a query and x is a message expression applicable to objects substituting for o.

— y_1 op y2 is a predicate expression where,

$$op \in \begin{cases} \{=, \neq, \leq, \geq, <, >\} & \text{if both } y_1 \text{ and } y_2 \text{ are single values from an atomic domain} \\ \{\in, \notin\} & \text{if } y_1 \text{ is a single value and } y_2 \text{ is a set of values} \\ \{\subseteq, \not\subset, =, \neq\} & \text{if both } y_1 \text{ and } y_2 \text{ are sets of values, } y_2 \text{ may be } T_{instances}(e) \text{ where e is a query expression} \\ \{=, =, \cong\} & \text{if both } y_1 \text{ and } y_2 \text{ are single values from a non-atomic domain, i.e., } T_{instances}(c) \text{ for some class c.} \end{cases}$$

[4]We use p(o) to denote the evaluation of predicate expression p by the object o substituting an object variable in p.

— $\forall$ | $\exists z \in y_1 \wedge z$ op y_2 is a predicate expression where, y_1 is a set of values and

$$\text{op} \in \begin{cases} \{=,\neq,>,<\} \text{ if } y_2 \text{ is a single value from an atomic domain} \\ \{\in, \notin\} \text{ if } y_2 \text{ is a set of values, } y_2 \text{ may be } T_{instances}(e) \\ \qquad \text{where e is a query expression} \\ \{=,\doteq,\cong\} \text{ if } y_2 \text{ is a single value from a non-atomic domain} \end{cases}$$

—$\exists z \subseteq y_1 \wedge z$ op y_2 is a predicate expression where y_1 is a set of values and

$$\text{op} \in \begin{cases} \{\subseteq, \not\subset, =, \neq\} \text{ if } y_2 \text{ is a set of values, } y_2 \text{ may be } T_{instances}(e) \\ \qquad \text{where e is a query expression} \\ \{\ni \not\ni\} \text{ if } y_2 \text{ is a single value} \end{cases}$$

- if p and q are predicate expressions, then (p), $\neg$ p, p $\wedge$ q and p $\vee$ q are predicate expressions.

So predicates within an object-oriented context are more powerful than in the relational model where only atomic values are compared. Furthermore, extending predicate expressions to allow quantifiers to propose the creation of objects increases query power. For example, $\exists$ x, x $\subseteq T_{instances}(c)$ for some class c, binds x to a subset of $T_{instances}(c)$; the subset objects to which x is bound could be built by this query. Such object creation gives the algebra the power to do recursive queries by enabling the formation of a powerset (Abiteboul & Beeri, 1988).

Project and One Level Project

For security purposes, it is sometimes desirable to hide some part of the objects in $T_{instances}(c)$ for some class c. This is possible by eliminating from the set of message expressions of a pair those message expressions related to the part to be hidden. For instance, to hide the salary of staff members, the corresponding message expression (i.e., salary()) is eliminated from message expressions of the staff class. This is possible because, although the set of objects in a pair is in general heterogeneous, the only values accessed in each object are those specified by the set of message expressions of the pair. So, by dropping some message expressions by the project operation some values are hidden from the accessible objects.

The project operation is defined as follows:

Project(A,M_1)=<$T_{instances}(A)$, M_1>

where $M_1 \subseteq M_e(A)$ (i.e., messages of M_1 could be any message expression satisfying Definition 2). Only message expressions in M_1 can be applied to objects in the pair resulting from a project operation. The inverse of the project operation is to add new elements to the set of message expressions of a pair (defined at the end of this section).

Example: Hide the salaries of staff members:

Project the pair <$T_{instances}$(staff), M_e(staff)> on M_e(staff)-{salary()} as follows:

Project(staff,M_e(staff)-{salary()})

to get the pair <$T_{instances}$(staff),M_e(person)>. Notice that in the result, $T_{instances}$(staff) is preserved while a subset of message expressions of the operand (i.e., M_e(person)) is returned in the result).

