A Query Model for Object-Oriented Databases

Reda ALHAJJ

Department of Computer Engineering
and Information Sciences
Bilkent University
Bilkent 06533, Ankara, TURKEY

M.Erol ARKUN

Department of Computer Engineering
and Information Sciences
Bilkent University
Bilkent 06533, Ankara, TURKEY

Abstract

A query language should be a part of any database system. While the relational model has a well defined underlying query model, the object-oriented database systems have been criticized for not having such a query model. One of the most challenging steps in the development of a theory for object-oriented databases is the definition of an object algebra. A formal object-oriented query model is described here in terms of an object algebra, at least as powerful as the relational algebra, by extending the latter in a consistent manner. Both the structure and the behavior of objects are handled. An operand and the output from a query in the object algebra are defined to have a pair of sets, a set of objects and a set of message expressions where a message expression is a valid sequence of messages. Hence the closure property is maintained in a natural way. In addition, it is proved that the output from a query has the characteristics of a class; hence the inheritance (sub/superclass) relationship between the operand(s) and the output from a query is derived. This way, the result of a query can be persistently placed in its proper place in the lattice.

Keywords: database system, object-oriented databases, query model, object algebra, query language.

1 Introduction

Object-oriented systems evolved to satisfy the demand for a more appropriate representation and modeling of real world entities. Such a demand comes mainly from data intensive applications including CAD/CAM, OIS and AI. To satisfy such kinds of applications, it was agreed that an integration of object-oriented concepts [18] with the database technology [14] leads to more appropriate representation methods and many object-oriented data models have been developed [10, 12, 16, 17, 21].

Comparing the relational model and an object-oriented model shows that the latter is more powerful at the modeling stage, but yet does not support a standard formal query model; one of the common complaints against object-oriented databases [23]. While the non-atomic domain concept is supported by the nested relational model [1, 25], we see inheritance, identity and encapsulation among the features that the relational model lacks. Identity provides for object sharing. Inheritance provides for structure and behavior sharing. Encapsulation provides for abstraction. As a result, an object-oriented query model should benefit from such features and hence should be at least as powerful as the relational query model.

It is true that object-oriented databases support implicit queries for simple operations, however a query language is required to be a part of any database system. For instance, the message **name()** when sent to an instance in the student class, the name of the particular student is returned. While a single message is sufficient for such an operation in the object-oriented context, a selection and a projection are necessary to get the same result in the relational model. An additional join should precede when name is not a column of the student relation. Another example can be seen in sending the message **courses()** to a student and the message **grade()** to the result obtained by the first message. Although it is handled due to the implicit join [20] present in object-oriented models, this corresponds to an explicit join in the relational model. The two messages **courses()** and **grade()** form a message expression. In general, a message expression is defined to be a valid sequence of messages $m_1 ... m_n$, with $n \geq 1$.

While message expressions give superiority to object-oriented systems over the relational model, an object-oriented query language is still needed for more complex situations and to support associative access. In other words, although the modeling power of an object-oriented database supports implicit joins [20] by allowing instances in a class to form the domain for an instance variable in another class, an explicit join is necessary in introducing new relationships into the model; otherwise the manipulation power of the model will be restricted. Allowing an explicit join raises the problem of maintaining the closure property. Therefore, it is necessary to have an object algebra that facilitates the introduction of new relationships and maintains the closure property; otherwise the relational model will be more powerful.

In this paper, we describe an object algebra for object-oriented data models [3, 4, 5, 6, 7, 8]. Our object algebra is a superset of the relational algebra, but with different semantics and operands. The main idea in our work is that an operator should equally handle objects as well as their behavior. So, an operand in our object algebra, as well as the output of any of the operations, has a pair of sets; a set of objects and a set of message expressions. The set of objects includes all objects that qualify to be in a class and in all of its direct and indirect subclasses; hence the set of objects is in general heterogeneous. The set of message expressions includes message expressions applicable to objects in the other set of the pair. By using such pairs as operands and in the output, the closure property is maintained in a consistent way.

The operators of our object algebra are the five basic operators of the relational algebra in addition to nest, one level project and aggregate function application. While the nest operation introduces a missing relationship into the model in a natural way, the one level project operation evaluates a subset of the message expressions of the operand against objects of the operand.

Using the object algebra operators, object algebra expressions are built and it is proved that every object algebra expression has the characteristics of a class. Moreover, the inheritance (sub/superclass) relationship between the result of an object algebra expression and the operand(s) is considered. Therefore, the result of any object algebra expression can be persistently and properly placed in the lattice in a natural way.

To sum up, the contributions of our work described in this paper can be enumerated as follows. Operands and the result of a query are defined in a way not to violate object-oriented constructs and to maintain the closure property. Behavior is equally handled as objects; creation of methods as well as objects in terms of other existing ones is facilitated. The addition of new classes is facilitated where we specify the characteristics of a class derived in terms of existing ones and handle its proper placement in the lattice. Aggregation functions are supported in a consistent way so that the result could be used as an operand. All of these are satisfied without loss of generality and formality in the description.

The rest of the paper is organized as follows. The related work is discussed in section 2. In section 3, the data model is described where the basic terminology used in the formalization is introduced. In section 4, the object algebra is defined by constructing object algebra expressions. Also, characteristics of the result of an object algebra expression are proved to be the same as the characteristics of a class and the relationship between an object algebra expression and the operand(s) is derived. Some illustrative examples are given in section 5. Section 6 is the conclusions.

2 Related Work

Several query languages such as those of GemStone [21], O_2 [13, 16], EXODUS [12, 30], IRIS [17], ORION [11, 20], *OSAM** [2], Postgres [26], PDM [15, 22], ENCORE [27] and the formal calculi and algebra developed by Straube and T. Özsu [29] in addition to others [9, 24] have been proposed.

These languages are classified as either preserving objects in the database [2, 11, 12, 21, 29] or providing operators for the creation of new objects [13, 15, 20, 22, 24, 27]. Such a distinction is due to the disagreement on whether it is possible to have all required relationships defined at the modeling phase. We and others, e.g., [24, 27], argue that the definition of new relationships and the creation of new objects, should be supported by a query model. However, it is necessary to resolve problems that arise due to the creation of objects; otherwise there will be inconsistencies. Among such problems is to maintain the closure property [2]. In other words, the output of a query should be allowed as an operand in the model.

