

An Object-Oriented Query Model: An Algebraic Approach With Closure

REDA ALHAJJ

Department of Math & Computer Science

American University of Sharjah

P. O. Box 26666, Sharjah, U.A.E.

E-mail: ralhajj@aus.ac.ae

An object algebra is presented as a formal query model for object-oriented databases. The algebra serves not only to access and manipulate the structure and behavior of objects, but also supports the creation of new objects and the introduction of new relationships into the schema. It provides a more powerful and flexible tool than do messages for effectively dealing with complex situations and meeting associative access requirements. Operands as well as the results of operations in the proposed algebra are formally characterized as pairs of sets - a set of objects capturing the states and a set of message expressions comprised of sequences of messages modeling the object behavior. The closure property is achieved in a natural way by letting the results of operations possess the same characteristics as do the operands in an algebra expression. Some operators of the algebra resemble those of the relational algebra but with different syntax and semantics. Additional operators are introduced to complement them. A class is shown to possess the properties of an operand by defining a set of objects and deriving a set of message expressions for it. Furthermore, the result of an object algebra expression is shown to have the characteristics of a class whose superclass/subclass relationships with its operand class(es) can be established, thus providing a mechanism to properly and persistently place it in the class lattice (schema).

Keywords: closure, database system, object algebra, object-oriented data model, object-oriented query language, query model

1. INTRODUCTION

Although the relational data model was found to be suitable for handling conventional business applications, the first normal form restriction motivated extensions to satisfy the requirements of new application areas. Early extensions relaxed the first normal form by allowing set-valued attributes. A still a more advanced extension is based on complex objects, where sets and tuples are arbitrarily nested [1, 2]. To achieve object sharing within complex objects, object identity was introduced. A more advanced step towards satisfying data intensive application requirements was the development of object-oriented systems. Object-oriented systems evolved to satisfy the demand for a more appropriate representation and modeling of real world entities in application domains such as CAD/CAM, OIS and AI. To satisfy the requirements of such applications, it was recognized that an integration of object-oriented concepts with database technology leads to more appropriate representation methods, and many object-oriented data models have, thus, been developed [3-7]. However, there is no agreement on standardization within the realm of

Received March 2, 1999; revised June 23, 1999; accepted August 19, 1999.
Communicated by Wei-Pang Yang.

object-orientation. The boundaries for a standard query model have not been set up, nor has an object-oriented query language been well defined. This is one of the common criticisms of object-oriented databases [8].

Comparing the relational model with an object-oriented model shows that the latter is more powerful in the modeling stage but does not support a standard formal query model. While the non-atomic domain concept is supported by the nested relational model [1, 2], we see inheritance, identity and encapsulation among the features that the nested relational model lacks. Identity provides for object sharing and object independence. Inheritance provides for structure and behavior sharing. Encapsulation provides for abstraction. Furthermore, in the nested relational model, if a relation r has an attribute with domain relation r_1 , then r_1 can not have any attribute with domain r ; however, such domain restrictions are relaxed in object-oriented models. 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. For instance, the message **name()**, when sent to an instance in the *student* class, returns the name of the particular student. Another example can be seen in sending the message **courses()** to a student and the message **grade()** to the result of the first message. Although it is handled due to the implicit join [9] 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 as a sequence of messages $m_1..m_n$, with $n \geq 1$.

While simple message expressions make object-oriented systems superior to the relational model, an object-oriented query language is still needed for more complex situations and to support associative access. In other words, although the modeling power of an object-oriented database supports implicit joins [9] by allowing instances in a class to form the domain for an attribute in another class, an explicit join is necessary for introducing new relationships into the model; otherwise the manipulative power of the model will be restricted. Allowing an explicit join raises the problem of closure maintenance. Therefore, it is necessary to have an object algebra that facilitates the introduction of new relationships while maintaining the closure property; otherwise the relational model will be more powerful.

Our approach is to retain the closure property of an object algebra without violating object-oriented concepts. An object-oriented model should be more powerful than the relational model in both the modeling and the manipulation phases. It is more powerful in the modeling phase due to such features as inheritance, encapsulation, identity and complex objects. It is more powerful in the manipulation phase due to the handling of both stored and derived values through various methods.

In this paper, we describe an object algebra for object-oriented data models. Our object algebra is a superset of the relational algebra, but with different semantics and operands. Also, as the schema of an object-oriented data model may contain cycles, and due to the growing interest in recursive queries, we handle recursive queries since they are of great interest to the application areas of object-oriented databases, e.g., CAD/CAM and Software Engineering applications, which are modeled in terms of recursive definitions. Therefore, even if recursive queries are not peculiar to object-oriented query languages only, query languages supporting advanced applications must include some forms of recursion. Although only linear recursion is considered in our work, this includes an important set of

recursive queries since most recursive queries encountered in real cases are linear. Furthermore, efficient processing strategies have already been developed to handle linear recursion [10].

The main idea of our work is that an operator should handle objects and their behavior uniformly. Therefore, 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 the objects that qualify for membership 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 former set of the pair. By using such pairs as operands and in the output, closure is maintained in a natural and consistent way. Furthermore, a message expression leads to the invocation of behavior and acts as a behavior constructor because it leads to the execution of methods underlying the constituting messages in sequence as if they all form a single method invoked by the message expression.

Using object algebra operators, we build object algebra expressions and show that every object algebra expression has the characteristics of a class. Moreover, we derive the inheritance (sub/superclass) relationship between the result of an object algebra expression and the operand(s). Therefore, the result of any object algebra expression can be persistent. In [11], we go further and seek the proper placement of a query result in the lattice. Other aspects of our on-going research on object-oriented databases can be found in [12-18].

To sum up, operands and the result of a query are defined in such a way as to not violate object-oriented constructs while retaining the closure property. Behavior, like objects, is also uniformly handled; 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. By considering the problems identified by Klug as well as his relational approach to overcoming them [19], aggregation functions are supported in a consistent way so that the result can be used as an operand. Recursive queries are handled without any need to have a PROLOG-like query language.

The rest of the paper is organized as follows. The related work is discussed in section 3. In section 3, the data model is described, and the basic terminology used in the formalization is introduced. We give the characteristics of a class, and later in section 4, we derive the same characteristics for the result of an object algebra expression. We define a set of objects and a set of message expressions as the constituents of pairs forming operand(s) and the output of a query. In section 4, we define the object algebra by constructing object algebra expressions. We derive characteristics of the result of an object algebra expression which are the same as the characteristics of a class. Some illustrative examples are given in section 5. Section 6 is the conclusion.

2. RELATED WORK

A query language must be a component of any database system [20]. Consequently, Kim identifies a query language among the requirements of object-oriented systems despite the use of messages to manipulate the database [21]. Thus, several query languages, such as those of IRIS [5], GemStone [7], ObjectStore [22], Postgres [23], O_2 [24, 25], EXODUS [4, 26], ORION [9, 27], OSAM* [28, 29], PDM [30, 31] and ENCORE [32] in addition to others [33-48], have been proposed. These languages are based on different paradigms.

Some query languages are based on the functional paradigm [30, 31] while others [9, 27] are based on the message-passing paradigm. Still other languages are based on extensions to the relational paradigm, such as extensions to QUEL [4, 23] and extensions to SQL [24]. The algebra described in [35] contains only the operations of the relational algebra. The authors did not mention anything related to closure. They only presented a generic mapping scheme to flatten data models. The work described in [41] presents a monoid comprehensive calculus-based query unnesting algorithm. The query language of IRIS [5] is based on both the functional and the relational paradigms, where functions are used in Object-oriented SQL (OSQL) constructs. An object-oriented temporal algebra is described in [47], in which the authors paid much attention to formalism. The work described in [40] presents an access method for supporting nested queries in object-oriented databases.