Example: Assume that the student class is not present in the lattice and the research-assistant class is defined as:

research-assistant <{staff},year:integer,courses:course>

To derive the student class we write:

student:=Project(research-assistant, {name(), age(), year(), courses()})

The derived student class will be a direct superclass of the research-assistant class. While not presented in this article, we can derive algorithms to maximize reusability through inheritance so that the derived student class will be recognized as a subclass of the person class and naturally placed in the lattice (see Alhajj, 1992).

The possibility of non-atomic domains for instance variables leads to the nesting of objects to arbitrary depth by having the value of an instance variable in an object also be an object. Thus, to facilitate the unnesting of values (decrease the depth of nesting) we define the one level project operation. The one level project operation aids in collecting values (found at arbitrary depths of nesting) within objects of the operand to form new objects. A given subset of the message expressions of the operand is evaluated against objects of the operand forming new objects and a set of message expressions is derived to facilitate accessing the values encapsulated within the derived objects. The one level project has the following form:

OLproject(A,M_1)=<{o | $\exists o_1 \in T_{instances}(A) \wedge$ value(o)=(o_1 M_1)},
{x | $\exists x_1 \in M_1$, x_1=(x_2 m)$\wedge$ len(x_1)=len(x_2)+1$\wedge$
$\exists x_3 \in M_e(A) \wedge x_3$=($x_2$ x) $\wedge$ x=(m x_4)}>

Example: Find the names and course codes of students attending at least one course:

OLproject(Select(student%s, s courses()$\neq\emptyset$), {name(), courses() code()})

Notice the use of the message expression, courses() code(), which is a concatenation of two messages, one from each of student and course classes. The input pair in this query is:

<$T_{instances}$(student), M_e(student)>

and the output pair is

<{<"Tom",{"CS530","CS565"}>,
<"Lee",{"CS530","CS565"}>}, {name(),code()}>

Example: Find names and ages of persons older than 25.

OLproject(Select(person%p, p age() > 25), {name(),age()})

The output pair is

<{<''Tom'',28>,<''Brown'',43>}, {name(),age()}>

Note the difference between the project and the one level project operations. While the former aims at merely hiding some specified values inside objects, the latter evaluates the provided set of message expressions and produces new objects out of the resulting values. However, the one level project and the project operations are equivalent under the condition that, given $M_1 \subseteq$ messages(A) and $M \subseteq M_e(A)$ such that elements of M return only stored values:

Project(A,M)=OLproject(A,M_1)

where M_1={m_1,m_2,...,m_n} and M={x_1,x_2,...,x_n} with x_i=(m_i x_{pi}) for $1 \leq i \leq n$ and arbitrary x_{pi}.

When required to be made persistent in the lattice, the result of the project operation is a superclass of the operand, while the

result of the one level project operation is, in general, a direct subclass of the root, the object class.

Cross-product and Nest

Generally speaking, it is not possible to have all the required relationships defined at the modeling stage. The flexibility of adding new relationships as they are needed can be facilitated by extending the values of the objects in a pair to include a new value that references objects in another pair. To achieve this, the nest operation is defined. The set of message expressions in the result is extended to include message expressions to access the new value added to the objects in the result. In other words, the nest operation takes two operands and it adds a value to each of the objects in the first operand, the added value having in its domain the objects in the second operand. The nest operation is defined as follows:

$Nest(A,B)=<\{o \mid \exists\, o_1 \in T_{instances}(A)\ \exists o_2 \in T_{instances}(B) \wedge$
$value(o)=value(o_1).identity(o_2)\}, M_e(A) \cup (m\ M_e(B))>$

where the domain of m is objects in $T_{instances}(B)$. The result of Nest(A,B), when required to be persistent, is a subclass of A (the first operand).

Example: Assume that both the student and the staff classes have an instance variable 'field' specifying the field of interest. To assign every student the set of staff members that he or she can consult, we write:

Nest(student%s_1, Select(Difference(staff,research-assistant)%s_2, s_1 field() = s_2 field()))

where s_1 and s_2 are object variables bound to objects of the student class and the result of the difference operation, respectively (the difference operation is defined next).

