Object-oriented query language facilitating construction of new objects

R Alhajj and M E Arkun

In object-oriented database systems, messages can be used to manipulate the database; however, a query language is still a required component of any kind of database system. In the paper, we describe a query language for object-oriented databases where both objects as well as behaviour defined in them are handled. Not only existing objects are manipulated; the introduction of new relationships and new objects constructed out of existing ones is also facilitated. The operations supported in the described query language subsumes those of the relational algebra aiming at a more powerful query language than the relational algebra. Among the additional operators, there is an operator that handles the application of an aggregate function on objects in an operand while still having the result possessing the characteristics of an operand. The result of a query as well as the operands are considered to have a pair of sets, a set of objects and a set of message expressions; where a message expression is a sequence of messages. A message expression handles both stored and derived values and hence provides a full computational power without having an embedded query language with impedance mismatch. Therefore the closure property is maintained by having the result of a query possessing the characteristics of an operand. Furthermore, we define a set of objects and derive a set of message expressions for every class; hence any class can be an operand. Moreover, the result of a query has the characteristics of a class and its superclass/subclass relationships with the operands are established to make it persistent.

database systems, object-oriented database systems, query language, object algebra, message expression

Database systems in their conventional sense proved to be non-appropriate for and fell short in meeting the requirements of engineering and information based applications including AI, CAD/CAM and OIS. Consequently, it was recognized that the relational model which could efficiently handle conventional business applications should undergo certain improvements to be adapted to new applications. Thus, set-valued attributes were allowed after relaxing the first normal form restriction. A more advanced extension is based on complex objects where sets and tuples are arbitrarily nested with the relational algebra and calculus being extended to facilitate the manipulation of the database[1-4].

To satisfy object sharing within complex objects, object identity was introduced and extended database systems were developed[5]. A more advanced step towards satisfying emerging applications requirements is the combination of object-oriented concepts[6,7,38] with the database technology in developing object-oriented database systems[8-13]. But, there is still no agreement on standardization within the realm of object-orientation. Neither the boundaries for the query model have been set up nor an object-oriented query language has been well defined yet. This is one of the common criticisms against object-oriented databases[14]. However, it is agreed that object-oriented database systems are more powerful than conventional databases at both the modelling and the manipulation phases. They are more powerful at the modelling phase due to the features of inheritance, encapsulation, identity and complex objects. They are more powerful at the manipulation phase due to messages that handle both stored and derived values which result in full computational power. We argue that this superiority should also be maintained at the query language level. This paper focuses in the direction where the shortages of the already proposed object-oriented query languages have been identified in order to try to overcome them in the query language presented.

A general powerful characteristic of object-oriented query languages is that messages substitute most queries of conventional databases. For instance, the message **name()** when sent to an instance in the *student* class, the name of the particular student is returned. Although a single message is sufficient for such an operation in the object-oriented context, a selection followed by a projection is necessary to get the same result in the relational model. An additional join should precede in case that name is not a column in the student relation. Another example can be seen in sending the message **courses()** to a student and the message **grade()** to the obtained result. Although it is handled due to the implicit join[15] present in object-oriented data models, this corresponds to an explicit join in the relational model. The two messages **courses()** and **grade()** form what we call *a message expression*. In general, a message expression is defined to be a valid sequence of messages $m_1, \ldots, m_n$, with $n \geq 1$.

However, messages alone do not completely satisfy the query language requirements. Rather it is widely accepted that a query language must be a part of any database system. Thus, an object-oriented query language is still required for more complex situations and to support associative access. In other words, although the modelling power of an object-oriented

Bilkent University, Faculty of Engineering, Department of Computer Engineering and Information Sciences, Bilkent 06533, Ankara-Turkey

 0950-5849/93/090519-11

database system presents implicit joins[16] by allowing instances in a class to form the domain for an instance variable in another class, an explicit join is still necessary to introduce new relationships into the model; otherwise the manipulation power of the model will be restricted. Allowing an explicit join raises the closure property problem[17]. Therefore, it is necessary to have a query language that handles the introduction of new relationships and maintains the closure property. The relational model satisfies the closure property with respect to the relational algebra operations and 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. This property is enforced in this paper by having the operands as well as the result of a query possess the same characteristics.

We now describe a query language for object-oriented databases[18–22]. An operand has a pair of sets, a set of objects and a set of message expressions defined on elements of the first set. Message expressions preserve encapsulation and information hiding, in addition to providing full computational power to the user via handling both derived and stored values without any need to have an embedded query language with impedance mismatch. Also, the output of any operation has a similar characterizing pair where the constituting sets are defined and derived from the sets in the operand(s). By doing this, none of the object-oriented features is violated while maintaining the closure property. The operations of the query language subsume those of the relational algebra aiming at a more powerful query language than the relational algebra. In addition to the relational operators, we define other operators, e.g., *Nest*, *One-Level-Project* and *Apply*. The Nest operation introduces a required relationship into the model; it is an explicit join that substitutes a missing implicit join; it is equivalent to the Cross-Product operation under certain conditions. The One-Level-Project operation outputs the result of the evaluation of a set of message expressions against objects of an operand; its aim is to reduce the depth of nesting. The relational algebra-like operation does not evaluate any message expressions but only drops some of them to limit the values accessible inside objects of the operand. The inverse of the Project operation is to add some message expressions to those applicable to a given set of objects; this operation is defined in terms of others as indicated later in the fourth section. The Apply operation handles the application of an aggregate function on objects in an operand. By using the operators of the language described in this paper, we will be able to manipulate existing objects and introduce new relationships among objects.