O_2 [24] has an object algebra which handles values as well as objects, and this leads to a kind of mismatch in which some operands violate encapsulation while others do not. The query languages of [4, 23, 34, 49] 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. In order to use nested relations to represent objects, a large amount of data has to be replicated in the representation. Our approach described in this paper does not suffer from such mismatches for both operands and the output of a query are identified via a pair of sets with no violation of encapsulation. Finally, the work described in [42] describes the design and implementation of a graph-based visual query language for the O_2 object-oriented database system.

Queries in Gemstone violate encapsulation because they are formed over the attributes of an object. A similar query language is that of the ObjectStore database management system [22]. In addition, such query languages do not facilitate the introduction of new relationships or the construction of new objects, which are two of the facilities provided by our approach. Postgres, a successor to INGRES [50], is an extended relational data model that utilizes relational query processing techniques. Such extended relational models with abstract and procedural data types are still considered value-based, record-oriented models. They aim to add extensibility and object management capabilities to the relational model.

EXCESS, the query language of EXODUS, has an underlying relational query processor. In EXCESS, new types created during query processing do not participate in inheritance in any way – they do not inherit any attributes or methods except those explicitly specified by the types from which they were created, nor do they become part of any inheritance hierarchy. A major drawback of the algebra described in [26] is that values are the output from any query. PDM modifies the relational algebra to handle functions; i.e., the operators and the results are functions. Major restrictions are that object identity is not supported, and that only union compatible items are allowed as operands for set-based operators. Osborn's object algebra [51] is an extension of the relational algebra. Major drawbacks of Osborn's algebra are that it does not support encapsulation, and that the closure property is not thoroughly maintained; set operations do not accept atoms or aggregate objects produced by other operations.

In the algebra of ENCORE [32], the output of a query is of the Tuple type, which is essentially the nested relational representation since it allows nesting of tuples. To insure type consistency of the result, in ENCORE, union compatibility is imposed on the algebra operators. A drawback of the algebra of ENCORE is that two identical queries do not give the same response. This is because every resulting collection is a newly identified object in

the database. For this reason, operators for eliminating duplicates are defined in this algebra. Such operators become unnecessary when the result from a query possesses the characteristics of a class that is properly and naturally placed in the lattice, as in our algebra. Another algebra which is similar to that of ENCORE is the one described in [52]. However, this algebra does not use methods, and it returns values rather than objects as the output from a query; consequently it does not generate object identifiers.

The first version of the query language developed for ORION [27] preserves objects in the database. A major drawback of this language is that a query on a class returns either the value of a single attribute or some objects of the class. Also, when more than one class are involved in a query, those classes have to be nested with respect to each other. However, in the second version of the query model of ORION [9], although the result of a query operation is a class, the improper placement of resulting classes in the lattice leads to duplication of class contents; hence, ORION violates the reusability feature of object-oriented systems. Moreover, it is an overhead to have a class as the output of a temporary query, is the case with ORION. To retain the closure property, the model described in [53] follows the approach proposed for ORION [9] and makes the result of a query a direct subclass of the root, TOP-CLASS. Furthermore, the result of a query does not have user defined methods; it very much resembles a set of tuples in a relation, so encapsulation is not respected. In this paper, we describe the output of a query based on the minimum requirements of an operand, and from such characteristics, we show how to derive the characteristics of a class when it is required to have the result persistent [11, 12]. In *OSAM**, operands in a query are the database itself and all the subdatabases derived from the original database by means of query operations; the result of a query is a subdatabase.

To sum up, these languages are classified as either object-preserving [4, 7, 22, 27, 28, 46] or object-creating [9, 24, 30-32, 51, 53]. This a distinction is made due to the disagreement about whether it is possible to have all required relationships defined in the modeling phase. We and others [32, 51] argue that the definitions of new relationships and the creation of new objects should be supported by a query model. A new relationship may have either a stored or a derived value and may be handled by the introducing of an attribute or a method to the definition of a class, respectively. For the attribute case, objects in the class to which the relationship is added are extended to include values for the new attribute [14]. For the method case, on the other hand, the behavior is extended without any stored value being added because the value of the relationship is determined by invoking the corresponding method as needed. A new object may be formed by collecting values from either objects in different classes or from the constituents of an object nested to an arbitrary level. However, it is necessary to resolve problems that arise due to the creation of objects; otherwise, there will be inconsistencies. One such crucial problem is closure maintenance [28]. In other words, the output of a query should be allowed as an operand in the model. Unfortunately, a major drawback of the languages already described in the literature [7, 27, 51, 52] is that they do not maintain the closure property. Others [4, 23, 24, 30, 31, 34, 49] introduce non-object-oriented constructs for 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. On the other hand, although some query models [9, 53] try to retain the closure property, in [9], the operands and the output from a query are classes, and in [53], the

output is a class with some restrictions make it much more like a relation. In this sense, this paper presents an intuitive approach to closure maintenance. We handle the output from a query neither as a value nor as a relation. Instead, we identify the output from a query as having the characteristics of an operand; a pair of sets consists of a set of objects and a set of message expressions which can be used to handle the objects in the former set. Moreover, while many of the query languages cited in this section characterize the output from a query as a relation, we show how it is possible to derive a class from any pair of sets obtained as the output from a query. In this way, the output from a query can be identified as a class with the same characteristics as the other classes in the schema, without violating reusability [9] or restricting class facilities [53].

3. THE DATA MODEL

3.1 Informal Description

In this section, we describe the data model features deemed necessary as they relate to the object algebra. Consider the class hierarchy shown in Fig. 1, where the state structures of the classes are indicated and the set of objects, from Fig. 2, included in each class is identified. In Fig. 1, any pair $iv:d$ represents an attribute defined such that iv is the attribute name and d is the underlying domain. A domain specified between braces indicates that a set is expected as the value of that attribute. For instance, $children:person$ specifies a set of persons as the children of a person. List values are also allowed by specifying the domain between square-brackets. For example, $x:[person]$ means that x takes lists of objects from the *person* class as its values. Not shown in Fig. 1 is that a class definition includes a set of methods (operations) that embodies the behavior of its objects to support encapsulation and information hiding. As shown in Fig. 1, classes are arranged in a hierarchy with the general class OBJECT at the root, i.e., a direct or indirect superclass of all other classes. Furthermore, a subclass may include additional attributes and behavior definitions. Some example objects are given in Fig. 2, where o_i represents an identifier. To retain the object-oriented features, it is important for the operators of the object algebra to uniformly handle both the state and the behavior of an object. Furthermore, an object has an identity and a value. A value may be a single value, a list of values or a set of values drawn from a domain. A domain is either atomic or non-atomic; an atomic domain may be any of the conventional domains, including integers, characters etc. A non-atomic domain includes the set of objects of a class represented by their identities.

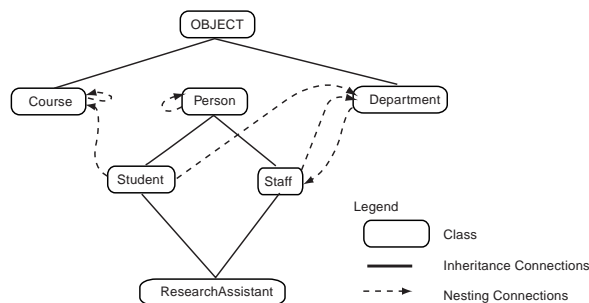


Fig. 1. An example class hierarchy.