It is obvious that the nest operation forces the extension of the objects in the first operand to include references to objects of the second operand. Consequently, it is not associative. Thus we define a cross-product operation, which is associative, for query optimization purposes. (This is discussed more fully in Alhajj, 1992.) The cross-product operation considers the ranges of message expressions and returns stored values inside both operands while producing the result. Given two pairs A and B, A x B is defined according to four different cases as follows:

First case: All values present in objects of A and B have non-atomic underlying domains:

$Cproduct\ (A,B)=<\{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge$
$value(o)=value(o_1).value(o_2)\},\ M_e(A) \cup M_e(B)>$

Second case: Only objects in $T_{\{instances\}}(A)$ include at least one atomic-valued underlying domain:

$Cproduct\ (A,B)=<\{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \wedge$
$value(o)=identity(o_1).value(o_2)\},\ (m_1\ M_e(A)) \cup M_e(B)>$

Third case: Only objects in $T_{instances}(B)$ include at least one atomic-valued underlying domain:

$Cproduct(A,B)=<\{o \mid \exists o_1 \in T_{instances}(A) E o_2 \in T_{instances}(B) \wedge$
$value(o)=value(o_1).identity(o_2)\},\ M_e(A) \cup (m_2\ M_e(B))>$

Fourth case: Objects in both of $T_{instances}(A)$ and $T_{instances}(B)$ include at least one atomic-valued underlying domain:

$Cproduct(A,B)=<\{o \mid \exists o_1 \in T_{instances}(A)\ \exists o_2 \in T_{instances}(B) \wedge$
$value(o)=identity(o_1).identity(o_2)\}, (m_1\ M_e(A)) \cup (m_2\ M_e(B))>$

By considering these four cases, the cross-product operation becomes associative; an important property as far as query optimization is concerned.

When required to be persistent in the lattice, the result of the cross-product operation is a subclass of the operand that has all underlying domains being non-atomic; otherwise it is a direct subclass of the root object class.

Notice the similarity between the nest operation and the second and third cases of the Cproduct operation definition. When combined with a selection operation, both the cross-product and the nest operations result in a join operation. While the join due to a nest is an outer join, the join due to a cross-product is an inner join.

The Cproduct operation is equivalent to a combination of the Nest, Project and OLproject operations as follows:

1. If all the stored values in $T_{instances}(A)$ and $T_{instances}(B)$ have non-atomic underlying domains:

$Cproduct(A,B)=OLproject(Nest(A,B),messages(A) \cup (m\ messages(B)))$

where $T_{instances}(B)$ is the domain of the result of the message m in the result of Nest(A,B)

2. If only the stored values in $T_{instances}(B)$ have non-atomic underlying domains:

$Cproduct(A,B)=Nest(B,A)$

3. If only the stored values in $T_{instances}(A)$ have non-atomic underlying domains:

$Cproduct(A,B)=Nest(A,B)$

4. If at least one of the stored values in each of $T_{instances}(A)$ and $T_{instances}(B)$ has an atomic underlying domain:

$Cproduct(A,B)=Nest(Project(Nest(A,B),m),A)$

where $T_{instances}(B)$ is the domain of the result of the message m in the result of Nest(A,B)

Under condition 4, we have:

$Nest(A,B)=OLproject(Cproduct(A,B),(m_1\ messages(A) \cup \{m_2\})$

where m_1 and m_2 are two messages in the result of the Cproduct operation with their domains being $T_{instances}(A)$ and $T_{instances}(B)$, respectively.

So using OLproject, nest and Cproduct operations, objects may be constructed out of existing ones. Also, since the result of any operation, including nest and Cproduct, is defined to have a pair of sets, the result has the characteristics of a class, derived from the pair. Therefore, we have the possibility of introducing new classes and hence supporting views via object algebra operations.

To drop a present relationship, we project on all message expressions of the operand except those related with the pair of the relationship to be dropped as follows:

Unnest(A,B)=Project(A,M_e(A) - (m M_e(B)))

where m ∈ messages(A) and m invokes the method which implements the function with range $T_{instances}$(B).