We define the set of total instances for a class c, denoted $T_{instances}(c)$, to be the union of its instances with all the instances of its subclasses. Also a set of message expressions for a class c, denoted $M_e(c)$ can be derived starting with the set of messages used to invoke its methods. Therefore, a class has a set of objects and a set of message expressions. Furthermore, it is possible to derive the class characteristics from any pair of objects and of message expressions. Such a possibility helps in making the output from a query persistent when required.

The rest of the paper is organized as follows. The second section summarizes the related work. In the third section we introduce the basic features of the data model on which the query language is based. The query language itself is described in the fourth section via illustrative examples, and the fifth section contains the conclusions.

RELATED WORK

Several query languages are described in the literature for particular object-oriented database systems. The pros and cons of those languages are summarized in this section to justifying the motivation for the development of the query language described in this paper. From among such query languages, those of Gemstone[12], O_2[10,23], EXODUS[24,25], IRIS[11], ORION[16,26], *OSAM**[17], Postgres[5], PDM[13,27], ENCORE[28] and the formal calculi and algebra developed by Straube and Özsu[29] in addition to those described in[30–34] are emphasized in this section. These languages are based on different paradigms. The query languages of[13,27] are based on the functional paradigm, while the query languages of[16,26] are based on the message-passing paradigm. Other languages are based on extensions to the relational paradigm: such as extensions of QUEL[5,24] and extensions of SQL[23]. The query language of IRIS[11] is based on both the functional and the relational paradigms where functions are used in an object-oriented SQL, OSQL, constructs. OSQL is embedded inside common LISP via macro extensions, hence does not overcome impedance mismatch.

These languages can be identified as either only preserving objects in the database[12,17,24,26,29] or providing operators for the creation of new objects[13,16,23,27,28,33]. Such a distinction is due to the disagreement on whether all required relationships are definable at the modelling phase. We and others, e.g.,[28,33], argue that the definition of new relationships and hence the creation of new objects, should be facilitated by the query model. But it is necessary to resolve problems that arise due to the creation of objects; otherwise there will be inconsistencies. One such problem is to maintain the closure property[17]. In other words, the output of a query should be allowed as an operand in further operations in the model.

A major drawback of languages such as those described in[12,26,29] 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 output of an operation is a relation which does not have the same structural and behavioural 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

instance, in O_2[10,23] 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 do not enforce it. The query languages of[5,24,31] use nested relations as their logical view of object-oriented databases. A nested relation is allowed as an operand in addition to other operands with object-oriented features. Although operators in these languages operate on and produce nested relations, we argue that nested relations do not form a proper logical representation of object associations. In order to use nested relations to represent objects, a large amount of data has to be replicated in the representation.

The query language of Gemstone is a calculus sublanguage embedded inside OPAL, the object-oriented programming language of Gemstone. Furthermore, queries in Gemstone violate encapsulation because they are formed over the instance variables of an object. Postgres stores QUEL and C procedures as attribute values.

The algebra described in[25] has an equivalent expressive power to the EXCESS query language of the EXTRA data model described in[24]; it assumes a data model in which several general type constructors are provided and data structures are built through free composition of those constructors. The Daplex functional data model[35] illustrates an integration of functions, relations and object-oriented features; its basic constructs are entities and functions. The Daplex query language has a set of iterators that apply a predicate to a set of values. The algebra of PDM[13,27] is based on an extension of the Daplex functional data model[35]; it modifies the relational algebra to handle functions, i.e., the operators and the result are functions. A major restriction in PDM is that object identity is not supported and only union compatible items are allowed as operands to set-based operators. The algebra of ENCORE[28] is based on a data model[36] that has all types as abstract data types whose implementations are hidden from the algebra. It comprises a set of built-in functions to collection objects. The output of a query is of the tuple type which is essentially the nested relational representation, since it allows the nestings of tuples. ENCORE views everything as an object with an identity.

Straube and Özsu developed a set-based object-oriented query algebra and a corresponding calculus, but their algebra does not handle the closure property. Also, they studied the problem of type unions in some detail. However, although their algebra has a formal basis, it is less expressive compared to others described in the literature. Osborn's object algebra[33] was developed for a general-oriented data model defined on the three generic classes of atomic, aggregate and set objects. A major drawback of Osborn's algebra is that it does not support encapsulation and the closure property is not maintained; set operations do not accept atomic and aggregate objects produced by other operations.

The first version of the query model of ORION[26] does not support the creation of new objects. However, the second version provides this property. In the query model of ORION[16] 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 persistency of the result is desired[18,22]. 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.

Siegelmann and Badrinath[37] describe an algebra where query results are presented as implicit answers (expressions), and where a class name replaces an explicit enumeration of all its instances in a step towards allowing information exchange at higher levels of abstraction: this is a useful capability in decision support systems. A subset of instances from a class are explicitly enumerated only in case that there is no class that includes all of them and no other instances. However, the data model on which their algebra is based supports only simple inheritance and atomic domains, i.e., no complex objects. Also, they do not describe any method for making an implicit answer explicit.