A major drawback of languages such as those described in [11, 21, 29] is that they do not maintain the closure property. Others introduce non-object-oriented constructs in maintaining the closure property. Although operands in such languages have object-oriented properties, the outputs are relations which do not have the same structural and behavioral properties as the original objects. Consequently, the result of a query cannot be further processed by the same set of language operators without violating encapsulation, for example. For instance in O_2 [13, 16] the value concept was introduced. O_2 has an object algebra which handles values as well as objects and this leads to a kind of mismatch in having some operands violating encapsulation while others not. The query languages of [9, 12, 26] use nested relations as their logical view of object-oriented databases.

The Postgres data model is an extended relational data model which includes abstract data types, data of type procedures and attribute and procedure inheritance. Its query language POSTQUEL is an extension of QUEL to satisfy the new constructs.

The algebra described in [30] has an expressive power equivalent to the EXCESS query language of the EXTRA data model described in [12]. The algebra of PDM [15, 22] is based on an extension of the Daplex functional data model [28]. While Daplex supports only functions whose values are stored in the database, PDM has been extended to include functions whose values are derived from other values or computed by arbitrary procedures. PDM modifies the relational algebra to handle functions, i.e., the operators and the result are functions. A major restriction is that object identity is not supported and only union compatible items are allowed as operands to set-based operators. In the algebra of ENCORE [27], the output of a query is of the Tuple type which is essentially the nested relational representation, since it allows the nesting of tuples.

Straube and Özsu developed a set-based object-oriented query algebra and a corresponding calculus, but their algebra does not satisfy the closure property. Also, they studied the problem of type unions in some detail. However, although it has a formal basis, their algebra is less expressive compared to others described in the literature. Osborn's object algebra [24] was developed for a general object-oriented data model defined on three generic classes of atomic, aggregate and set objects. She extends relational algebra. A major drawback of Osborn's algebra is that it does not support encapsulation and the closure property is not well maintained; set operations do not accept atom and aggregate objects produced by other operations.

Although, in the query model of ORION [20] the result of a query operation is a class, but 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. However, we argue that it is an overhead to have a class as the output of a temporary query, as ORION does. In this paper we describe the output of a query by the minimum requirements of an operand and from such characteristics we can derive the characteristics of a class when it is required to have the result persistent [3, 4]. In *OSAM** operands in a query are the database itself and all subdatabases derived from the original database by query operations; the result of a query is a subdatabase.

3 The Data Model

The object algebra described in this paper is based on a data model that includes classes, objects and methods. A class definition includes a set of instance variables that reflects properties of objects in the class, a set of methods (operations) applicable to objects in the class, to support encapsulation and information hiding, and a set of superclasses to provide reusability. Related to a class c we use the following notations:-

- instances(c) is the set of objects in class c but not in any of its subclasses.
- $T_{instances}(c)$=instances(c)$\bigcup_{i=1}^{card(S)} T_{instances}(S_i)$ where $S = \{S_1, S_2, ..., S_{card(S)}\}$ is the set of direct subclasses of class c.
- $I_{variables}(c)$ is the set of all instance variables defined in or inherited by class c. For any instance variable *iv*, domain(*iv*) and value(*iv*) denote the domain and the value of instance variable *iv*. A domain is either atomic such as the set of integers, the set of characters, etc, or $T_{instances}(c_i)$ for any class c_i. A value is drawn from the underlying domain; either an element or a subset [*] of the underlying domain.
- messages(c) is the set of messages used to invoke any of the methods defined in or inherited by class c.

Elements of messages(s) are used only to invoke methods in class c. When the result is an object o_i, messages in the class of object o_i are used to invoke methods applicable to it. So combining from class c a message which returns an object o_i as a result with any of the messages in the class of object o_i will form pairs applicable to objects in class c to access possible values in related objects from the class of object o_i. Also when any of such pairs returns an object as a result, messages in the class of the latter object could be combined with that pair forming triples applicable to objects in class c. By the same way, quadruples, quintuples and so on, could be formed. For instance, let o_1 be an object in the *student* class; a method in the *student* class could be *courses*() to invoke the method implemented to return the set of courses registered by a given student and so o_1 *courses*() returns objects from the *course* class. Any of the messages in the *course* class, e.g. *code*(), could be applied to a returned object. At this point one could say that the combination *courses*() *code*() could be applied to an object in the *student* class. It is recognized that both *courses*() and *courses*() *code*() are elements of the superset of messages(*student*) which does not include the element *courses*() *code*(). We call such a superset the set of message expressions of class *student* and every element of this set is called a *message expression*. $M_e(c)$ is the set of message expressions of class c. Every element of $M_e(c)$ returns either a stored value or a derived value. As formally stated in the following definition, elements of $M_e(c)$ are recursively defined in terms of messages, starting with messages(c).

[*]since ϕ is subset of any set, *nil* is a value representing the empty set

Definition 3.1 (Message expressions)
Given a class c, the set $M_e(c)$ is defined by:
- *messages(c)$\subseteq M_e(c)$*
- *if $x \in M_e(c)$ and x returns a value from $T_{instances}(c_1)$ then (x messages(c_1))[†] $\subseteq M_e(c)$*

Therefore, starting from messages(c) we can determine elements of $M_e(c)$. □

We use len(x) to denote the length of message expression x, i.e., the number of messages constituting x.

After introducing message expressions, it is necessary to decide on the relationship between the sets of message expressions and the sets of messages of two classes.

Lemma 3.1 *Given two classes c_1 and c_2 $M_e(c_1) \subseteq M_e(c_2) \Longleftrightarrow$ messages(c_1)$\subseteq$messages(c_2), i.e., $\forall x \in M_e(c_1)$ such that len(x)=1 we have $x \in M_e(c_2)$.* □

Lemma 3.1 will be utilized while constructing object algebra expressions in definition 4.2 and while deciding on the inheritance relationship between classes that correspond to object algebra expressions in section 4. A message expression when received by an object, returns a value from a particular domain. This particular domain is the range of the last message in the message expression. A returned value is either a stored or a derived value, a property that gives a full computational power to the user without having an embedded query language leading to impedance mismatch.

Related with the subclass/superclass relationship between classes, we define a partial ordering ($\leq_c$) among classes.