Set Operations

As mentioned before, the object algebra described in this paper handles and produces pairs of sets, a set of objects and a set of message expressions to handle objects in the former set. Since we deal with sets, two basic set operations—union and difference—are supported by the object algebra; intersection is defined in terms of these.

The union operation returns a pair where the set of objects is, in general, heterogeneous and the set of message expressions is calculated as the intersection of the sets of message expressions of the operands. The heterogeneous set of objects is the union of the sets of objects of the operands. Formally:

Union (A,B)=<$T_{instances}$(A) ∪ $T_{instances}$(B), M_e(A) ∩ M_e(B)>

When required to be persistent in the lattice, the resulting pair has the characteristics of a class which is a superclass of both operands.

Example: Assume that the person class is not present in the lattice with student and staff classes defined as follows:

student<∅,name:string,date-of birth:date,year:integer,courses: course>

staff<∅,name:string,date-of-birth:date,salary:integer>

The person class is derived as: person:=Union(student, staff)

The result from this query is the person class.

Under the condition that M_e(A)-M_e(B) ≠∅, the difference operation has the following form:

Difference(A,B)=<{o | o ∈ $T_{instances}$(A) ∧ o ∉ $T_{instances}$(B)}, M_e(A)-M_e(B)>

However, if M_e(A)-M_e(B)=∅, then M_e(A)-M_e(B) is replaced by M_e(A) in the definition to get:

Difference(A,B)=<{o | o∈ $T_{instances}$(A) ∧ o ∉ $T_{instances}$(B)}, M_e(A)>

Example: Find students who are not research assistants:

Difference(student, research-assistant)

Since M_e(student)-M_e(research-assistant)=∅, in the output pair M_e(student) is returned. When required to be persistent in the lattice, the result of a difference operation is a superclass of the first operand.

In terms of the difference operation we define the intersection operation as follows:

Intersection(A,B)=Difference(A,Difference(A,B))

Finally, the inverse of the Project operation—I project—is defined in terms of other operations. To add a subset M of M_e(B) to M_e(A), first nest A and B, then do a one level projection to have all M_e(B) and M_e(A) together forming one set; then project on M_e(A) ∪ M to get the target set of message expressions in the resulting pair. Thus,

Iproject(A,B:M)=Project(OLproject(Nest(A,B), messages(A) ∪ (m messages(B))), M_e(A) ∪ M)

where M ⊆ M_e(B) is the set of message expressions to be added to M_e(A), and m is a message in the result of Nest(A,B) with its domain being $T_{instances}$(B). Notice that the OLproject operation results in a pair that contains M_e(A) ∪ M_eB). So, we use the project operation to get the required message expressions in the result.

The above formulation of the inverse project operation is valid for the case of adding some existing methods to a class. Should M consist of new methods, like M={m_1:f_1,m_2:f_2,...,m_n:f_n}, where m_i:f_i specifies that message m_i invokes the method that implements the function f_i, the definition is modified to:

A[M]=<$T_{instances}$(A), M_e(A) ∪ {m_1,m_2,...,m_n}>

Superiority of the Object Algebra over the Relational Algebra

It is important to emphasize that, since we have the five basic operators of relational algebra, the object algebra has at least the power of the relational algebra. In fact, object algebra is more powerful because the relational algebra handles only atomic domains and only stored values can be retrieved (which is nothing more than a restriction that leads to an embedded query language and hence impedance mismatch) in contrast to the object algebra, which handles stored as well as derived values and hence is computationally complete. Furthermore, the proposed model allows the use of set-based predicates, and predicates admit quantifiers in contrast to atomic predicates in relational algebra. Also, it supports encapsulation, object identity and inheritance. Generally speaking, the expressiveness of the constructors of an object-oriented data model effect the expressiveness of the corresponding query language; an object-oriented data model allows the definition of data through abstraction, supports derived data in addition to multivalued properties, and allows complex objects, identity and inheritance. As a result, the same real-world situation can be expressed more simply using an object-oriented schema than a relational schema. All queries that are coded using the relational algebra could be expressed using an object-oriented query language; however, the reverse is not true. Hence, an object-oriented query language is more expressive than the relational algebra for capturing the distinguishing properties of an object-oriented data model.