BASIC FEATURES OF THE DATA MODEL

In this section we briefly describe the required features in a data model for the sake of the query language. It is required to have objects, classes and methods. An object has a state and behaviour where the state is reachable via the behaviour. To maintain the object-oriented features, it is important for the query language to equally handle both the state and the behaviour of objects. Furthermore, an object has an identity and a value. Identity distinguishes one object in the database from other existing objects and provides for object sharing[7]. 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. On the other hand, a non-atomic domain contains the set of objects of a class represented by their identities. The following are objects where o_i represents identity:

$o_1 \langle$"*Jack*", 21,"*M*", $\phi \rangle$
$o_2 \langle$"*Mary*", 48,"*F*", $\{o_1, o_3\} \rangle$
$o_3 \langle$"*Michel*", 25,"*M*", ϕ, 5, $\{o_6, o_7\}$, $o_8 \rangle$
$o_4 \langle$"*John*", 52,"*M*", $\{o_1, o_3\}$, 42*K*, $o_8 \rangle$
$o_5 \langle$"*Susan*", 28,"*F*", ϕ, 5, $\{o_6, o_7\}$, o_8, 15*K*, $o_8 \rangle$
$o_6 \langle$"*CS578*","*Parallel Machines*", 4 $\rangle$
$o_7 \langle$"*CS565*","*Database Theory*", 3 $\rangle$
$o_8 \langle$"*Computer Science*", $o_4 \rangle$

We use *value*(o) and *identity*(o) to denote the value and the identity of object o, respectively. To avoid confusion, the identity function will be dropped and o will be used to represent identity(o). 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 ($o_1 = o_2$) *if and only if* $identity(o_1) = identity(o_2)$
—shallow-equal($o_1 \doteq o_2$) *if and only if* $value(o_1) = value(o_2)$
—deep-equal ($o_1 \tilde{=} o_2$) *if and only if* by recursively replacing every object identity o_i in $value(o_1)$ and $value(o_2)$ by $value(o_i)$, equal values are obtained.

$$(o_1 = o_2) \Rightarrow (o_1 \doteq o_2) \Rightarrow (o_1 \tilde{=} o_2)$$

identical ⇒ shallow-equal ⇒ deep-equal

Objects that have the same state structure are collected in one class. For instance, looking at the previous objects, it seems that o_1 and o_2 should be in the same class. Inheritance is supported to overcome duplication and allow for reusability. Inheritance covers state structure and behaviour. Next are the state structures of the classes related to the previous objects:

person ⟨ ∅, *name* : *string*, *age* : *integer*, *sex* : {"*M*","*F*"}, *children* : {*person*} ⟩
student ⟨ {*person*}, *year* : *integer*, *courses* : {*course*}, *student-in*:*department* ⟩
staff ⟨ {*person*}, *salary* : *integer*, *works-in*: department ⟩
research-assistant ⟨ {*student*, *staff*} ⟩
course ⟨ ∅, *code* : *string*, *name* : *string*, *credit* : *integer* ⟩
department ⟨ ∅, *name* : *string*, *head* : *staff* ⟩

where any pair *iv*:*d* represents an instance variable defined such that *iv* is the instance variable name and *d* is its underlying domain. For example, the domain of the *sex* instance variable is the set {"*M*","*F*"}. A domain specified between braces indicates that always a set is expected as the value of that instance variable; even a single element is represented by a singleton set. For example, *courses*:{*course*} specifies a set of objects (represented by their identities) from the *course* class as the courses registered by a student.

The first argument in a class definition is a set with elements being 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 behaviour definition. 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. We use $T_{instances}(c_i)$ to denote the set of total instances of class c_i which contains objects in c_i and all objects in its direct and indirect subclasses:

$T_{instances}(person) = \{o_1, o_2, o_3, o_4, o_5\}$
$T_{instances}(student) = \{o_3, o_5\}$
$T_{instances}(course) = \{o_6, o_7\}$
$T_{instances}(staff) = \{o_4, o_5\}$
$T_{instances}(research\text{-}assistant) = \{o_5\}$
$T_{instances}(department) = \{o_8\}$

A class has a set of methods. A method implements a function and is invoked using a corresponding message. A method also has a number of arguments $n \geq 0$. In other words, every method T is invoked via a corresponding message and implements a predefined function

$$f : d_1 \times d_2 \times \cdots \times d_n \rightarrow d_r,$$

where d_1 is the domain of the receiver, $d_2, d_3, \ldots, d_n$ are the domains of the arguments of f and d_r is the domain of the result of the application of f on objects of d_1, i.e., d_r is the range of f. Given objects $o_i \in d_i$, where $i = 1$ to n and r,

$$f(o_1, o_2, \ldots, o_n) = o_r.$$

The message that invokes the method T should have $(n-1)$ arguments drawn from the domains d_2 to d_n, respectively. We use messages(c) to denote the set of messages of class c. Among the methods found in a class there exists a method corresponding to each of the instance variables of the class. For instance, the method invoked by the message *name*() implements the function

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

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

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

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. For instance,

$$f_2(o_4, 2K) = o_4\ salary() + 2K = 42K + 2K = 44K.$$

Therefore, methods are used not only to deal with properties of objects but also to manipulate either stored values or in deriving new values in terms of properties and existing values of objects. Some other examples on methods which return existing stored values are,

o_1 *age()* returns 20,
o_5 *courses()* returns $\{o_6, o_7\}$
o_5 *courses() code()* returns {"CS565", "CS578"}.

Looking at the previous examples, it is obvious that *age*() ∈ *messages*(*person*), *courses*() ∈ *messages*(*student*) and *code*() ∈ *messages*(*course*), while there does not exist any class c such that *courses*() *code*() ∈ *messages*(c). It is recognized that *courses*() *code*() is an element of a superset of the set *messages*(*student*). Such a superclass is called the set of message expressions of the *student* class. The set of message expressions of a class c is defined to include any combination of messages which when applied to an object of the class c causes the execution of the underlying methods and in the same sequence as if they all together were a single method

invoked by the message expression to return a desired value. Formally, a message expression is defined next.