Definition 3.2 *(Partial ordering ($\leq_c$) among classes)*
Given two classes c_1 and c_2, we say that $c_1 \leq_c c_2$ iff:

- *$I_{variables}(c_2) \subseteq I_{variables}(c_1)$*
  i.e., $\forall iv_2 \in I_{variables}(c_2)\ \exists iv_1 \in I_{variables}(c_1)$ such that, $iv_2 = iv_1 \wedge$ (domain(iv_1)$\leq_c$domain(iv_2) $\vee$ domain(iv_2)=domain(iv_1))
- *methods(c_2) $\subseteq$ methods(c_1)* □

An object has an identity, a value and belongs to a certain class. Related to an object o we use value(o) to denote the value (The value of an object is a set of values of the instance variables defined in its class; simple values or identities of nested objects). Similarly, *identity*(o) denotes the identity of object o. Based on the notion of *value* and *identity* we define equality of objects:

Definition 3.3 (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.* □

[†]x is concatenated with every element of the set of messages of class c_1. For example, $(x\ \{m_1, m_2\}) = \{x_{11}, x_{21}\}$ where $x_{11} = (x\ m_1)$ and $x_{21} = (x\ m_2)$

A method implements a certain function and has a number of arguments, $n \geq 0$. Every method is invoked via a corresponding message. We address properties of an object by using messages. Therefore, methods are used either to deal with properties of objects, stored values, or to derive some values in terms of properties of objects. For instance, the method invoked by the message *name*() implements the function

$f_1 : T_{instances}(person) \longrightarrow string.$

Function f_1 does not expect any argument because corresponding domains are not specified. The message *increase-salary(i)* invokes the method implementing the function

$f_2 : T_{instances}(staff) \times integer \longrightarrow integer,$

where given $o \in T_{instances}(staff)$,

$f_2(o, i) = (o\, salary()) + i$

The domain of the receiver of f_2 is $T_{instances}(staff)$ and f_2 expects a single argument from the domain that is the set of integers. Also, the result of f_2 is from the set of integers, i.e., range of f_2 is the set of integers.

4 The Object Algebra

In this section, the object algebra is described. An operand e in the object algebra should have a pair of sets, a set of objects and a set of message expressions, denoted by $<T_{instances}(e), M_e(e)>$; elements of $T_{instances}(e)$ can be accessed using elements of $M_e(e)$. Since a class has a defined set of objects and a derived set of message expressions, a class can be an operand. The output of an operation as well should have a pair of sets derived in terms of the pair(s) of operand(s). Thus, an operand in a query could be replaced by another query whose output is the actual operand. Any operand, whether an actual pair or an unevaluated query is called an *object algebra expression*.

Concerning the operators, the object algebra includes the five basic operators of the relational algebra in addition to nest, one level project and aggregate function applications. The selection operation presents a restriction on objects of the operand. In the object algebra, the selection has a single operand and produces an output consisting of a pair, where the included objects are those satisfying a stated predicate expression, defined next. The set of message expressions of the resulting pair is the same as that of the operand.

Definition 4.1 (Predicate expressions)
The following are predicate expressions:

P1: *T and F are predicate expressions representing true and false.*

P2: *Given two values y_1 and y_2 with the same underlying domain such that at least y_1 or y_2 is of the form (ox), 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 o.*

P2.1: $y_1 \; op \; y_2$ *is a predicate expression where,*

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

P2.2: $\forall/\exists z \in y_1 \wedge z \; op \; y_2$ *is a predicate expression where, y_1 is a set of values and*

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

P2.3: $\exists z \subseteq y_1 \wedge z \; op \; y_2$ *is a predicate expression where, y_1 is a set of values and*

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

P3: *if p and q are predicate expressions then (p), $\neg p$, $p \wedge q$ and $p \vee q$ are predicate expressions.*$\square$

Let s_1 and s_2 be object variables ranging over instances of the *student* class: $"CS590" \in s_1\, courses()\, code()$ is an example on P2.1 to check students attending $"CS590"$; $\exists c \in s_1\, courses() \wedge c \in s_2\, courses() \wedge s_1 \neq s_2$ is an example on P2.2 to check whether two given students have at least one course in common; $\forall c \in s_1\, courses() \wedge c \notin s_2\, courses()$ is an example on P2.2 to check whether two given students do not have any course in common; $\exists c \subseteq s_1\, courses() \wedge c \subseteq s_2\, courses()$ is an example on P2.3 to check whether two given students have some courses in common.

Although the set of objects of an operand is in general heterogeneous, the only values accessible in each object are those specified by the set of message expressions of the pair. So, dropping some message expressions by the project operation hides some values from the accessible objects. The inverse of the project operation is to extend the set of message expressions in a pair to include more message expressions applicable to objects of the pair, i.e., give more facilities to the user; this operation is defined in terms of others as shown later in this section. On the other hand, the one level project operation evaluates a provided set of message expressions and forms objects out of the obtained values; a corresponding set of message expressions is also determined to facilitate accessing the values encapsulated within the derived objects.

Despite the fact that many relationships between objects are represented by the objects themselves, an explicit operation is required to handle cases when a relationship is not defined in the model. Both the cross-product and the nest operations are defined to introduce such relationships. While the cross-product operation is defined to be associative, the nest operation is not. However, the two operations are equivalent under certain conditions [5]; in [5] we also present the equivalence of some object algebra expressions. Associativity of the cross-product operation is useful in query optimization [3, 5], although not discussed in this paper. The cross-product operation creates new objects, out of objects in the operands, and a set of message expressions to handle the new objects is derived. Also, the nest operation introduces missing relationships. While the nest operation extends the value of each object in the first operand to include a reference to object(s) in the second operand, the result of the cross-product operation depends on domains of the messages of the operands as explicitly stated in definition 4.2 given next in this section.

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 first set. So as we deal with sets, two basic set operations, union and difference, are supported by the object algebra; intersection is defined in terms of the difference operations. 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. The difference operation is handled in one of two ways depending on the relationship between the sets of message expressions of the operands. If the set of message expressions of the first operand is subset from that of the second operand, the difference operation returns objects from the first operand which are not in the second operand. Otherwise, it is handled as a projection of objects in the first operand on values that have no corresponding message expression in the second operand.

After this informal description of the object algebra, we move into the formal definition. Since a class is defined to have a set of objects and a set of message expressions can be derived for a class by definition 3.1, a class is an object algebra expression. Next we formally define object algebra expressions. When speaking about len(x) in any of the constraints (if-statements) given next in this section, we will consider only message expressions x such that x returns a stored value with the underlying domain being atomic.