Concerning the nested relational algebra: although it handles non-atomic domains, it imposes the restriction of manipulating only stored values, which is equivalent to having only message expressions that return stored values, and excluding those that return derived values. Hence it does not overcome the impedance mismatch problem. Also, it does not support inheritance, neither identity nor encapsulation. In other words, the nested relational model aims to represent complex objects by nesting relations, but still they are value-based and record-oriented models.

Next we give the object algebra equivalents to the nest and unnest operations of the nested relational algebra. We assume that A has a set of attributes N_1 and consider every element of N_1 as a message that returns the corresponding stored value in a receiving object (a tuple in a nested relation). Furthermore, we assume $T_{\{instances\}}(A)$ is the same as the set of tuples in an equivalent nested relation and $M_e(A)$ has an equivalent calculation starting with attributes of A and combining with nested attributes. Now given $N \subseteq N_1$:

Nest:

(A,N)= Select(Nest(Project(A, $M_e(A)$-{x | x∈ $M_e(A)$∧ ∃m∈ N ∧ x=(m x_j)}),
Project(A, {x | x ∈ $M_e(A)$∧ ∃m∈ N∧ x=(m x_j)}))%s,
$\exists s_1 \in T_{instances}(A) \wedge s\ m_1() N = s_1 N \wedge$
$s\ (M_e(A)-N) = s_1\ (M_e(A)-N)$

where $m_1()$ is a message added to the result of the nest operation to facilitate the access of objects in the second operand.

Unnest:

(A,N)= OLproject(A messages(A)-{m2}∪ (m_2 messages(B)))

where messages(B) corresponds to the set of attributes N and $m_2 \in$ messages(A). The result of the execution of m_2 is in the domain $T_{instances}(B)$.

So, the one level project operation corresponds to a sequence of unnest followed by a projection in the nested relational model (Jaeschke & Schek, 1982; Abiteboul & Beeri, 1988; Colby, 1989). The one level project operation functions like the project and image operations described in Manola & Dayal (1986) and Zdonik (1988), the apply operation of Osborn (1988) and the map operation described in Straube & Ozsu (1990), but maintains the closure property without additional constructs.

The object algebra is in fact more powerful than the relational algebra since it can manipulate stored as well as derived values in addition to supporting the object-oriented features.

Handling Schema Evolution Functions Using the Object Algebra

A taxonomy of some schema evolution functions is found in (Alhajj, 1992; Banerjee, et al., 1987). In this section, we show how the basic schema evolution functions can be handled using the operators of the already presented object algebra.

1. Add an instance variable with domain c_1 to class c:
   c:=Nest(c, c_1)

   Notice that the result of the Nest operation is considered to be a subclass of the first operand c. However, the assignment is used to have this result replacing the class c itself. In this way, the instance variable iv_1 with domain(iv_1)=$T_{instances}(c_1)$ is added to $I_{\{variables\}}(c)$ (the instance variables of the class c).

2. Drop from $I_{variables}(c)$ the instance variable whose domain is specified as the class c_1 (the corresponding value in each object is handled via the message m()):
   c:=Project(c, messages(c)-{m()})

   where m() ∈ messages(c) and m() handles the value of the instance variable iv ∈ $I_{variables}(c)$ with domain (iv)=$T_{instances}(c_1)$

   It is an implementation issue to decide on whether the value of the deleted instance variable is to be physically dropped from the corresponding objects or not. In our model the only means that could be used to access the values constituting an object are the corresponding messages. Thus, after dropping the message that could access a certain value, it will be impossible to access that value inside any of the objects, although the value may still be present.

3. Add to the methods of class c_1 one or more methods from class c_2 with {m_1,m_2,...,m_i} being their corresponding messages:
   c1:=Iproject(c_1, c_2:{m_1,m_2,...,m_i}

   However, for the case where m_1, m_2, ..., m_i are new methods, the following formulation is valid:
   c_1:=Iproject(c_1, {m_1:f_1,m_2:f_2,...,m_i:f_i})

Example: Add to the staff class the method net-salary(i) which deducts taxes at the rate of i from the salary.

staff:=Iproject(staff({net-salary(i)(o,i)=o salary() * (1-i)})

The message net-salary(i) with $0 \le i \le 1$, could be used to invoke the new method added to the staff class to implement the function f(o,i) where o is an object variable bound to objects of the staff class, i.e., o ∈ $T_{instances}$(staff) and indicates the receiver of the message. This method is automatically implemented.