Definition 2: Message expressions
Starting from the set of messages of a class c, the set of possible message expressions of class c can be determined by:

- messages(c) is subset from the set of message expressions of class c
- if the domain of the result of an element x_i of the set of message expressions of class c is $T_{instances}(c_i)$ for some class c_i, then the concatenation of x_i with every element of messages(c_i) is an element of message expressions of class c, i.e., if $m \in$ messages(c_i), then $(x_i\ m) = x_{il}$ is an element of the set of message expressions of class c

We use $M_e(c)$ to denote the set of message expressions of class c. The two steps of definition 2.2 are used in deciding whether a given message expression is an element of $M_e(c)$ for a given class c. For instance, the set of message expressions of the person class is given next: $M_e(person) = messages(person) \cup children()^+$ *messages* $(person)\dagger = children()^*messages(person)$.

Due to the facility provided by message expressions for providing the value of a relationship in terms of existing ones, not all required relationships need to be stored within the realm of object-oriented databases. Thus, derivable relationships are also possible. For instance, it is possible to have *brother-of*, *sister-of*, *wife-of* and *husband-of* as derived values depending on the *sex* and the stored-valued *children* relationship between persons. Each of *brother-of*, *sister-of*, *wife-of* and *husband-of* is handled as a message with an underlying method implementing the desired relationship. In general, a derived value is determined after executing a sequence of one (or more) method(s) underlying the message(s) constituting a corresponding message expression. Such a facility saves both space and time required in storing and maintaining related values in a consistent state.

THE QUERY LANGUAGE

In this section, we describe a query language which maintains the closure property in a natural way without violating the object-oriented features. Although most of the existing query languages are devoted to the manipulation of existing objects without creating new ones, we and others[28,33] recognize the need for a more powerful query language that allows the creation of new objects in addition to the manipulation of existing ones. This adds the flexibility of introducing new relationships into the model making the manipulation more powerful. An operand has a pair of sets, a set of objects and a set of message expressions. 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 behaviour definition of the result of a query from those of the operand(s); hence it is a class[18,22].

Starting from a set of objects and a corresponding set of message expressions, it is possible to derive class characteristics[18,22]. To remember, a class has a set of objects, a set of instance variables, a set of methods with corresponding messages in a one to one relationship, and a set of superclass. 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 by including 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 in this section and detailed in[18,22].

In the rest of this section, the different operations of the query language are introduced together with illustrative examples. In these examples, we differentiate between temporary and persistent evaluation of a query. An assignment free query is always evaluated on a temporary basis and we use = and := to differentiate between temporary and persistent evaluations, respectively. While a temporary evaluation of a query ends by finding the pair of sets in the result, a persistent evaluation continues with the finding of class characteristics of the determined 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 defining the operators, A and B are assumed to be either pairs, i.e., $\langle T_{instances}(A), M_e(A)\rangle$ and $\langle T_{instances}(B), M_e(B)\rangle$, or query expressions. A query expression is a sequence of one or more query operators applied to some operands to produce a pair of sets.

Selection

The Selection operation presents a restriction on objects of the operand. 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 Selection operation has the following form:

$$\text{Select(A, p)} = \langle \{o \mid o \in T_{instances}(A) \wedge p(o)\}, M_e(A)\rangle$$

where p is a predicate expression built using object variables, message expressions and constants; also

†Notice that a^* is used to indicate zero or more concatenations of a with itself, i.e., e, a, aa, ..., while a^+ indicates one or more concatenations of a with itself, i.e., a, aa, aaa, ...

quantifiers may be present in a predicate. One object variable is bound by $T_{instances}(A)$ and other object variables are bound by other queries. An object variable followed by a message expression returns either a stored or a derived value. A returned value can be compared with another value or constant using conventional comparison operators in addition to $\subseteq$, $\not\subseteq$, $\in$ and $\notin$ added to support set-based comparisons and $=$, $\doteq$ and $\tilde{=}$ for identical, shallow-equal and deep-equal comparisons of objects, respectively. Given an object o, we use $p(o)$ to denote the evaluation of predicate expression p by substituting o for an object variable in p. To illustrate this, consider the following examples on predicate expressions. Let s_1 and s_2 be object variables ranging over instances of the *student* class:

$"CS565" \in s_1$ *courses*() *code*() is a predicate to check students attending the course *"CS565"*;

$\exists c \in s_1$ *courses*() $\wedge\, c \in s_2$ *courses*() $\wedge\, s_1 \neq s_2$ is a predicate 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 another example of a predicate to check whether two given students do not have any courses in common;

$\exists c \subseteq s_1$ *courses*() $\wedge\, c \subseteq s_2$ *courses*() is an example of a predicate to check whether two given students have some courses in common.

Example 1 *Find brothers of 'Adams'.*

Select (*person* % p_1, p_1 *sex* ( ) $= "M" \wedge \exists p_2 \in T_{instances}$ (*person*) $\wedge\, p_2$ *name*() $= "Adams" \wedge \exists p_3 \in T_{instances}$(*person*) $\wedge \{p_1, p_2\} \subseteq p_3$ *children*())

where % indicates that the variable p_1 is bound to and ranges over the objects of the operand, here the person class. More than one variable may independently range over objects of an operand. For example, person%p_1%p_2 indicates that p_1 and p_2 range over objects of the person class.