$o_1 < \text{"Jack"}, 21, \text{"M"}, \phi >$
 $o_2 < \text{"Mary"}, 42, \text{"F"}, \{o_1, o_7\} >$
 $o_3 < \text{"Michel"}, 5, \text{"M"}, \phi >$
 $o_4 < \text{"John"}, 65, \text{"M"}, \{o_8, o_6\} >$
 $o_5 < \text{"Susan"}, 25, \text{"F"}, \phi, 5, \{o_{16}, o_{11}\}, o_{10} >$
 $o_6 < \text{"Smith"}, 45, \text{"M"}, \{o_1, o_7\}, 50K, o_{10} >$
 $o_7 < \text{"Tom"}, 18, \text{"M"}, \phi, 3, \{o_{13}, o_{14}\}, o_{10} >$
 $o_8 < \text{"Adams"}, 40, \text{"M"}, \phi, 43K, o_{10} >$
 $o_9 < \text{"George"}, 22, \text{"M"}, \phi, 5, \{o_{16}, o_{11}\}, o_{10}, 15K, o_{10} >$
 $o_{10} < \text{"Computer Science"}, o_8 >$
 $o_{11} < \text{"CS565"}, \text{"Database Theory"}, 3, \phi >$
 $o_{12} < \text{"CS101"}, \text{"Introduction to Programming"}, 3, \phi >$
 $o_{13} < \text{"CS211"}, \text{"Design of Programming Languages"}, 3, \{o_{12}\} >$
 $o_{14} < \text{"CS330"}, \text{"Data Structures"}, 3, \phi >$
 $o_{15} < \text{"CS450"}, \text{"Database Design"}, 3, \{o_{13}, o_{14}\} >$
 $o_{16} < \text{"CS578"}, \text{"Parallel Machines"}, 4, \phi >$

Fig. 2. Example objects from the classes shown in Fig. 1.

3.2 Basic Notations

For a class c , we use the following notations:

- $W_{behavior}(c)$ is the set of messages used to invoke the methods defined in or inherited by class c . In other words, every method T is invoked via a corresponding message and implements a predefined function $f: d_1 \times d_2 \times \dots \times d_n \rightarrow d_r$, where d_1 is the domain of the receiver, d_2, d_3, \dots, 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 $r, f(o_1, o_2, \dots, o_n) = o_r$. The message that invokes the method T should have $(n - 1)$ arguments drawn from domains d_2 to d_n , respectively. Methods are used not only to deal with properties of objects, but also to manipulate either stored values or to derive new values in terms of properties and existing values of objects.

Example 3.1 (Class Behavior) Related to the classes of Fig. 1, the following sets of messages are assumed:

$$\begin{aligned}
 W_{behavior}(person) &= \{name(), age(), sex(), children()\}, \\
 W_{behavior}(staff) &= \{name(), age(), sex(), children(), salary(), works-in(), net-salary(t), increase-salary(t)\}, \\
 &= W_{behavior}(person) \cup \{salary(), works-in(), net-salary(t), increase-salary(t)\}, \\
 W_{behavior}(student) &= W_{behavior}(person) \cup \{year(), courses(), student-in()\}, \\
 W_{behavior}(research-assistant) &= W_{behavior}(student) \cup W_{behavior}(staff), \\
 W_{behavior}(department) &= \{name(), head()\}, \\
 W_{behavior}(course) &= \{code(), name(), credit(), prerequisites()\}. \quad \square
 \end{aligned}$$

Given a class c , let $m \in W_{behavior}(c)$ and $o \in W_{instances}(c)$, where $W_{instances}(c)$ denotes the extent of class c ; it is the union of instances of class c with the instances of its subclasses. Sending message m to an object o , i.e., $o m$, returns a value from the range of the function which corresponds to message m . Let V be the set of all such returned values. The returned value may be a single value, a list of values or a set of values since the latter is allowed as a value in Definition 3.2 (given below). Furthermore, given $O \subseteq W_{instances}(c)$, sending m to objects in O , i.e., $O m$, returns a set of values with each individual element being a single value, a list or by itself a set. This is because message m is actually applied to individual objects constituting the set O and the obtained results form a set. To formalize, let $U = \{u_1, u_2, \dots, u_n\}$ be a set of values, where for every i , $(\exists d \in D, u_i \in d) \vee (u_i = \{u_{i1}, u_{i2}, \dots, u_{in}\})$. $Um = \bigcup_{i=1}^n set(u_i m)^1$, and in general, $u_i m = u_j$.

- $W_{behavior}(c)$ is the set of all attributes defined in or inherited by a class c .

Example 3.2 (Attributes) For the classes shown in Fig. 1, the following sets of attributes are assumed:

$$\begin{aligned} W_{attributes}(person) &= \{name: string, age: integer, sex: [“M”, “F”], children: \{person\}\} \\ W_{attributes}(student) &= W_{attributes}(person) \cup \{course: \{course\}, student-in: department\} \\ W_{attributes}(department) &= \{name: string, head: department\} \\ W_{attributes}(course) &= \{name: string, code: string, credit, integer, prerequisite: \{course\}\} \\ W_{attributes}(staff) &= W_{attributes}(person) \cup \{salary: integer, works_in: department\} \\ W_{attributes}(research-assistant) &= W_{attributes}(student) \cup W_{attributes}(staff) \quad \square \end{aligned}$$

For any attribute iv , $domain(iv)$ and $value(iv)$ denote the domain and the value of the attribute iv , respectively. Formally, given a class c , with $iv \in W_{attributes}(c)$, then $value(iv) \in domain(iv)$.

Definition 3.1 (Domain) The set of domains D is defined so as to include:

- $d \in D$ and $2^d \in D$, where d is any of the atomic domains, such as the set of integers, the set of reals, the set of characters, the set of strings etc., and 2^d indicates the powerset or the set of all subsets of the set d .
- for any class c , $W_{instances}(c) \in D$ and $2^{W_{instances}(c)} \in D$. □

Definition 3.2 (Value) The set of values V is defined so as to include:

- $\forall d \in D$, we have $d \subseteq V$; since ϕ is a subset of any set, it follows that $\phi \in V$ due to the use of the powerset to specify some domains in D . ϕ is used to indicate the value nil , and sending any message to nil returns nil itself.
- $[v_1, v_2, \dots, v_n] \in V$ such that $\exists d_i \in D$, with $\{v_1, v_2, \dots, v_n\} \subseteq d$. □

¹ $set(i) = \{i\}$, i.e., a singleton set with i being its element. Consequently, $set(u_i m)$ is a singleton set with its element being the result of the application of the message m on u_i .

- $L_{instances}(c)$ is the set of objects in class c but not in any of its subclasses, as given for each of the example classes in Fig. 1.

Example 3.3 (Local Instances) For the classes shown in Fig. 1, the following sets of local instances are assumed:

$$\begin{aligned} L_{instances}(person) &= \{o_1, o_2, o_3, o_4\}, \\ L_{instances}(student) &= \{o_5, o_7\}, \\ L_{instances}(research-assistant) &= \{o_9\}, \\ L_{instances}(staff) &= \{o_6, o_8\}, \\ L_{instances}(course) &= \{o_{11}, o_{12}, o_{13}, o_{14}, o_{15}, o_{16}\}, \\ L_{instances}(department) &= \{o_{10}\}. \end{aligned}$$

□

For an object o , we use $value(o)$ and $identity(o)$ to denote the value (where the value of an object is a set of values of the attributes defined in its class) and the identity of object o , respectively. The identity of an object, i.e., $identity(o)$, is a logical representative of the object o .

Given an object $o \in L_{instances}(c)$ for some class c ,

$$value(o) \in X_{i=1}^{card(W_{attributes}(c))} (domain(W_{attributes}(c)_i))^2.$$

Unless confusion may arise, the $identity$ function will be dropped, and o will be used to represent $identity(o)$.