4. Drop one or more methods from class c_1, their corresponding messages being {m_1,m_2,...,m_i}:
   c_1:=Project(c_1, messages(c_1)-{m_1,m_2,...,m_i})

   In this way, all message expressions prefixed by a message drawn from the set {m_1,m_2,...,m_i} are dropped from $M_e(c_1)$. It is important to indicate that the instance variable deletion is recognized to be a special case of method dropping.

5. Add a class c to the lattice with the domains of its instance variables
   iv_1, iv_2,...,iv_n being $T_{instances}(c_1)$,$T_{instances}(c_2)$,...,$T_{instances}(c_n)$, respectively

   A new class may either have the OBJECT class as a direct superclass or have other existing class(es) in its superclass list. Furthermore, a new class may have zero, one or more subclasses. Thus,
   c:= Nest(OBJECT, Nest(c_1, ... Nest(c_{n-1}, c_n)...))

   The OBJECT class is used to have the new class c as a direct subclass of the root. If the class c is desired to be a direct subclass of an existing class, say c_p, object is replaced by the class c_p in the above formulation. All the example classes given could be defined through the use of this schema evolution function.

6. Drop an existing class c from the lattice:
   Let $T_{instances}(c_1)$,...,$T_{instances}(c_n)$ be the domains of the instance variables defined in class c without being inherited, with their corresponding messages being $m_1()$, $m_2()$, ..., $m_n()$.When all the

definition and contents of the class c are dropped, then class c is deleted and its immediate supers replace it in the inheritance mechanism. Thus,

c:=Project(c, {})

7. Add a class c_1 to the superclass list of the class c;
   c_1 has instance variables iv_1, iv_2, ..., iv_n with domains $T_{instances}(c_2)$,$T_{instances}(c3)$,...,$T_{instances}(c_n)$, respectively
   c:=Nest(c, Nest(c2, ... Nest($c_{[n-1}$, c_n) ...))
   c1:=Project(c, {m_1(), m_2(), ..., m_n()})

where {m_1(), m_2(),..., m_n()} are the messages corresponding to the instance variables of the class c_1. This is true when c_1 is a new class. However, when c_1 is an existing class, the following is done:

c:=OLproject(Nest(c,c1),messages(c)$\cup${m()messages(c_1)}))

where m() is the message added to Nest(c, c_1) and m() corresponds to the instance variable iv_1 with domain (iv_1)=$T_{instances}(c_1)$.

c1:=Project(c, messages(c_1))

8. Remove class c_1 from the superclass list of class c:
   c:=Project(c, Me(c)-$M_e(c_1)$)

Summary and Conclusions

An object algebra for object-oriented database systems has been described. A query is coded using the operators of the object algebra applied on some operands. An operand should have a pair of sets, a set of objects and a set of message expressions. Elements of the second set are used in the invocation of behavior as well as behavior constructors because a message expression leads to the execution of all the methods underlying the constituting messages and in the same order as if all together form a single method. Concerning the result of a query, it is a pair of sets, the same as those of the operands. So, the output of one query can be the input to another without any problems, and hence the closure property is maintained in a natural way. In producing the output pair of a query, the two constituting sets are derived in terms of those of the operand(s) and hence the operators act on behavior as well as on the state of objects. While doing this, heterogeneous sets are considered and this adds much to the power of the object algebra.

Message expressions deal with both stored and derived values and hence provide greater computational power to the user. This property is valid for the object algebra as a whole, where computed as well as stored values are manipulated. Therefore, not only is the object-oriented data model more powerful than the relational data model, the object algebra is also more powerful than the relational algebra. In supporting object construction, behavior construction via message expressions, and in dealing with the behavior as well the state of objects. Behavior manipulation is necessary in maintaining the encapsulation feature of object-oriented data models.