Although Straube claims that his multiple operand Selection is more powerful[29], we will insist on supporting a single operand Selection. Because Straube does not support the closure property in his algebra, he has the Cross-Product operation embedded into the Selection. We argue that on comparing two algebras, the power of the whole algebra must be considered, not particular operations. A language that supports the creation of new objects is necessary and considered more powerful than any other language devoted only to the manipulation of existing objects.

Project and One-Level-Project

The Project operation hides some of the message expressions of the operand without the set of objects being affected. 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, dropping some message expressions by the Project operation hides some values from the accessible objects. The Project operation is defined as follows:

$$\text{Project(A, } M_1) = \langle T_{instances}(A), M_1 \rangle$$

where $M_1 \subseteq M_e(A)$, i.e., an element 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 the Project operation. On the other hand, the inverse of the Project operation is to add new elements to the set of message expressions of a pair and it is defined at the end of this section, after introducing the other operations in terms of which it is represented.

Example 2 *Assume that the staff class is not present in the lattice and the research-assistant class is defined as:*

research-assistant $\langle \{student\}, salary : integer,$
$works\text{-}in : department \rangle$

Assuming that it is not necessary for a student to work for the department he attends, we write:

staff := *Project(research-assistant, {name(), age(), sex(), children(), salary(), works-in()})*

From the messages of the research-assistant class, {year(), courses()} are the messages that the created staff class does not respond as they are hidden by the Project operation. In this query it is also possible to use the set messages(person) to replace the explicit enumeration of its elements, i.e., {name(), age(), sex(), children()}. In general, when possible, it is also permitted to replace an explicit enumeration of elements of $M_e(c)$ for some pair $\langle T_{instances}(c), M_e(c) \rangle$ by $M_e(c)$ itself, to have the expression in an implicit form providing for more readability. The derived staff class will be a direct superclass of the research-assistant class and $T_{instances}(staff) = T_{instances}$(research-assistant) just after this query. Not presented in this paper, we derive algorithms to maximize reusability so that the derived staff class will be recognized as a subclass of the person class and naturally placed in the lattice[18,22].

While the Project operation does not evaluate any of the provided message expressions, on the other hand, the One-Level-Project operation computes a new set of objects and a corresponding set of message expressions. 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. More explicitly, the one level project operation is handled as follows.

A subset M_1 of the message expressions in $M_e(A)$ is applied to every object in $T_{instances}(A)$ for A being an operand. The obtained values are collected to form the value of an object in the result of the one level project operation.

Message expressions applicable to the resulting objects are obtained by:

- Let x_1 be a message expression in M_1 and let m be the last simple message of x_1 which serves to map an object identifier to the value of the object.
- Find a message expression x_3 in $M_e(A)$ such that it is prefixed by x_1, i.e., $x_3 = x_2 m x_4$ and $x_1 = x_2 m$.
- Thus, the set of message expressions applicable to objects in the result of the one level project operation are all message expressions x such that $x = m x_4$.

The purpose is to collect together in a class all objects constructed by collecting the values reachable by the message expressions in M_1 applied to objects in $T_{instances}(A)$. Consequently, the One-Level-Project has the following form:

$$\begin{aligned}\text{OLproject(A, } M_1) = \langle \{o \mid \exists o_1 \in T_{instances}(A) \wedge value(o) \\ = (o_1\ M_1)\}, \{x \mid \exists x_1 \in M_1, x_1 = (x_2 m) \wedge len(x_1) \\ = len(x_2) + 1 \wedge \exists x_3 \in M_e(A) \wedge x_3 = (x_2 x) \wedge x = (m\ x_4)\}\rangle\end{aligned}$$

where $M_1 \subseteq M_e(A)$. The One-Level-Project operation corresponds to a sequence of unnest operations followed by a projection in the nested relational model[1,3,4]. For instance, OLproject(A, (messages(A) $-$ $\{m_1\}$)$\cup$(m_1 messages(B))), unnests A and B where $m_1 \in$ messages(A) and domain of m_1 is $T_{instances}(B)$. The depth of nesting decreases as the length of the longest message expression in M_1 increases.

Example 3 *Find the student names and course codes of students attending at least one course:*

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

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 result of this operation is the pair which corresponds to a class whose instances are constructed by collecting the name and course codes for all students attending one or more courses and whose message expressions are $\{name(), code()\}$.

Example 4 *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 t. To get the names and net salaries of staff members, assuming* $t = 0.1$, *we write*:

$$OLproject(staff, \{name(), net\text{-}salary(0.1)\})$$

The One-Level-Project operation does the function of Project and Image operations described in[28], the Apply of[33] and the Map operation described in[29], but we maintain the closure property without additional constructs.

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 OBJECT class which is the root.

Cross-Product and Nest

Although many relationships between objects are represented by the objects themselves, an explicit operation is required to handle cases when a relationship is not present 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[19]. Associativity of the Cross-Product operation is useful in query optimization[19,22], although not discussed in this paper. A query expression is optimized after representing it by a binary tree with leaf nodes being operands as pairs and non-leaf nodes are operators of the query language.

The Cross-Product operation has four different forms depending on the domains of the instance variables of the operands. These four forms, given next, are needed to make the Cross-Product operation associative; a property useful in query optimization[19,22].