- $W_{instance}(c) = L_{instances}(c) \cup_{i=1}^{card(S)} W_{instances}(S_i)$, where $S = \{S_1, S_2, \dots, S_{card(S)}\}$ is the set of direct subclasses of class c .

Example 3.4 (Class Extent) For the classes shown in Fig. 1, the following extents are derived:

$$\begin{aligned} W_{instances}(course) &= \{o_{11}, o_{12}, o_{13}, o_{14}, o_{15}, o_{16}\}, \\ W_{instances}(department) &= \{o_{10}\}, \\ W_{instances}(research-assistant) &= \{o_9\}, \\ W_{instances}(staff) &= \{o_6, o_8, o_9\} = W_{instances}(research-assistant) \cup \{o_6, o_8\}, \\ W_{instances}(student) &= \{o_5, o_7, o_9\} = W_{instances}(research-assistant) \cup \{o_5, o_7\}, \\ W_{instances}(person) &= \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9\} \\ &= W_{instances}(student) \cup W_{instances}(staff) \cup \{o_1, o_2, o_3, o_4\}. \end{aligned}$$

□

- $C_p(c)$ is the set of direct superclasses of class c , except that $\forall c_i \in C_p(c)$, so we have $c_i \neq OBJECT$. The OBJECT class is implicitly a direct superclass of any class c for $C_p(c) = \emptyset$; otherwise, it is an indirect superclass for a direct or indirect superclass of a class in $C_p(c)$.

² Given two sets A and B, $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$. In general $A \times B \neq B \times A$. However, here we assume that equality always holds; i.e., order is not important inside the resulting tuples because those values are handled via messages.

3.3 Message Expressions

Any sequence of messages applicable to objects of a class c is said to be a *message expression* of class c . For example, o_9 is an object in the *student* class; message $courses()$ in the *student* class invokes the method implemented to return the set of courses registered by a given student and so, $o_9.courses()$ returns $\{o_{11}, o_{16}\}$ from the *course* class. Also, the combination $courses().code()$ can be applied to any object in the *student* class, and $o_9.courses().code()$ returns $\{\text{"CS565"}, \text{"CS578"}\}$. Thus, $courses()$ and $courses().code()$ are elements of the message expressions of the *student* class.

Definition 3.3 (Message expressions) Given a class c , the set of message expressions of class c , denoted $M_{expressions}(c)$, is recursively defined by:

- $W_{behavior}(c) \subseteq M_{expressions}(c)$.
 - If $x \in M_{expressions}(c)$ and x returns a value from $W_{instances}(c')$, for some class c' , then $(x W_{behavior}(c')) \subseteq M_{expressions}(c)$.
- Here, x is postfixed with every message in $W_{behavior}(c')$. For example, let $W_{behavior}(c') = \{m_1, m_2\}$; then $(x \{m_1, m_2\}) = \{x_{11}, x_{21}\}$, where $x_{11} = (x m_1)$ and $x_{21} = (x m_2)$. Therefore, starting from $W_{behavior}(c)$, we can determine whether a given message expression is an element of $M_{expressions}(c)$. \square

Example 3.5 (Message Expression) For the classes shown in Fig. 1, the following sets of message expression are derived:

$$\begin{aligned}
 M_{expressions}(person) &= W_{behavior}(person) \cup children()^+ W_{behavior}(person)^3 \\
 &= children()^* W_{behavior}(person), \\
 M_{expressions}(student) &= M_{expressions}(person) \cup \{year(), courses(), student-in()\} \cup courses() \\
 &\quad M_{expressions}(course) \cup student-in() M_{expressions}(department), \\
 M_{expressions}(staff) &= M_{expressions}(person) \cup \{salary(), work-in(), net-salary(), increase-salary() \\
 &\quad \} \cup works-in() M_{expressions}(department), \\
 M_{expressions}(research-assistant) &= M_{expressions}(student) \cup M_{expressions}(staff), \\
 M_{expressions}(department) &= W_{behavior}(department) \cup head() M_{expressions}(staff), \\
 M_{expressions}(course) &= prerequisites()^* W_{behavior}(course). \quad \square
 \end{aligned}$$

The length of a message expression x , denoted $len(x)$, returns the number of messages constituting x . We differentiate between implicit and explicit representations of $W_{instances}(c)$, $W_{behavior}(c)$ and $M_{expressions}(c)$ for a given class c . In an implicit representation, a subset of the elements of the represented set is replaced with a single set name while an explicit representation enumerates all elements of a set. For instance, $\{name(), age(), sex(), children(), year(), courses(), student-in()\}$ is an explicit enumeration of the set $W_{behavior}(student)$ while $W_{behavior}(person) \cup \{year(), courses(), student-in()\}$ is an implicit representation of the same set. As illustrated in the example classes and by definition, for any class c , $W_{instances}(c)$ can be implicitly represented in terms of $W_{instances}(c_i)$, where c_i is a subclass of c ; $W_{behavior}$

³ Notice that a^* is used to indicate zero or more concatenations of a with itself, i.e., e, a, aa, \dots , while a^+ indicates one or more concatenations of a with itself, i.e., a, aa, aaa, \dots

(c) can be implicitly represented in terms of $W_{behavior}(c_j)$, where $c_j \in C_p(c)$; $M_{expressions}(c)$ can be implicitly represented in terms of $M_{expressions}(c_k)$, where $c_k \in C_p(c)$ or $c_k = domain(iv)$ for some $iv \in W_{attributes}(c)$. Sometimes, it is not possible to explicitly enumerate elements of $M_{expressions}(c)$ for some class c , especially when a direct or indirect cycle is used to specify the domain of attributes. In such cases, an implicit representation becomes necessary and is easier to follow and understand. For instance, because of the cycle caused by having the *person* class itself as the domain of the *children* attribute in the *person* class, there is no finite explicit representation of $M_{expressions}(person)$; however an implicit representation is possible and is given above in the examples on message expressions.

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. Such a relationship is important as a class is derived from the result of a query later in section 4.

Lemma 3.1 Given two classes c_1 and c_2 ,

$$M_{expressions}(c_1) \subseteq M_{expressions}(c_2) \Leftrightarrow W_{behavior}(c_1) \subseteq W_{behavior}(c_2).$$

Proof:

$$\begin{aligned} & \text{(if part) } x \in W_{behavior}(c_1) \Rightarrow x \in M_{expressions}(c_1) \text{ (by Definition 3.3)} \\ & \Rightarrow x \in M_{expressions}(c_2) \text{ (because } M_{expressions}(c_1) \subseteq M_{expressions}(c_2), \text{ given)} \\ & \Rightarrow x \in W_{behavior}(c_2) \text{ (because } \text{len}(x) = 1 \text{ and the only elements of } M_{expressions}(c_2) \text{ of} \\ & \quad \text{length one are elements of } W_{behavior}(c_2), \text{ from Definition 3.3)} \\ & \Rightarrow W_{behavior}(c_1) \subseteq W_{behavior}(c_2), \\ & \text{(only if part) } x \in M_{expressions}(c_1) \Rightarrow x = x_1 \dots x_n, \text{ with } n \geq 1 \text{ such that } x_1 \in W_{behavior}(c_1) \text{ (by} \\ & \quad \text{Definition 3.3)} \\ & \Rightarrow x_1 \in W_{behavior}(c_2) \text{ (because } W_{behavior}(c_1) \subseteq W_{behavior}(c_2), \text{ given)} \\ & \Rightarrow x \in M_{expressions}(c_2) \text{ (by Definition 3.3)} \\ & \Rightarrow M_{expressions}(c_1) \subseteq M_{expressions}(c_2). \quad \square \end{aligned}$$

We will utilize Lemma 3.1 to construct object algebra expressions in Definition 4.2. Informally, the proof of Lemma 3.1 follows from Definition 3.3, where starting from message expressions of length one, i.e., messages of a class, it is possible to derive all other possible message expressions.