Definition 4.2 (Object Algebra Expressions) *Let E be the set of object algebra expressions.*
Being an object algebra expression, every element of the set E must have a pair of sets -a set of objects and a set of message expressions. Thus, formally speaking, $\forall e \in E$, $M_e(e)$ is defined and $T_{instances}(e)$ is defined.
Given $e_1 \in E$ and $e_2 \in E$; let $M_e(e_1)=X_1$, $M_e(e_2)=X_2$, $T_{instances}(e_1)=T_1$, and $T_{instances}(e_2)=T_2$
Elements of E are enumerated as follows:

- *Given a class c_i, by definition $M_e(c_i)$ and $T_{instances}(c_i)$ are both defined, then $c_i \in E$*
- *Selection: Given a predicate expression p, $e_1[p] \in E$ with*
  $M_e(e_1[p])=M_e(e_1)=X_1$
  $T_{instances}(e_1[p])=\{o \mid o \in T_1 \wedge p(o)$ [‡]$\}$
- *Projection: Given $X \subseteq X_1$, $e_1[X] \in E$ with*
  $M_e(e_1[X])=X$
  $T_{instances}(e_1[X])=T_{instances}(e_1)$
- *Cross-Product: $(e_1 \times e_2) \in E$ with,*

$$M_e(e_1 \times e_2) = \begin{cases} (m_1\ X_1) \cup (m_2\ X_2) & \text{if } \exists x_i \in X_1, len(x_i)=1, \\ & \quad \exists x_j \in X_2, len(x_j)=1 \\ X_1 \cup (m_2\ X_2) & \text{if } \forall x_i \in X_1, len(x_i)>1, \\ & \quad \exists x_j \in X_2, len(x_j)=1 \\ (m_1\ X_1) \cup X_2 & \text{if } \exists x_i \in X_1, len(x_i)=1, \\ & \quad \forall x_j \in X_2, len(x_j)>1 \\ X_1 \cup X_2 & \text{if } \forall x_i \in X_1, len(x_i)>1, \\ & \quad \forall x_j \in X_2, len(x_j)>1 \end{cases}$$

where m_1 and m_2 are two messages with T_1 and T_2 being the domains of the results of m_1 and m_2, respectively.

$$T_{instances}(e_1 \times e_2) = \begin{cases} \{o \mid \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = \\ \quad identity(o_1).identity(o_2)\} \\ \quad \text{if } \exists x_i \in X_1, len(x_i)=1 \wedge \\ \quad \exists x_j \in X_2, len(x_j)=1 \\ \{o \mid \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = \\ \quad value(o_1).identity(o_2)\} \\ \quad \text{if } \forall x_i \in X_1, len(x_i)>1 \wedge \\ \quad \exists x_j \in X_2, len(x_j)=1 \\ \{o \mid \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = \\ \quad identity(o_1).value(o_2)\} \\ \quad \text{if } \exists x_i \in X_1, len(x_i)=1 \wedge \\ \quad \forall x_j \in X_2, len(x_j)>1 \\ \{o \mid \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = \\ \quad value(o_1).value(o_2)\} \\ \quad \text{if } \forall x_i \in X_1, len(x_i)>1 \wedge \\ \quad \forall x_j \in X_2, len(x_j)>1 \end{cases}$$

where . is being used to indicate a concatenation of the two arguments; it is commutative because the resulting value is actually a set of values constructed out of the values constituting the two arguments.

- *Union: $(e_1 \cup e_2) \in E$ with*
  $M_e(e_1 \cup e_2)=X_1 \cap X_2$
  $T_{instances}(e_1 \cup e_2)=T_1 \cup T_2$
- *Difference: $(e_1 - e_2) \in E$ with*

$$M_e(e_1 - e_2) = \begin{cases} X_1 & \text{if } X_1 \subseteq X_2 \text{ (by lemma 3.1)} \\ X_1 - X_2 & \text{otherwise} \end{cases}$$

$$T_{instances}(e_1 - e_2) = \begin{cases} T_1 - T_2 & \text{if } X_1 \subseteq X_2 \\ T_1 & \text{otherwise} \end{cases}$$

[‡]Given an object o, we use $p(o)$ to denote the evaluation of predicate expression p by o substituting an object variable in p.

- *Nest:* $(e_1>>e_2)\in E$ *with*
  $M_e(e_1>>e_2)=X_1\cup(m_2\ X_2)$, *where* T_2 *is the domain of the result of message* m_2.
  $T_{instances}(e_1>>e_2)=\{o\,|\,\exists o_1\in T_1 \wedge\ value(o)= value(o_1).v,\ where\ v=(o\ m_2)\wedge v\in T_2\}$
- *One level project: Given* $X\subseteq X_1$, $e_1![X]\in E$ *with*
  $M_e(e_1![X])=\{x|\exists x_1\in X,\ x_1=(x_2\ m)\wedge len(x_1)= len(x_2)+1\wedge\exists x_3\in X_1\wedge x_3=(x_2\ x)\wedge x=(m\ x_4)\}$
  $T_{instances}(e_1![X])=\{o|\exists o_1\in T_1\wedge value(o)=(o_1\ X)$[§]$\}$
  The depth of nesting decreases as the length of the longest message expression in X increases. In other words, the depth of nesting is inversely proportional to the length of message expressions in X.
- *Aggregation: Given* $X\subseteq X_1$ *and* $x_i\in X_1$, $e_1<X,f,x_i>\in E$ *with*
  $M_e(e_1<X,f,x_i>)=(m_1\ X_1)\cup\{m_3\}$, *where* T_1 *is the domain of the result of message* m_1, *and the domain of the result of the function f is the domain of the result of message* m_3.
  $T_{instances}(e_1<X,f,x_i>)=\{o|(o\ m_1)\subseteq T_1\wedge(o\ m_3)= f(\{(o_1\ x_i)|o_1\in T_1\wedge\forall o_2\in(o\ m_1),\ (o_2\ X)=(o_1\ X)\})\}$
  The aggregation function is applied on e_1 *by evaluating the function f on the result of the message expression* x_i *for all objects that return the same values for elements of the set of message expressions X.*
- *Unnest: defined in terms of projection as,*
  $$(e_1<<e_2)=e_1[X_1-X\ |\ X=(m_2\ X_2)\wedge \forall o_1\in T_1,\ (o_1\ m_2)\in T_2]$$
  We project on all message expressions of e_1 *except those leading to* e_2.
- *Intersection: defined in terms of the difference operation as,* $(e_1\cap e_2)=e_1-(e_1-e_2)$
- *Inverse project: to add a subset X of* $M_e(e_2)$ *to* $M_e(e_1)$, *first* e_1 *and* e_2 *are nested then a one level projection is done to have all* $M_e(e_2)$ *and* $M_e(e_1)$ *together forming one set; after that projection of the result on* $M_e(e_1)\bigcup X$ *is done to get the target set of message expressions in the resulting pair.*
  $$e_1]X[=(e_1\gg e_2)![messages(e_1)\bigcup(m_2\ messages(e_2))] [M_e(e_1)\bigcup X]$$
  where $X\subseteq M_e(e_2)$ *is the set of message expressions to be added to* $M_e(e_1)$, *and* m_2 *is a message in the result of* $e_1>>e_2$ *with its domain being* $T_{instances}(e_2)$.
- *Join: defined in terms of cross-product or nest combined with selection,*
  $$e_1<p>e_2=e_1\times e_2\,[p]=e_1>>e_2\,[p]. \qquad \Box$$