By assuming two messages m_1 and m_2 with domains being $T_{instances}(A)$ and $T_{instances}(B)$, respectively, the four cases are:

First case: if objects in each of $T_{instances}(A)$ and $T_{instances}(B)$ have all included values drawn from non-atomic underlying domains:

$$\begin{aligned}Cproduct(A, B) = \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \\ \wedge value(o) = value(o_1).value(o_2)\}, \\ M_e(A) \cup M_e(B)\rangle\end{aligned}$$

Second case: if only objects in $T_{instances}(A)$ include at least one atomic underlying domain:

$$\begin{aligned}\text{Cproduct(A, B)} = \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \\ \wedge value(o) = identity(o_1).value(o_2)\}, \\ (m_1\ M_e(A)) \cup M_e(B)\rangle\end{aligned}$$

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

$$\begin{aligned}\text{Cproduct(A, B)} = \langle \{o \mid \exists o_1 \in T_{instances}(A) \exists o_2 \in T_{instances}(B) \\ \wedge value(o) = value(o_1).identity(o_2)\}, \\ M_e(A) \cup (m_2\ M_e(B))\rangle\end{aligned}$$

Fourth case: if objects in each of $T_{instances}(A)$ and $T_{instances}(B)$ include at least one atomic underlying domain:

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

By considering these four cases, the Cross-Product operation becomes associative[19].

When persistency in the lattice of the result is desired, the result of the Cross-Product operation is made a subclass of the operand that has all underlying domains being non-atomic and a direct subclass of the root otherwise.

The Nest operation takes two operands; it adds a value to each object in the first operand, the underlying domain of the added value is the objects in the second operand, i.e., $T_{instances}(B)$. It is defined as follows:

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

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, i.e., the first operand. Notice the similarity between the Nest operation and the second and third cases of the Cproduct operation.

When combined with the Selection operation, both of the Cross-Product and the Nest operations result in a join operation. Although the join due to the Nest is an outer-join, the join due to the Cross-Product is an inner-join.

Example 5 *Find the department whose head is "Adams".*

Select(department %d, d head() name() = "Adams")

The same query can be coded in two other forms as:

Select(Nest(department %d, staff %s), d head() = s ∧ s name() = "Adams")

and

Nest(Select(staff %s, s name() = "Adams", Select(department %d, d head() = s))

Notice that, the second and third query expressions given in this example explicitly show the benefit of maintaining the closure property by having the output from any query operation to be a pair usable as input to another query operation.

Example 6 *Find students attending the department whose head is "Adams".*

Select(student %s, s student-in() head() name() = "Adams")

The same query can also be coded as:

Select(Nest(student %s$_1$, staff %s$_2$), s$_1$ student-in() head() = s$_2$ ∧ s$_2$ name() = "Adams")

Example 7 *Find students attending the department in which "Adams" is working.*

Select(Nest(student %s$_1$, staff %s$_2$), s$_1$ student-in() = s$_2$ works-in() ∧ s$_2$ name() = "Adams")

Example 8 *Find students attending the same courses*

Cproduct(student %s$_1$, Select(student %s$_2$, s$_1$ courses() = s$_2$ courses() ∧ s$_1$ name()⟨s$_2$ name()))

Notice that the result of the query of example 8 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 as 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, new values each consisting of the identity of a student together with the set of identities of related students[19,22].

On the other hand, 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:

$$\text{Unnest(A, B)} = \text{Project(A, } M_e(A) - (m\, M_e(B)))$$

where $m \in messages(A)$ has domain as $T_{instances}(B)$.

Set operations

As mentioned before, the query language described in this paper handles and produces a pair of sets, a set of objects and a set of message expressions to handle the objects. So because we deal with sets, two basic set operations, Union and Difference, are supported in the query language; *intersection* is defined in terms of the difference operation, while the *symmetric difference* operation is defined in terms of the union, the difference and the intersection 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 Union operation is defined as follows:

$$\text{Union(A, B)} = \langle T_{instances}(A) \cup T_{instances}(B),\ M_e(A) \cap M_e(B) \rangle$$

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

student⟨∅, *name* : *string*, *age* : *integer*, *sex* : {"*M*","*F*"},
children : *student*, *year* : *integer*, *courses* : *course*,
student-in:department⟩
staff⟨∅, *name* : *string*, *age* : *integer*, *sex* : {"*M*","*F*"},
children : *student*, *salary* : *integer*, *works-in:department*⟩

The person class is derived as:

person := *Union(student, staff)*

The derived person class is a superclass of both operands and includes the union of their objects, but the intersection of their message expressions as stated in the definition of the Union operation.

Concerning the Difference operation, under the condition that $M_e(A) - M_e(B) \neq \phi$, the Difference operation has the following form:

$$\text{Difference(A, B)} = \langle \{o \mid o \in T_{instances}(A) \wedge o \notin T_{instances}(B)\}, M_e(A) - M_e(B) \rangle$$

However, if it occurs that $M_e(A) - M_e(B) = \phi$, then $M_e(A) - M_e(B)$ is replaced by $M_e(A)$ in the definition to get:

$$\text{Difference(A, B)} = \langle \{o \mid o \in T_{instances}(A) \wedge o \notin T_{instances}(B)\}, M_e(A) \rangle$$

Example 10 *Find students who are not research assistants.*

Difference(student, research-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.*

Remembering that $T_{instances}(research\text{-}assistant) \subseteq T_{instances}(student)$, *the same query can be coded using the select operation as follows:*

Select(student%*s*, $s \notin T_{instances}(research\text{-}assistant)$*)*

When persistency in the lattice is required, the result of the Difference operation becomes 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))

The symmetric difference operation is defined as follows:

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

Other operations

To have a more powerful query language, it is necessary to have the result of the application of an aggregate function used as an operand. The following operator is defined for that purpose. Given $X \subseteq M_e(A)$ and $x_i \in M_e(A)$, the application of an aggregate function f is defined as:

$$Apply(f, A, X, x_i) = \langle \{o \mid (o\ m_1) \subseteq T_{instances}(A) \wedge (o\ m_3) = f(\{(o_1\ x_i) \mid o_1 \in T_{instances}(A) \wedge \forall o_2 \in (o\ m_1), (o_2\ X) = (o_1\ X)\})\}, (m_1\ M_e(A)) \cup \{m_3\} \rangle$$

where $T_{instances}(A)$ is the domain of the result of message m_1, and the domain of the result of f is the domain of the result of message m_3.