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.

Derivation of the value of an implicit relationship in terms of existing ones is facilitated by message expressions through the execution of the underlying sequence of methods that correspond to a selection (implicit join) producing the derived value. For instance, it is possible to have the relationships *brother-of* and *sister-of* as derived values using the stored-valued *children* relationship between persons with the *sex* attribute recorded for persons. Both *brother-of* and *sister-of* are handled as messages by using an underlying method to implement the desired relationship in terms of the selection operation. In general, a derived

value is determined after executing a sequence of one or more methods underlying the message(s) constituting a corresponding message expression. This saves both space and time needed to store and keep related values in a consistent state.

4. THE QUERY MODEL

Although many of the existing query languages are restricted to the manipulation of existing objects and cannot create new ones, we and others [32, 51] argue that a more powerful query language is needed that can handle new objects manipulate existing ones. This adds the flexibility of instantaneously introducing new relationships into the model, thus making the manipulation more powerful. Our object algebra is a superset of the relational algebra, hence, it is at least as powerful as the relational algebra. Although we give a different interpretation and semantics for our algebra operators, we still use the terminology of relational operators. A lot of work has been done on relational algebra and its optimization. Thus, we intend to benefit from this accumulation of knowledge and techniques while still being aware of the benefits of not violating the object-oriented principles. An operand e in our object algebra should have a pair of sets: a set of objects and a set of message expressions, denoted by $\langle W_{instances}(e), M_{expressions}(e) \rangle$. Using elements of $M_{expressions}(e)$, we can access elements of $W_{instances}(e)$. 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 also should have a pair of sets derived in terms of the pair(s) of operand(s). Therefore, a query can appear anywhere an operand can appear. We call any operand, whether an actual pair or an unevaluated query, an *object algebra expression*. Therefore, our object algebra acts on and produces items that have defined pairs. Hence, our object algebra intuitively retains the closure property without violating object-oriented principles.

4.1 Informal Description

The object algebra proposed here includes the five basic operators of the relational algebra in addition to nest and one level project operators and aggregate function applications. Our treatment of aggregate functions is similar to that of Klug for overcoming the problems mentioned in [19]. A *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 below. The set of message expressions of the resulting pair is the same as that of the operand. For instance, the result of selecting courses with no prerequisites from the pair corresponding to the *course* class is the pair $\langle \{o_{11}, o_{12}, o_{14}, o_{16}\}, M_{expressions}(course) \rangle$.

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

P_1 : T and F are predicate expressions representing the truth values *true* and *false*.

P_2 : Given two values y_1 and y_2 having compatible underlying domains such that at least y_1 or y_2 is of the form $x(o)$, 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 .

$P_{2.1}$: $y_1 \text{ op } y_2$ is a predicate expression, where

$$op \in \begin{cases} \{=, >, <\} & \text{if both } y_1 \text{ and } y_2 \text{ are atomic values} \\ \{\in\} & \text{if } y_1 \text{ is a single value and } y_2 \text{ is a set of values} \\ \{\subseteq, =\} & \text{if both } y_1 \text{ and } y_2 \text{ are sets of values, } y_2 \text{ may be} \\ & W_{instances}(e) \text{ where } e \text{ is an OAE} \\ \{in\} & \text{if } y_1 \text{ is a single value and } y_2 \text{ is a list of values} \\ & \text{or both } y_1 \text{ and } y_2 \text{ are lists of values} \\ \{\equiv, \doteq, \equiv_i\}^4 & \text{if both } y_1 \text{ and } y_2 \text{ are single values from a non-atomic domain,} \\ & \text{i.e., } W_{instances}(c) \text{ for some class } c. \end{cases}$$

$P_{2.2}$: $\forall \exists z op_1 y_1 \wedge z op y_2$ is a predicate expression, where y_1 is a set of values and op_1 is \in , or y_1 is a list of values and op_1 is in , and

$$op \in \begin{cases} \{=, >, <\} & \text{if } y_2 \text{ is a single atomic value} \\ \{\in\} & \text{if } y_2 \text{ is a set of values; } y_2 \text{ may be } W_{instances}(e), \text{ where } e \text{ is an OAE} \\ \{\equiv, \doteq, \equiv_i\}^4 & \text{if } y_2 \text{ is a single value from a non-atomic domain} \end{cases}$$

$P_{2.3}$: $\exists z op_1 y_1 \wedge z op y_2$ is a predicate expression, where z and y_1 are sets of values and op_1 is \subseteq , or z and y_1 are lists of values and op_1 is in , and

$$op \in \begin{cases} \{\subseteq, =\} & \text{if } z \text{ and } y_2 \text{ are sets of values, } y_2 \text{ may be } W_{instances}(e), \text{ where } e \text{ is an OAE} \\ \{\exists\} & \text{if } y_2 \text{ is a single value and } z \text{ is a set of values} \\ \{includes\} & \text{if } y_2 \text{ is a single value and } z \text{ is a list of values} \end{cases}$$

P_3 : If p and q are predicate expressions, then (p) , $\neg p$, $p \wedge q$ and $p \vee q$ are predicate expressions.

Predicates as defined here within an object-oriented context are more powerful than they are in the relational model, where only atomic values are compared. Furthermore, extending predicate expressions to allow quantifiers to enable the creation of objects does affect the query power. For example, $\exists O \wedge O \subseteq W_{instances}(c)$ for some class c , binds O to a subset of $W_{instances}(c)$; the subset objects to which O is bound can be found by this query by incrementally enumerating the proper subsets of $W_{instances}(c)$. Such an object creation facility gives the algebra the power to recursively form a powerset [1].

Although the set of objects in 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. Therefore, dropping some message expressions by means of the *project* operation hides some values from the accessible objects. For instance, by projecting the pair corresponding to the person class on $X = \{name(), age(), sex()\}$, the pair $\langle W_{instances}(person), \{name(), age(), sex()\} \rangle$ is obtained, where only message expressions constituting X can be used to deal with objects in the other set of the resulting pair. On the other hand, the *inverse project*

⁴The symbols \equiv , \doteq and \equiv_i are used to check whether two objects are identical, shallow-equal or deep-equal-to-level- i .

operation is used to extend the set of message expressions in a pair to include more message expressions applicable to the objects of the pair, i.e., to provide more facilities for the user; this operation is defined in terms of others as will be shown later in section 4.2.

To facilitate the evaluation of a subset X of the message expressions against the objects of a given pair, the *one level project* operation is defined. Objects in the result are constructed from the values obtained by evaluating message expressions constituting X against objects of the operand. A corresponding set of message expressions is also determined so as to facilitate accessing the values encapsulated within the derived objects. For instance, the *one level project* of the pair corresponding to the *student* class on $\{name(), student-in()\}$ $name() \rightarrow name1()\}$, which is a subset of $M_{expressions}(student)$, results in derivation of the following pair:

$$\langle \langle \langle \text{"Susan"}, \text{"Computer Science"} \rangle, \langle \text{"Tom"}, \text{"Computer Science"} \rangle, \langle \text{"George"}, \text{"Computer Science"} \rangle, \{name(), name1()\} \rangle.$$

The \rightarrow denotes a renaming mechanism used to resolve a conflict in naming messages. Here, $name1()$ is a message that can be used to retrieve from objects of the result the department-name value as retrieved by the original message expression $student-in() name()$ from objects of the operand.