Using operations of the query language, objects may be constructed out of existing ones and new relationships may be introduced into the model. A new relationship is an extension to either the state of objects or their behavior. In other words, a new relationship has either a stored or a derived value. A stored value is due to the Nest operation which takes two operands and extends each object in the first to include a value referencing object(s) in the second operand, while a derived value is due to the inverse of the Project operation which extends the behavior of objects in the operand without their states being affected. On the other hand, the One-Level-Project operation constructs new objects out of existing objects by collecting values found at different levels of nestings. Also the fourth case in the definition of the Cross-Product operation results in new objects, while other cases introduce new relationships.

After the formal definition of object algebra expressions, we claim that every object algebra expression has the characteristics of a class and this follows from the lemmas given next in this section. However, before going into the details of the lemmas, it is important to remind the reader that, as stated in section 3, by definition a class has a set of superclasses, a set of instance variables, a set of methods and a set of objects. According to definition 4.2, an object algebra expression has a set of objects and a set of message expressions. In addition, given a class c, methods(c) and $I_{variables}(c)$ are defined to include methods and instance variables of superclasses of class c. Therefore, finding methods and instance variables of a class implicitly leads to the set of its superclasses. Furthermore, for every method there exists a corresponding message; so, finding a set of messages for an object algebra expression is equivalent to finding of a set of methods. As a result, for any object algebra expression to have the characteristics of a class, it is enough to find for that object algebra expression a set of instance variables and a set of messages; a set of objects is already defined.

Let e_1 and e_2 be two object algebra expressions such that $M_e(e_1)=X_1$ and $M_e(e_2)=X_2$. According to definition 4.2, a class is an object algebra expression. In other words, some object algebra expressions are classes. Thus, assume that $I_{variables}(e_1)$, $I_{variables}(e_2)$, messages(e_1) and messages(e_2) are all defined. Based on this assumption, we have the following lemmas, 4.1 to 4.8, leading to the sets of messages and instance variables of other object algebra expressions and this leads to the fact that every object algebra expression corresponds to a class.

Lemma 4.1 *Messages and Instance variables of* $e_1[p]$*: where p is a predicate expression*
$M_e(e_1[p])=X_1 \Longrightarrow$. $messages(e_1[p])=messages(e_1)$
. $I_{variables}(e_1[p])=I_{variables}(e_1)$ $\Box$

Before going into the lemma 4.2 on the Project operation, the following algorithm returns the instance variables of $e_1[X]$ where $X\subseteq X_1$.

Algorithm 4.1 *Instance variables of* $e_1[X]$*:*

```
0. for every $m_i \in messages(e_1)$
1.   Let $X_i \subseteq M_E$[¶] such that $(m_i\ X_i) \subseteq X$
2.     if $X_i \neq \phi$ then
3.       if $\exists iv_i \in I_{variables}(e_1)$ such that
               $X_i = M_e(OAE(domain(iv_i)))$[‖] then
4.           $iv_i \in I_{variables}(e_1[X])$
```

[§]$(o_1\ X)$ returns the set of the results of the application of message expressions in X to object o_1.

[¶]Set of all message expressions, i.e., for any class c, $M_e(c)\subseteq M_E$

[‖]Evaluating an object algebra expression e leads to the pair $<T_{instances}(e), M_e(e)>$. OAE($T_{instances}(e)$) denotes the object algebra expression e.

```
5.        elseif $\exists iv_i \in I_{variables}(e_1)$ such that
              $X_i \subseteq M_e(OAE(domain(iv_i)))$ then
6.            $iv_i \in I_{variables}(e_1[X])$ and
              $domain(iv_i)$ in $e_1[X]$ is:
7.            $domain(iv_i) := < domain(iv_i),$
              $M_e(OAE(domain(iv_i))) > [X_i]$
8.        endif
9.    elseif $\exists iv_i \in I_{variables}(e_1)$ such that
          $value(iv_i) = (o\ m_i)$ then
10.           $iv_i \in I_{variables}(e_1[X])$
11.   endif
12. endfor
```
□

Lemma 4.2 *Messages and Instance variables of* $e_1[X]$*: Given* $X \subseteq X_1$,
. $messages(e_1[X]) = \{m \mid m \in messages(e_1) \wedge \exists x \in X \text{ with } x = m\ x_i\}$
. $I_{variables}(e_1[X])$ *is derived in algorithm 4.1.* □

Lemma 4.3 *Messages and Instance variables of* $e_1 \times e_2$ *:*

1: *if* $\exists x_1 \in X_1, len(x_1)=1 \wedge \exists x_2 \in X_2, len(x_2)=1$ *then*
$M_e(e_1 \times e_2) = (m_1\ X_1) \bigcup (m_2\ X_2) \Longrightarrow$
. $messages(e_1 \times e_2) = \{m_1, m_2\}$
. $I_{variables}(e_1 \times e_2) = \{iv_1, iv_2\}$,
where $domain(iv_1) = 2^{T_{instances}(e_1)}$ *and* $domain(iv_2) = 2^{T_{instances}(e_2)}$

2: *if* $\forall x_1 \in X_1, len(x_1) > 1 \wedge \exists x_2 \in X_2, len(x_2)=1$ *then*
$M_e(e_1 \times e_2) = X_1 \bigcup (m_2\ X_2) \Longrightarrow$
. $messages(e_1 \times e_2) = messages(e_1) \bigcup \{m_2\}$
. $I_{variables}(e_1 \times e_2) = I_{variables}(e_1) \bigcup \{iv_2\}$,
where $domain(iv_2) = 2^{T_{instances}(e_2)}$.

3: *if* $\exists x_1 \in X_1, len(x_1)=1 \wedge \forall x_2 \in X_2, len(x_2) > 1$ *then*
$M_e(e_1 \times e_2) = (m_1\ X_1) \bigcup X_2 \Longrightarrow$
. $messages(e_1 \times e_2) = \{m_1\} \bigcup messages(e_2)$
. $I_{variables}(e_1 \times e_2) = \{iv_1\} \bigcup I_{variables}(e_2)$,
where $domain(iv_1) = 2^{T_{instances}(e_1)}$.