The aggregation function is applied on A 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. In other words, objects in $T_{instances}(A)$ are partitioned into equivalence classes† based on the result of the evaluation of message expressions in X against those objects. Then, the aggregate function f is applied to objects in each of the equivalence classes by considering the value returned by the message expression x_i applied to each such object.

Example 11 *Find staff members earning more than the average salary in their department.*

Project(Select(Nest(staff%s_1, *Apply(average, staff,*
{*works-in()*}, *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(). In other words, first the set $T_{instances}(staff)$ *is partitioned into equivalence classes based on the result of the message works-in() by collecting in the same equivalence class staff members working for the same department. The second step is the application of the message salary() to every object and the aggregate function average is applied to get the average salary for objects in every equivalence class,*

†An equivalence class is a set of objects having common characteristics such that every two equivalence classes are disjoint, i.e., given any two equivalence classes A_1 and B_1, $A_1 \cap B_1 = \phi$.

separately. Then those staff members satisfying the given predicate expression are selected and finally projection on name() is performed.

Finally the inverse of the Project operation, Iproject, is defined at this point, as stated before, in terms of other operations. To add a subset M of $M_e(B)$ to $M_e(A)$, we 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; after that we project on $M_e(A) \cup M$ to get the target set of message expressions in the resulting pair:

$$\text{Iproject(A, B:M)} = \text{Project(OLproject(Nest(A, B), messages(A)} \cup (m \text{ messages(B))}), M_e(A) \cup M)$$

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

Conclusions

We described a query language for object-oriented database systems. A query expression is coded by applying operators 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 behaviour as well as behaviour constructors because a message expression leads to the execution of all the methods underlying constituting messages and in the same order as if all together form a single method. Concerning the result of a query expression, it is again a pair of sets, the same as those of the operands. So, the output of one query expression can be a further operand without any problem. Hence the closure property is maintained in a natural way. In producing the output pair from a query expression, the two constituting sets are derived by considering those of the operand(s). Therefore, the operators act on behaviour as well as on objects. While doing this, heterogeneous sets are considered and this adds much to the power of the described query language.

Message expressions deal with both stored and derived values and hence provide a full computational power making the OLproject operation of the query language more powerful than the unnest operation of the nested relational model. This property is also valid for the query language as a whole, where computed as well as stored values may be manipulated. Therefore, the object-oriented data model is not only more powerful than the relational data model, we also have a query language which is more powerful than the relational algebra. Furthermore, our query language is more powerful than others described in the literature in supporting object construction, behaviour construction via message expressions, and deals equally with the behaviour as well as the state of objects. Behaviour is necessary in maintaining the encapsulation feature of object-oriented data models.

Using the 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 behaviour. 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 (Iproject) which extends the behaviour of objects in the operand without their states being affected. On the other hand, the OLproject 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 Cproduct operation results in new objects, while other cases introduce new relationships.

Finally, 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.
- Behaviour is also uniformly handled like objects; creation of methods as well as objects in terms of other existing ones are facilitated.
- The addition of new classes is facilitated where we derive the characteristics of a class in terms of those of existing classes.
- Aggregation functions are supported in a consistent way so that the result could be used as an operand.
- Computational completeness is maintained without any need to have an embedded query language; an embedded query language leads to the impedance mismatch problem.

All of these are satisfied without loss of generality in the description. Concerning the current state of our research, we are examining the completeness of the described query language to determine whether and how it is possible to improve its power. For instance, the Apply operation that handles the application of an aggregate function adds much to the power. Also equivalents of different combinations of operators are being experimentally tested, and how much that improves query optimization is being considered.

REFERENCES

1 **Abiteboul, S and Beeri, C** 'On the power of languages for the manipulation of complex objects' INRIA, *Tech Rep No* 846 (May 1988)

2 **Date, C J** *An introduction to database systems* (4th edn) Vol 1 and 2, Addison-Wesley (1986)

3 **Jaeschke, G and Schek, H J** 'Remarks on the algebra of non-first normal form relations' *Proc. Symp. Principles of Database Systems* (March 1982) pp 127–138

4 **Roth, M A, Korth, H F and Silberschatz A** 'Extending algebra and calculus for nested relational databases' *ACM Trans. Database Systems* Vol 13 No 4 (December 1988) pp 389–417

5 **Rowe, L A and Stonebraker, M R** 'The Postgres data model' *Proc. 13th Int. Conf. Very Large Databases* Brighton (1987) pp 83–96

6 **Goldberg, A and Robson, D** *Smalltalk-80: the language and its implementation* Addison-Wesley (1983)

7 **Khoshafian, S N and Copeland G P** 'Object identity' *Proc. Int. Conf. Object-Oriented Programming Systems, Languages and Applications* Portland, OR (September 1986) pp 406–416

8 **Banerjee, J *et al.*** 'Data model issues for object-oriented applications' *ACM Trans. Office Inf. Systems* Vol 5 No 1 (1987) pp 3–26