The purpose is to collect together in a new class all the objects constructed by collecting the values reachable by the message expressions in X applied to objects in the operand. In general, the set of message expressions in the result of the *one level project* operation is defined, depending on message expressions in X , to include:

- all message expressions prefixed by any message m such that there exists an element x_1 in X with m being the last message in the sequence of messages constituting x_1 ;
- all messages that correspond to those message expressions in X that return derived values. The underlying method of each such new message solely returns a value embodied in each of the result objects.

Despite the fact that many relationships between objects are embodied within the objects themselves, an explicit operation is required to handle cases where a relationship is not defined in the model. Both the *cross-product* and the *nest* operations are defined to introduce such 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 operand⁵, the result of the *cross-product* operation depends on the nature of the domains of the messages of the operands as explicitly stated in Definition 4.2 and given in section 4.2. While the *cross-product* operation is defined as being associative, the *nest* operation is not. However, the two operations are equivalent under certain conditions [11, 13]. Associativity of the *cross-product* operation is useful in query optimization [11, 13], but this property will not be discussed

⁵ In [11], it is shown that the result of the *nest* operation is a subclass of the first operand. Accordingly, objects of the first operand migrate into the result with their structure being extended to carry the new property due to the *nest* operation.

further in this paper. As an example, nesting the pairs corresponding to the *student* class and the *staff* class so as to assign to every student the head of his department as the supervisor results in the following pair:

$$\langle \langle \langle \text{"Susan"}, 25, \text{"F"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, o_8 \rangle, \langle \text{"Tom"}, 18, \text{"M"}, \phi, 3, \{o_{13}, o_{14}\}, o_{10}, o_8 \rangle, \langle \text{"George"}, 22, \text{"M"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, 15K, o_{10}, o_8 \rangle \rangle, M_{\text{expressions}}(\text{student}) \cup (m() M_{\text{expressions}}(\text{staff})) \rangle.$$

Here, $m()$ is a new message understandable by the objects in the resulting pair and used to return the supervisor, o_8 in this example. Since the supervisor is himself a staff member, any of the message expressions in $M_{\text{expressions}}(\text{staff})$ can be applied to the result of message $m()$.

On the other hand, use of the cross-product results in the following pair:

$$\langle \langle \langle o_5, o_8 \rangle, \langle o_7, o_8 \rangle, \langle o_9, o_8 \rangle \rangle, \{m_1() M_{\text{expressions}}(\text{student}) \cup m_2() M_{\text{expressions}}(\text{staff})\} \rangle,$$

where message $m_1()$ is for returning the first component, i.e., the identity of a student object, while $m_2()$ is for returning the identity of a supervisor object. The creation of objects with object identifiers representing some or all of their components necessitates the introduction of new messages to retrieve the values of these components via their object identifiers. Such a mechanism is introduced to prevent migration of objects which include atomic valued components that have non-uniform and varying length values. In the above example, the student objects o_5 , o_7 and o_8 each contain atomic valued components that are handled by the messages *name()*, *salary()*, *year()* etc. Before any one of these messages can be applied to the new objects defined by the cross-product operation, viz., $\langle o_5, o_8 \rangle$, message $m_1()$ needs to be applied to such an object to retrieve its student object identifier component, o_5 in this case. Similarly, $m_2()$ can be applied to retrieve a staff object identifier.

The cross-product operation can give the same result as the nest operation if all the message expressions of length one in $M_{\text{expressions}}(\text{student})$ return non-atomic values and at least one of the message expressions of length one from $M_{\text{expressions}}(\text{staff})$ were to return an atomic value. To illustrate a third case of the cross-product operation, assume that all the message expressions of length one from $M_{\text{expressions}}(\text{staff})$ return non-atomic values, and that at least one of the message expressions of length one from $M_{\text{expressions}}(\text{student})$ returns an atomic value. In this case, the resulting pair is

$$\langle \langle \langle o_5, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle, \langle o_7, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle, \langle o_9, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle \rangle, (m(), M_{\text{expressions}}(\text{student})) \cup M_{\text{expressions}}(\text{staff}) \rangle,$$

where message $m()$ returns the identity of a student object supervised by the receiving staff object. To illustrate the fourth case of the cross-product operation, assume that all the message expressions of length one from $M_{\text{expressions}}(\text{staff})$ and $M_{\text{expressions}}(\text{student})$ return non-atomic values. The cross-product operation under this condition will result in the following pair:

$$\langle \langle \langle \text{"Susan"}, 25, \text{"F"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle, \langle \text{"Tom"}, 18, \text{"M"}, \phi, 3, \{o_{13}, o_{14}\}, o_{10}, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle \rangle,$$

$$\langle \text{"George"}, 22, \text{"M"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, 15K, o_{10}, \text{"Adams"}, 40, \text{"M"}, \phi, 60K, o_{10} \rangle, \\ M_{expressions}(student) \cup M_{expressions}(staff) \rangle.$$

As mentioned before, the object algebra described in this work handles and produces a pair of sets. Therefore, as it deals with sets, two basic set operations, *union* and *difference*, are supported by the object algebra; *intersection* is defined in terms of the difference operation. 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. This intersection contains messages which handle the values drawn from the common underlying domains. The heterogeneous set of objects is the union of the sets of objects of the operands. For instance, the union of the pairs corresponding to the *student* and *staff* classes is the following pair:

$$\langle W_{instances}(staff) \cup W_{instances}(student), M_{expressions}(staff) \cap M_{expressions}(student) \rangle.$$

The difference operation is handled in one of two ways depending on the relationship between the message expressions of the operands. If the message expression, of the first operand is a subset of that of the second operand, then the difference operation returns objects from the first operand which are not in the second operand; the message expressions of the result are those of the first operand. Otherwise, it is handled as a projection of objects in the first operand on values that have no corresponding message expressions in the second operand. To illustrate the first case, the difference of the pairs corresponding to the *person* and *student* classes is the following pair: $\langle L_{instances}(person) \cup \{o_6, o_8\}, M_{expressions}(person) \rangle$. On the other hand, the second case is illustrated by considering the difference between the pairs corresponding to the *research-assistant* and *student* classes; the result is the following pair: $\langle W_{instances}(research-assistant), M_{expressions}(staff) \rangle$.

4.2 Object Algebra Expressions

An *object algebra expression* is a pair of sets, a set of objects and a set of message expressions. Since a class is defined a set of objects and a set of message expressions, it corresponds to an object algebra expression. Next, we will formally define object algebra expressions. When $len(x)$ is involved in any of the constraints (if-statements) given in the rest of this section, we will consider only message expressions x such that x returns a stored atomic value.

Definition 4.2 (Object Algebra Expressions) Let E be the set of object algebra expressions.

Being an object algebra expression, every element of set E must have a pair of sets – a set of objects and a set of message expressions. Formally,

$$\forall e \in E, \text{ both } M_{expressions}(e) \text{ and } W_{instances}(e) \text{ are defined.}$$

Given $e_1 \in E$ and $e_2 \in E$, let $M_{expressions}(e_1) = X_1$, $M_{expressions}(e_2) = X_2$, $W_{instances}(e_1) = T_1$, and $W_{instances}(e_2) = T_2$.

Elements of E are enumerated as follows:

- Given a class c_i , by definition, $M_{expressions}(c_i)$ and $W_{instances}(c_i)$ are both defined; hence, $c_i \in E$.

- *Selection*: Given a predicate expression p , $e_1[p] \in E$ with

$$M_{expressions}(e_1[p]) = M_{expressions}(e_1) = X_1,$$

$$W_{instances}(e_1[p]) = \{o \mid o \in T_1 \wedge p(o)\}, \text{ where } p(o) \text{ denotes the evaluation of predicate expression } p \text{ on object } o.$$

- *Projection*: Given $X \subseteq X_1$, $e_1[X] \in E$ with

$$M_{expressions}(e_1[X]) = X,$$

$$W_{instances}(e_1[X]) = W_{instances}(e_1).$$

The projection operation serves to drop the message expressions that are in $X_1 - X$ while preserving all the instances.