4: *if* $\forall x_1 \in X_1, len(x_1) > 1 \wedge \forall x_2 \in X_2, len(x_2) > 1$ *then*
$M_e(e_1 \times e_2) = X_1 \bigcup X_2 \Longrightarrow$
. $messages(e_1 \times e_2) = messages(e_1) \bigcup messages(e_2)$
. $I_{variables}(e_1 \times e_2) = I_{variables}(e_1) \bigcup I_{variables}(e_2)$ □

Lemma 4.4 *Messages and Instance variables of* $e_1 \bigcup e_2$*:*
$M_e(e_1 \bigcup e_2) = X_1 \bigcap X_2 \Longrightarrow$
. $messages(e_1 \bigcup e_2) = messages(e_1) \bigcap messages(e_2)$
. $I_{variables}(e_1 \bigcup e_2) = I_{variables}(e_1) \bigcap I_{variables}(e_2)$ □

Lemma 4.5 *Messages and Instance variables of* $e_1 - e_2$*:*

1: - *if* $X_1 \subseteq X_2$ *then*
$M_e(e_1 - e_2) = X_1 \Longrightarrow$
. $messages(e_1 - e_2) = messages(e_1)$
. $I_{variables}(e_1 - e_2) = I_{variables}(e_1)$

2: - *if* $X_1 \not\subseteq X_2$ *then*
$M_e(e_1 - e_2) = X_1 - X_2 \Longrightarrow$
. $messages(e_1 - e_2) = messages(e_1) - messages(e_2)$
. $I_{variables}(e_1 - e_2) = I_{variables}(e_1) - I_{variable}(e_2)$ □

Lemma 4.6 *Messages and Instance variables of* $e_1 >> e_2$*:*
$M_e(e_1 >> e_2) = X_1 \bigcup (m_2\ X_2) \Rightarrow$
. $messages(e_1 >> e_2) = messages(e_1) \bigcup \{m_2\}$
. $I_{variables}(e_1 >> e_2) = I_{variables}(e_1) \bigcup \{iv_1\}$,
where $domain(iv_1) = 2^{T_{instances}(e_2)}$. □

Lemma 4.7 *Messages and Instance variables of* $e_1![X]$*: given* $X \subseteq X_1$,
$M_e(e_1![X])$ *given in definition 4.2* $\Longrightarrow$
.$messages(e_1![X]) = \{m \mid \exists x \in M_e(e_1![X]) \text{ with } x = m\ x_j\}$
.$I_{variables}(e_1![X]) = \{iv \mid domain(iv) = 2^{d_i} \wedge \forall o \in T_{instances}(e$
$\exists m_i \in messages(e_1![X]) \text{ with } (o\ m_i) \in d_i\}$ □

Lemma 4.8 *Messages and Instance variables of* $e_1 < X, f, x_i >$*: given* $X \subseteq X_1$ *and* $x_i \in X_1$,
$M_e(e_1 < X, f, x_i >)$ *given in definition 4.2* $\Longrightarrow$
. $messages(e_1 < X, f, x_i >) = \{m_1, m_3\}$
. $I_{variables}(e_1 < X, f, x_i >) = \{iv_1, iv_2\}$
where $domain(iv_1) = T_{instances}(e_1)$ *and*
$domain(iv_2) =$ *the domain of the result of* f □

The proofs of lemmas 4.1 to 4.8 are omitted. Informally, since every object algebra expression has a set of message expressions, then by considering message expressions of length one, the set of messages is derived. Furthermore, by definition every instance variable has a corresponding message and this leads to the derivation of the set of instance variables of an object algebra expression depending on its set of messages, i.e., collect from the set of instance variables of the operand those instance variables having a corresponding message in the determined set of messages.

Combining definition 4.2 and lemmas 4.1 to 4.8, every object algebra expression has a set of objects, a set of messages and a set of instances variables; the set of superclasses of the corresponding class is determined by lemmas 4.9 to 4.16 given next this section. The set of messages leads to the set of methods because every message has a corresponding method. Therefore, an object algebra expression has the charactersitics of a class leading to the following corollary.

Corollary 4.1 $\forall e \in E$*, e corresponds to a class c.* □

After having every object algebra expression to be a class, it is necessary to decide on the inheritance relationship between an object algebra expression and other existing classes.

Given two object algebra expressions e_1 and e_2; let $M_e(e_1) = X_1$ and $M_e(e_2) = X_2$. Lemmas 4.9 to 4.16 give the inheritance relationship between object algebra expressions.

Lemma 4.9 *Inheritance relationship of* $e_1[p]$ *with* e_1*, where p is a predicate expression,*
$$e_1[p] \leq_e e_1$$ □

Lemma 4.10 *Inheritance relationship of* $e_1[X]$ *with* e_1*, where* $X \subseteq X_1$,
$$e_1 \leq_e e_1[X].$$ □

Lemma 4.11 *Inheritance relationship of* $e_1 \times e_2$ *with* e_1 *and* e_2*:*

1: if $\exists x_1 \in X_1, len(x_1)=1 \wedge \exists x_2 \in X_2, len(x_2)=1$ *then*
$(e_1 \times e_2) \not\leq_e e_1$ *and* $(e_1 \times e_2) \not\leq_e e_2$

2: if $\forall x_1 \in X_1, len(x_1)>1 \wedge \exists x_2 \in X_2, len(x_2)=1$ *then*
$(e_1 \times e_2) \leq_e e_1$

3: if $\exists x_1 \in X_1, len(x_1)=1 \wedge \forall x_2 \in X_2, len(x_2)>1$ *then*
$(e_1 \times e_2) \leq_e e_2$

4: if $\forall x_1 \in X_1, len(x_1)>1 \wedge \forall x_2 \in X_2, len(x_2)>1$ *then*
$(e_1 \times e_2) \leq_e e_1$ *and* $(e_1 \times e_2) \leq_e e_2$ $\square$

Lemma 4.12 *Inheritance relationship of* $e_1 \bigcup e_2$ *with* e_1 *and* e_2*:*
$e_1 \leq_e (e_1 \bigcup e_2)$ *and* $e_2 \leq_e (e_1 \bigcup e_2)$. $\square$

Lemma 4.13 *Inheritance relationship of* $e_1 - e_2$ *with* e_1 *and* e_2*:*
1: - if $X_1 \subseteq X_2$ *then*
$(e_1 - e_2) \leq_e e_1$
2: - if $X_1 \not\subseteq X_2$ *then*
$e_1 \leq_e (e_1 - e_2)$ $\square$

Lemma 4.14 *Inheritance relationship of* $(e_1 >> e_2)$ *with* e_1,
$(e_1 >> e_2) \leq_e e_1$ $\square$

Lemma 4.15 *Inheritance relationship of* $e_1 ![X]$ *with* e_1*, where* $X \subset X_1$*,*
$e_1 ![X] \not\leq_e e_1$ *and* $e_1 \not\leq_e e_1 ![X]$. $\square$

Lemma 4.16 *Inheritance relationship of* $e_1 < X, f, x_i >$ *with* e_1*, where* $X \subseteq X_1$ *and* $x_i \in X_1$
$e_1 < X, f, x_i > \not\leq_e e_1$ *and* $e_1 \not\leq_e e_1 < X, f, x_i >$ $\square$

When no superclass is determined, the root OBJECT class is assumed. Although omitted, the proofs of lemmas 4.9 to 4.16 follow from definitions 4.2 and lemmas 4.1 to 4.8.