A query is handled on either a temporary or a persistent basis. Concerning the second case, for the output pair we derive the characteristics of a class and the inheritance relationship with other existing classes to place it in the lattice in a logical way, benefiting from reusability. As a consequence, we show how some schema changes can be handled using the object algebra.

Currently, we are examining the completeness of the described object algebra. Also, equivalences between different combinations of the operators and the use of equivalences in query optimization are under investigation.

References

Abiteboul, S. and Beeri, C. "On the Power of Languages for the Manipulation of Complex Objects," *INRIA, Tech. Rep. No. 846*, May.

Alashqur, A. Su, S.Y. and Lam, H. (1989). "OQL: A Query Language for Manipulating Object-Oriented Databases," *Proceedings of the15th International Conference on Very Large Databases*, Amsterdam, Holland, pp. 433-442.

Alhajj, R. (1992). "A Query Model and an Object Algebra for Object-Oriented Databases," Ph.D. Dissertation, Bilkent University, Turkey.

Alhajj, R. and Arkun, M.E. (1992). "A Schema Modification Methodology for an Object-Oriented Database System," *Proceedings of the XVIII Latin America Conference on Computers, PANEL'92.*

Atkinson, M. et al., (1989). "The Object-Oriented Database System Manifesto," *Proceeding of the International Conference on Deductive Object-Oriented Databases*, Kyoto, Japan.

Banerjee, J. et al., (1987) "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, pp. 3-26.

Banerjee, J. Kim, W. and Kim, K.C. (1988). "Queries in Object-Oriented Databases," *Proceedings of the 4th IEEE International Conference on Data Engineering*, Los Angeles, CA., pp. 31-38.

Carey, M.J., DeWitt, D.J. and Vandenberg, S.L. (1988). "A Data Model and a Query Language for EXODUS," *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Chicago,IL., pp. 413-423.

Cluet, S. and Delobel, C. (1992). "A General Framework for the Optimization of Object-Oriented Queries," *Proceedings of ACM-SIGMODInternational Conference on Management of Data.*

Colby, L. (1989). "A Recursive Algebra and Query Optimization for Nested Relations," *Proceedings of ACM-SIGMOD International Conferenceon Management of Data*, pp. 273-283.

Dayal, U. (1989). "Queries and Views in an Object-Oriented Data Model," *Proceedings of the Second International Workshop on Database Programming Languages*, pp. 80-102.

Jaeschke, G. and Schek, H.J. (1982). "Remarks on the Algebra of Non-First Normal Form Relations," *Proceedings of the Symposium on Principles of Database Systems*, pp. 127-138.

Kim, W. (1989). "A Model of Queries for Object-Oriented Databases," *Proceedings of the 15th International Confer-*

ence on Very Large Databases, Amsterdam, Holland, pp. 423-432.

Kim, W. (1990). "Object-Oriented Databases: Definition and Research Directions," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, pp. 327-341.

Maier, D. and Stein, J. (1987). "Development and Implementation of an Object-Oriented DBMS," *Research Directions in Object-Oriented Programming*, Shriver B. and Wegner, P., Eds., Cambridge: MIT Press.

Manola, F. and Dayal, U. (1986) "PDM: An Object-Oriented Data Model" *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, pp. 18-25,.

Osborn, S.L. (1988). "Identity Equality and Query Optimization," *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, Ebernburg, pp. 346-351.

Shaw, G. and Zdonik, S. (1990). "A Query Algebra for Object-Oriented Databases," *Proceedings of the 6th IEEE International Conference on Data Engineering*, Los Angeles, pp. 154-162.

Straube, D.D. and Ozsu, M.T. (1990). "Queries and Query Processing in Object-Oriented Database Systems," *ACM Transactions on Information Systems*, Vol. 8, No. 4, pp. 387-430.

Zdonik, S.B. (1988.) "Data Abstraction and Query Optimization," *Proceedings of the Second Workshop on Object-Oriented Database Systems*, Ebernburg, pp. 368-373.