- *Cross-Product*: $(e_1 \times e_2) \in E$ with

$$M_{expressions}(e_1 \times e_2) = \begin{cases} (m_1 X_1) \cup (m_2 X_2)^6 & \text{if } \exists x_i \in X_1, \text{len}(x_i) = 1 \wedge \exists x_j \in X_2, \text{len}(x_j) = 1 \\ X_1 \cup (m_2 X_2) & \text{if } \forall x_i \in X_1, \text{len}(x_i) > 1 \wedge \exists x_j \in X_2, \text{len}(x_j) = 1 \\ (m_1 X_1) \cup X_2 & \text{if } \exists x_i \in X_1, \text{len}(x_i) = 1 \wedge \forall x_j \in X_2, \text{len}(x_j) > 1 \\ X_1 \cup X_2 & \text{if } \forall x_i \in X_1, \text{len}(x_i) > 1 \wedge \forall x_j \in X_2, \text{len}(x_j) > 1, \end{cases}$$

where m_1 and m_2 are two messages returning values (object identifiers) drawn from the domains T_1 and T_2 , respectively.

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

where \bullet indicates a concatenation of the two arguments; it is commutative.

- *Union*: $(e_1 \cup e_2) \in E$

⁶ $m_i X_i$ returns all the message expressions $m_i m_j m_k \dots m_n$ such that the sequence $m_j m_k \dots m_n$ is an element in the set of message expressions X_i .

$$\begin{aligned} M_{expressions}(e_1 \cup e_2) &= X_1 \cap X_2, \\ W_{instances}(e_1 \cup e_2) &= T_1 \cap T_2. \end{aligned}$$

Only message expressions in $X_1 \cap X_2$ return some values, including *nil* from objects in $T_1 \cup T_2$.

- *Difference*: $(e_1 - e_2) \in E$ with

$$\begin{aligned} M_{expressions}(e_1 - e_2) &= \begin{cases} X_1 & \text{if } X_1 \subseteq X_2 \text{ (by Lemma 3.1)} \\ X_1 - X_2 & \text{otherwise,} \end{cases} \\ W_{instances}(e_1 - e_2) &= \begin{cases} T_1 - T_2 & \text{if } X_1 \subseteq X_2 \\ T_1 & \text{otherwise.} \end{cases} \end{aligned}$$

- *Nest*: $(e_1 >> e_2) \in E$ with

$$\begin{aligned} M_{expressions}(e_1 >> e_2) &= X_1 \cup (m_2 X_2), \\ &\text{where } m_2 \text{ is a message such that the value returned by the message } m_2 \text{ from any object} \\ &\text{in the result of } (e_1 >> e_2) \text{ is drawn from the domain } T_2: \end{aligned}$$

$$W_{instances}(e_1 >> e_2) = \{o \mid \exists o_1 \in T_1 \wedge \text{value}(o) = \text{value}(o_1). \text{identity}(o_2)\}.$$

- *One level projection*: Given $X \subseteq X_1$, $e_1![X] \in E$ with

$$\begin{aligned} M_{expressions}(e_1![X]) &= \{x \mid \exists x_1 \in X \text{ with } x_1 \text{ returning a stored value, } x_1 = (x_2 m) \wedge \\ &\quad \text{len}(x_1) = \text{len}(x_2) + 1 \wedge \exists x_3 \in X_1 \wedge x_3 = (x_2 x) \wedge x = (m x_4)\} \\ &\cup \{x \mid \exists x_1 \in X \text{ with } x_1 \text{ returning a derived value, } \text{len}(x) = 1 \wedge \\ &\quad \forall o_1 \in W_{instances}(e_1) \exists o \in W_{instances}(e_1![X]) \text{ such that } o_1 x_1 = o x\}, \end{aligned}$$

$$W_{instances}(e_1![X]) = \{o \mid \exists o_1 \in T_1 \wedge \text{value}(o) = (o_1 X)\}, \text{ where } (o_1 X) \text{ is a set of values, each of which is the result of applying an element of } X \text{ to } o_1.$$

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*: Assume that $X \subseteq X_1$ and $x_i \in X_1$, and let f be an aggregate function. The aggregate function f is evaluated on the result of the message expression x_i for all objects in T_1 that return the same values for message expressions in X . In other words, objects in T_1 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 the values returned by the message expression x_i , applied to objects in each equivalence class. Consequently, $e_1 < X, f, x_i > \in E$ with $M_{expressions}(e_1 < X, f, x_i >) = (m_1 X_1) \cup \{m_3\}$,

where m_1 is a message which when sent to any object in the result, returns the object identifier of the related object in T_1 ; the value returned by message m_3 from any object in the result has a domain which is the range of the aggregate function f .

$$W_{instances}(e_1 < X, f, x_i >) = \{o \mid (o \ m_1) \subseteq T_1 \wedge (o \ m_3) = f(\{(o_1 \ x_i) \mid o_1 \in T_1 \wedge \forall o_2 \in (o \ m_1), (o_2 \ X) = (o_1 \ X)\})\} \quad \square$$

The following object algebra expressions are defined in terms of the primitive ones given in Definition 4.2.

- *Unnest*: defined in terms of projection so as to drop an existing relationship between two object algebra expressions. It leaves the projected out components in objects but renders them inaccessible:

$$(e_1 \ll e_2) = e_1[X_1 - X \mid X = (m_2 \ X_2) \wedge \forall o_1 \in T_1 (o_1 \ m_2) \in T_2].$$

We project on all the 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 projection*: to add a subset X of $M_{expressions}(e_2)$ to $M_{expressions}(e_1)$, first e_1 and e_2 are nested, then a one level projection is done to have all $M_{expressions}(e_2)$ and $M_{expressions}(e_1)$ together forming one set; after that projection of the result on $M_{expressions}(e_1) \cup X$ is done to get the target set of message expressions in the resulting pair.

$$e_1[X] = (e_1 \gg e_2)![W_{behavior}(e_1) \cup (m_2 \ W_{behavior}(e_2))] [M_{expressions}(e_1) \cup X]$$

where m_2 is a message in the result of $e_1 \gg e_2$ with its domain being $W_{instances}(e_2)$.

- *Join*: defined in terms of the cross-product or nest combined with selection. □

Using operations of the query language, objects may be constructed from 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 exists due to the nest operation, which takes two operands and extends each object in the first one to include a value referencing object(s) in the second operand, while a derived value exists due to the inverse of the project operation, which extends the behavior of objects in the operand without affecting their states. In contrast, the one-level-project operation constructs new objects from existing objects by collecting values found in different levels of nesting. Also, the fourth case in the definition of the cross-product operation results in new objects while the other three cases introduce new relationships.

4.3 From an Object Algebra Expression to a Class

Having formally defined object algebra expressions, we claim that every object algebra expression has the characteristics of a class, and this follows from the lemmata given in this section. However, before going into the details of the lemmata, it is important to recall that, as stated in section 3, by definition, a class has a set of superclasses, a set of attributes,

a set of methods and a set of objects. According to Definition , an object algebra expression has a set of objects and a set of message expressions. In addition, given a class c , $\text{methods}(c)$ and $W_{\text{attributes}}(c)$ are defined to include methods and attributes of superclasses of class c . Therefore, finding methods and attributes of a class implicitly leads to the set of its superclasses. Furthermore, a method implements a specific function, and it is invoked via a corresponding message. Consequently, for every method, there exists a corresponding message; therefore, finding a set of messages for an object algebra expression is equivalent to finding a set of methods; the corresponding method of a given message m is the one which is invoked by message m when used from the current class. 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 attributes 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 $M_{\text{expressions}}(e_1) = X_1$ and $M_{\text{expressions}}(e_2) = X_2$. According to Definition 4.2, a class is an object algebra expression. In other words, some object algebra expressions are classes. Thus, assume that $W_{\text{attributes}}(e_1)$, $W_{\text{attributes}}(e_2)$, $W_{\text{behavior}}(e_1)$ and $W_{\text{behavior}}(e_2)$ are all defined. Based on this assumption, we have the following Lemmata, 4.1 to 4.9, leading to the sets of messages and attributes of other object algebra expressions, and this leads to the fact that every object algebra expression corresponds to a class. Lemmata 4.1 to 4.9 have similar proofs. Therefore, only the proofs of Lemmata 4.1 and 4.3 are given; the others can be proved similarly.