5 Illustrative Examples

In this section, several examples are included to illustrate the distinguishing aspects of the query model presented in section 4. The examples given next in this section will assume the following classes:

$person < \emptyset, name : string, age : integer,$
$sex : character, children : \{person\} >$
$student < \{person\}, year : integer, courses : \{course\},$
$student\text{-}in : department >$
$staff < \{person\}, \text{salary:integer}, works\text{-}in\text{:department} >$
$research\text{-}assistant < \{student, staff\} >$
$course < \emptyset, code : string, name : string,$
$credit : integer, prerequisites : \{course\} >$
$department < \emptyset, name : string, head : staff >$

Example 5.1 *Find students attending the course "CS565"*

$S_1 = student\%s\ [''CS565'' \in s\ courses()\ code()]$ *where % indicates that the variable s is bound to and ranges over the objects of the operand, here the student class. In the predicate expression, "CS565"* $\in$ *s courses() code(), the right hand side is of the form (o x); hence satisfies definition 4.1. The use of =, calls for an evaluation of this query on a temporary basis.*

We differentiate between temporary and persistent evaluations of a query, where an assignment free query is always evaluated on a temporary basis while we use = and := to differentiate between temporary and persistent based evaluations, respectively. 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 determined pair by using lemmas 4.1 to 4.16.

Example 5.2 *Find the spouse of "Smith".*
$person\%p[\exists p_1 \in T_{instances}(person) \wedge p_1\ name() = ''Smith'' \wedge p\ sex() = ''F'' \wedge p_1\ chidlren() = p\ children()]$

Example 5.3 *Assume that the student class were not present in the lattice and the research-assistant class is defined as:*
$research\text{-}assistant < \{staff\}, year\text{:}integer, courses\text{:}course >$
To derive the student class as a persistent class and assuming that a student attends the department he works for, the research-assistant class is projected on {name(), age(), sex(), children(), year(), courses(), works-in()→student-in()}, where works-in()→student-in() indicates message renaming. In the projection set, the subset {name(), age(), sex(), children()}, could be replaced by messages(person) because the latter is the implicit representation of the former. Thus, the query is:
$student := research\text{-}assistant[messages(person) \bigcup$
$\{year(), courses(), works\text{-}in() \rightarrow student\text{-}in()\}]$
According to lemma 4.10, the derived student class will be a direct superclass of the research-assistant class. However, we have derived algorithms which aim at maximizing reusability [3] and accordingly, the derived student class is recognized as a subclass of the person class and naturally placed in the lattice.

Example 5.4 *Find the names and courses of students attending at least one course*
$student\%s[s\ courses() \neq \phi]![\{name(), courses()\ code()\}]$
First students attending some courses are selected, then the one level project is performed to get the result. Notice the use of the message expression, courses() code(), which is a concatenation of two messages, one from each of student and course classes, respectively.

Example 5.5 *Find couples having at least one child.*
$person\%p_1 >> person\%p_2\ [p_1\ sex() = ''M'' \wedge p_2\ sex() = 'F' \wedge p_1\ children() \neq \phi \wedge p_1\ children() = p_2\ children()]$

Example 5.6 *Find students attending the department in which "Adams" is working.*
$student\%s_1 \gg staff\%s_2[s_1\ student\text{-}in() = s_2\ works\text{-}in()$
$\wedge s_2\ name() = ''Adams'']$

Example 5.7 *Find students who are not research assistants*
$student - research\text{-}assistant$
Since $M_e(student) - M_e(research\text{-}assistant) = \phi$*, because* $M_e(student) \subseteq M_e(research\text{-}assistant)$*, in the output pair* $M_e(student)$ *is returned according to definition 4.2. Also, remembering that* $T_{instances}(research\text{-}assistant) \subseteq T_{instances}(student)$*, the same query can be coded using the select operation as follows:*
$student\%s\ [s \notin T_{instances}(research\text{-}assistant)]$

Example 5.8 *Let net-salary(t) be a method defined in the staff class to return the net salary of a staff member after deducting taxes at the rate of t. Assume t=0.1 for research-assistants and t=0.15 for other staff members. It is required to find the names and net salaries of staff members:*

$$(staff - research\text{-}assistant)![\{name(), net\text{-}salary(0.15)\}]$$
$$\bigcup research\text{-}assistant![\{name(), net\text{-}salary(0.1)\}]$$

First the difference operation is used to find staff members who are not research assistants; then the one level project operation is applied on the result with t=0.15 and on research-assistants with t=0.1; the union of both results is considered to be the output from this query.

Example 5.9 *Find students attending the same courses*

$$(student\%s_1 \times student\%s_2)\ [s_1\ courses() = s_2\ courses() \wedge s_1\ name() < s_2\ name()]$$

Remember from definition 4.2 that, when combined with a selection operation, both of 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. Notice that the result of the query of *example 5.9* will be a direct subclass of the root because the *student* class has some instance variables with atomic domains. However, using nest instead of cross-product forces the result to be a subclass of the *student* class. The difference is due to the fact that while the nest operation will append to every student a set of identities of related students, the cross-product operation on the other hand forms, according to the definition of cross-product operation in definition 4.2, new values each consisting of the identity of a student together with the set of identities of related students.

Example 5.10 *Find staff members earning more than the average salary in their department*

$$staff\%s_1 \gg staff < \{works\text{-}in()\}, average, salary() > \%s_2[s_1\ salary() > s_2\ avsalary()]\ [\{name()\}]$$

where avsalary() is a message to return the calculated average salary in the result of the aggregate function application; it is a concatenation of the first two letters of the applied function, average, with the last message in the used message expression, here salary(). We nest staff with the result of the application of the aggregate function average on staff members grouped by works-in(). Then those staff members satisfying the given predicate expression are selected and finally projection on name() is performed.