9 **Carey, M J and Dewitt D J** 'The architecture of the EXODUS extensible DBMS' *Proc. IEEE Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA (September 1986) pp 52–65

10 **Deux, O *et al.*** 'The O_2 system' *Comm. ACM* Vol 34 No 10 (1991)

11 **Fishman, D H *et al.*** 'IRIS: an object-oriented database management system' *ACM Trans. Office Inf. Systems* Vol 5 No 1 (1987) pp 48–69

12 **Maier, D and Stein J** 'Development and implementation of an object-oriented DBMS' in **Shriver, B and Wegner, P (eds)** *Research directions in object-oriented programming* MIT Press, Cambridge (1987)

13 **Manola, F and Dayal, U** 'PDM: an object-oriented data model' *Proc. Int. Workshop on Object-Oriented Databases*, Pacific Grove, CA (1986) pp 18–25

14 **Neuhold, E and Stonebraker, M** 'Future directions in DBMS research' *Technical Report 88-001* Int. Computer Science Inst. Berkeley, CA (May 1988)

15 **Kim, W** 'Object-oriented databases: definition and research directors' *IEEE Trans. Knowl. and Data Eng.* Vol 2 No 3 (1990) pp 327–341

16 **Kim, W** 'A model of queries for object-oriented databases' *Proc. 15th Int. Conf. Very Large Databases*, Amsterdam (1989) pp 423–432

17 **Alashqur A, Su, S and Lam, H** 'OQL: a query language for manipulating object-oriented databases' *Proc. 15th Int. Conf. Very Large Databases*, Amsterdam (August 1989) pp 433–442

18 **Alhajj (Al-Hajj), R and Arkun, M E** 'A data model for object-oriented databases' *Proc. 6th Int. Symp. Comp. Inf. Sciences*, Antalya (October 1991)

19 **Alhajj (Al-Hajj), R and Arkun, M E** 'A formal data model and object algebra for object-oriented databases' *Applied Math. Comp. Science*, Vol 2 No 1 (1992) pp 49–63

20 **Alhajj (Al-Hajj), R and Arkun, M E** 'A query language for object-oriented databases' *Proc. 7th Int. Symp. Comp. Inf. Sciences*, Kemer (November 1992)

21 **Alhajj (Al-Hajj), R and Arkun, M E** 'Queries in object-oriented database systems' *Proc. ISMM Int. Conf. Inf. Knowl. Manage.* Maryland (November 1992)

22 **Alhajj (Al-Hajj), R and Arkun, M E** 'A query model for object-oriented databases' *Proc. 9th IEEE Int. Conf. Data Eng.* Vienna (April 1993) (to appear)

23 **Cluet, S *et al.*** 'Reloop, an algebra based query language for an object-oriented database system' *Proc. 1st Int. Conf. object-oriented and deductive databases* (December 1989)

24 **Carey, M J, DeWitt, D J and Vandenberg, S L** 'A data model and a query language for EXODUS' *Proc. ACM-SIGMOD Conf. Management of Data*, Chicago (May 1988) pp 413–423

25 **Vandenberg, S L and DeWitt, D J** 'Algebraic support for complex objects with arrays, identity and inheritance' *Tech. Rep., CS-TR-987* University of Wisconsin-Madison (December 1990)

26 **Banerjee, J, Kim, W and Kim, K C** 'Queries in object-oriented databases' *Proc. 4th Int. Conf. Data Eng.* Los Angeles, CA (February 1988) pp 31–38

27 **Dayal, U** 'Queries and views in an object-oriented data model' *Proc. 2nd Int. Workshop on Database Programming Languages* (June 1989) pp 80–102

28 **Shaw, G and Zdonik, S** 'A query algebra for object-oriented databases' *Proc. 6th Int. Conf. Data Eng.* Los Angeles, CA (1990) pp 154–162

29 **Straube, D D and Özsu, M T** 'Queries and query processing in object-oriented database systems' *ACM Trans. Inf. Syst.* Vol 8 No 4 (1990) pp 387–430

30 **Albano, A, Cardelli, L and Orsini, R** 'Gelileo: a strongly-typed interactive conceptual language' *ACM transactions on database systems* Vol 10 No 2 (1985) pp 230–260

31 **Bancilhon, F *et al.*** 'FAD: a powerful and simple database language' *Proc. 13th Int. Conf. Very Large Databases*, Brighton (1987) pp 97–105

32 **Kuper, G and Vardi, M** 'A new approach to database logic' *Proc. ACM PODS* (1984)

33 **Osborn, S L** 'Identity equality and query optimization' *Proc. 2nd Int. Workshop on object-oriented database systems* Ebernburg (September 1988) pp 346–351

34 **Zaniolo, C** 'The database language GEM' *Proc. ACM-SIGMOD Conf. Management of Data* San Jose, CA (May 1983) pp 207–218

35 **Shipman, D** 'The functional data model and the data language daplex' *ACM Trans. Database Systems* Vol. 6 No 1 (1981)

36 **Hornick, M F and Zdonik, S B** 'A shared segmented memory system for an object-oriented database' *ACM Trans. Office Inf. Systems* Vol 5 No 1 (1987) pp 70–95

37 **Siegelmann, H T and Badrinath, B R** 'Integrating implicit answers with object-oriented queries' *Proc. 17th Int. Conf. Very Large Databases* Barcelona (September 1991) pp 15–24

38 **Stefik, M and Bobrow, D G** 'Object-oriented programming: Themes and variations' *AI Magazine* (January 1986) pp 40–62