6 Conclusions

In this paper, we formally described a query model for object-oriented database systems. Our query model is not restricted to handle existing objects only, however, the introduction of new relationships as well as new objects is also facilitated. A new relationship could have a stored value by extending objects in the operand to include new values for the new instance variables. It is also possible for a new relationship to have a derived value in terms of existing values by extending the behavior of the operand to facilitate the derivation of the required relationship. Operands and the output of a query are defined to have a pair of sets, a set of objects and a set of message expressions. Thus having the characteristics of an operand, the output from a query could itself be an operand and hence the closure property is naturally maintained.

A message expression results in the evaluation of the underlying methods and in the same sequence as if they all together form a single method invoked by that message expression. Furthermore, message expressions are used in the invocation of behavior as well as behavior constructors. Also, message expressions facilitate accessing of stored and derived values leading to computational completeness without having an embedded query language leading to in impedance mismatch. Consequently, methods could be coded solely by utilizing the object algebra and hence simplify the optimization process. On the other hand, proposals that do not overcome the impedance mismatch problem are still suffering from not supporting full optimization for being unable to resolve methods.

The operators of our object algebra subsumes those of the relational and nested algebras and hence it is more powerful than either one. The equal handling of objects as well as the behavior defined on them is an important requirement of an object algebra; thus we satisfied it in the presented query model. This is due to the presence of data and behavior in an object-oriented data model in contrast to having only data in the relational data model. Behavior is handled via message expressions. We support aggregate functions whose outputs are also pairs of sets like any operand.

We started by defining a set of objects and a set of message expressions for a class. Having such a pair, a class is shown to be an operand. By this, some operands were defined to be existing classes. Other operands are defined to be the outputs of queries. As the only known characteristics of the output from a query are a pair of sets -a set of objects and a set of message expressions, we have proven that from such a pair other class characteristics could be derived. Having the characteristics of a class, the output from a query is in fact a class. Thus, we decided on the proper placement of such a class in the lattice.

Concerning the current status of our research, we are working on the completeness of the described object algebra by studying its different aspects. Also, the handling of recursive queries is under consideration to determine whether any further extensions to the algebra improves its power.

References

[1] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects," INRIA, Tech.Rep.No. 846, May 1988.

[2] A. Alashqur, S. Su and H. Lam, "OQL: A Query Language for Manipulating Object-Oriented Databases," *Proceedings of the 15th International Conference on Very Large Databases,* Amsterdam, pp. 433-442, August 1989.

[3] R. Alhajj (Al-Hajj), "A Query Model and a Query Language for Object-Oriented Database Systems," Technical Report, Bilkent University, 1991.

[4] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Data Model for Object-Oriented Databases," *Proceed-*

ings of the 6th International Symposium on Computers and Information Sciences, Antalya, October 1991.

[5] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Formal Data Model and Object Algebra for Object-Oriented Databases," *Applied Mathematics and Computer Science,* Vol. 2, No. 1, pp. 49-63, 1992.

[6] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Query Language for Object-Oriented Databases," *Proceedings of the 7th International Symposium on Computers and Information Sciences,* Kemer, November 1992.

[7] R. Alhajj (Al-Hajj) and M.E. Arkun, "Queries in Object-Oriented Database Systems," *Proceedings of the ISMM International Conference on Information and Knowledge Management,* Maryland, November 1992.

[8] R. Alhajj (Al-Hajj) and M.E. Arkun, "Object-Oriented Query Language," *Accepted to the Journal of Informationn and Software Technology.*

[9] F. Bancilhon, et.al., "FAD: A Powerful and Simple Database Language," *Proceedings of the 13th International Conference on Very Large Databases,* Brighton, pp. 97-105, 1987.

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

[11] J. Banerjee, W. Kim and K.C. Kim, "Queries in Object-Oriented Databases," *Proceedings of the 4th International Conference on Data Engineering,* Los Algeles, CA, pp. 31-38, February 1988.

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

[13] S. Cluet, et. al., "Reloop, an Algebra Based Query Language for an Object-Oriented Database System," *Proceedings of the 1st International Conference on Object-Oriented and Deductive Databases,* December 1989.

[14] C.J. Date, *An Introduction to Database Systems,* 4th Edition, Vol. 1 and Vol. 2, Addison-Wesley, 1986.

[15] U. Dayal, "Queries and Views in an Object-Oriented Data Model," *Proceedings of the 2nd International Workshop on Database Programming Languages,* pp. 80-102, June 1989.

[16] O. Deux, et al., "The O_2 System," *Communication of ACM,* Vol. 34, No. 10, 1991.

[17] D.H. Fishman, et al., "IRIS: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, pp. 48-69, 1987.

[18] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation,* Addison Wesley, 1983.

[19] S.N. Khoshafian and G.P. Copeland, "Object Identity," *Proceedings of the International Conference on Object- Oriented Programming Systems, Languages and Applications,* Portland, OR, pp. 406-416, September 1986.

[20] W. Kim, "A Model of Queries for Object-Oriented Databases," *Proceedings of the 15th International Conference on Very Large Databases,* Amsterdam, pp. 423-432, 1989.

[21] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," Research Directions in Object- Oriented Programming, Shriver B. and P. Wegner Eds, MIT Press, Cambridge, MA, 1987.

[22] F. Manola and U. Dayal, "PDM: an object-oriented data model," *Proceedings of the International Workshop on Object-Oriented Databases,* Pacific Grove, CA, pp. 18-25, 1986.

[23] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research," Technical Report 88-001, Intl. Computer Science Inst. Berkeley California, May 1988.

[24] S.L. Osborn, "Identity Equality and Query Optimization," *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems,* Ebernburg, pp. 346-351, September 1988.

[25] M.A. Roth, H.F. Korth and A. Silberschatz, "Extending Algebra and Calculus for Nested Relational Databases", *ACM Transactions on Database Systems,* Vol.13, No.4, pp.389-417, December 1988.

[26] L.A. Rowe and M.R. Stonebraker, "The Postgres Data Model," *Proceedings of the 13th International Conference on Very Large Databases,* Brighton, pp. 83-96, 1987.

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

[28] D. Shipman, "The Functional Data Model and the Data Language Daplex," *ACM Transactions on Database Systems,* Vol. 6, No. 1, March 1981.

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

[30] S.L. Vandenberg and D.J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity and Inheritance," *Proceedings of ACM-SIGMOD Conference on Management of Data,* June 1991.