In general, the set of messages of the class corresponding to an object algebra expression includes all the messages which prefix a message expression of that object algebra expression. Including more messages leads, according to Definition 3.3, to a set of message expressions which is a superset of the actual set of message expressions of the object algebra expression. On the other hand, dropping a message which prefixes some message expressions of the object algebra expression results in the set of message expressions as a subset of the actual set of message expressions. Concerning the attributes, for every stored value in the objects of an object algebra expression, an attribute is included in the set of attributes of the corresponding class such that the set of messages of that class contains a message which can to retrieve that stored value. For example, the objects in the result of a selection operation are a subset of those of the operand. Consequently, the class corresponding to the result shares the same messages and attributes with that corresponding to the operand. This is proved in Lemma 4.1, given below. For the other operations, this is formally stated in Lemmata 4.1 to 4.9, where messages and attributes of the object algebra expressions given in Definition 4.2 are specified.

Lemma 4.1 Messages and attributes of $e_1[p]$, where p is a predicate expression:

$$\begin{aligned} M_{\text{expressions}}(e_1[p]) = X_1 \Rightarrow \\ \cdot W_{\text{behavior}}(e_1[p]) = W_{\text{behavior}}(e_1), \\ \cdot W_{\text{attributes}}(e_1[p]) = W_{\text{attributes}}(e_1). \end{aligned}$$

Proof:

$$\begin{aligned} m \in W_{\text{behavior}}(e_1[p]) \Leftrightarrow \exists x \in M_{\text{expressions}}(e_1[p]), \text{ such that } x = m \ x_i \text{ (by Definition 3.3)} \\ \text{(but } M_{\text{expressions}}(e_1[p]) = M_{\text{expressions}}(e_1), \text{ by Definition 4.2)} \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow \exists x \in M_{expressions}(e_1), \text{ such that } x = m \ x_i \\ &\Leftrightarrow m \in W_{behavior}(e_1) \text{ (by Definition 3.3).} \end{aligned}$$

Therefore, $m \in W_{behavior}(e_1[p]) \Leftrightarrow m \in W_{behavior}(e_1)$.

Hence, $W_{behavior}(e_1[p]) = W_{behavior}(e_1)$

and

$$\begin{aligned} iv \in W_{attributes}(e_1[p]) &\Leftrightarrow \exists m \in W_{behavior}(e_1[p]) \text{ such that,} \\ &\quad \text{given } o \in W_{instances}(e_1[p]), o \text{ satisfies } (o \ m) = value(iv) \\ &\quad \text{(but } W_{behavior}(e_1[p]) = W_{behavior}(e_1), \text{ already proved} \\ &\quad \text{and } W_{instances}(e_1[p]) \subseteq W_{instances}(e_1) \text{ by Definition 4.2)} \\ &\Leftrightarrow iv \in W_{attributes}(e_1). \end{aligned}$$

Therefore, $iv \in W_{attributes}(e_1[p]) \Leftrightarrow iv \in W_{attributes}(e_1)$.

Hence, $W_{attributes}(e_1[p]) = W_{attributes}(e_1)$. \square

Before going into the details of the lemma that leads to the messages and attributes of $e_1[X]$, consider the following algorithm that derives the attributes of $e_1[X]$.

Algorithm 4.1 attributes of $e_1[X]$:

```

0. for every  $m_i \in W_{behavior}(e_1)$ 
1.   Let  $X_i \subseteq M_E$  such that  $(m_i \ X_i) \subseteq X$ 
   /*  $M_E$  denotes the set of all message expressions, i.e., for any class  $c$ ,  $M_{expressions}(c) \subseteq M_E$ 
2.     if  $X_i \neq \emptyset$  then
   /* the attribute that corresponds to  $m$  has a non-atomic domain
3.       if  $\exists iv_i \in W_{attributes}(e_1)$  such that  $X_i = M_{expressions}(OAE(domain(iv_i)))$ 7 then
4.          $iv_i \in W_{attributes}(e_1[X])$ 
5.       elseif  $\exists iv_i \in W_{attributes}(e_1)$  such that  $X_i \subseteq M_{expressions}(OAE(domain(iv_i)))$  then
6.          $iv_i \in W_{attributes}(e_1[X])$  and  $domain(iv_i)$  in  $e_1[X]$  is:
7.            $domain(iv_i) := W_{instances}(\langle domain(iv_i), M_{expressions}(OAE(domain(iv_i))) \rangle [X_i])$ 
8.         endif
9.       elseif  $\exists iv_i \in W_{attributes}(e_1)$  such that, given  $o \in W_{instances}(e_1)$ ,  $value(iv_i, o)$ 8 =  $(o \ m_i)$  then
   /* message  $m_i$  corresponds to the attribute  $iv_i$  that has an atomic domain
10.         $iv_i \in W_{attributes}(e_1[X])$ 
11.      endif
12.    endfor

```

Lemma 4.2 Algorithm 4.1 returns attributes of $e_1[X]$

⁷ Evaluating an object algebra expression e leads to the pair $\langle W_{instances}(e), M_{expressions}(e) \rangle$, and $OAE(W_{instances}(e))$ denotes the object algebra expression e .

⁸ Returns the value of the attribute iv_i in object o .

Proof: The *for-loop* in step 0 of Algorithm 4.1 iterates over all the messages in $W_{behavior}(e_1)$ to determine those corresponding to the attributes in $W_{attributes}(e_1)$. When such a message m_i is found, the corresponding attribute iv_i is added to $W_{attributes}(e_1[X])$ with its domain specified depending on the results of the *if-statements* where:

In step 1, from X , a subset that has all the message expressions starting with m_i , i.e., $m_i X_i$, is determined. The tag X_i is considered in step 2 for being non-empty; accordingly a non-atomic domain is determined for the attribute iv_i added to $W_{attributes}(e_1[X])$. In step 4, iv_i in $W_{attributes}(e_1[X])$ has a domain that is the same as iv_i in $W_{attributes}(e_1)$ because X_i happens to be $M_{expressions}(OAE(domain(iv_i)))$. For the other case, i.e., $X_i \in M_{expressions}(OAE(domain(iv_i)))$, in steps 6 and 7, the domain of iv_i is specified as the result of the projection of $\langle domain(iv_i), M_{expressions}(OAE(domain(iv_i))) \rangle$ on X_i , a subset of its message expressions. Since the class which corresponds to the result of the projection was proved in [11] to be a superclass of that which corresponds to the operand, the domain of iv_i in $W_{attributes}(e_1[X])$ is a superclass of the domain of iv_i in $W_{attributes}(e_1)$. In step 10, the domain of iv_i in $W_{attributes}(e_1[X])$ is determined to be the same as the domain of iv_i in $W_{attributes}(e_1)$ when $domain(iv_i)$ is atomic. \square

Lemma 4.3 Messages and attributes of $e_1[X]$: Given $X \subseteq X_1$,

- $W_{behavior}(e_1[X]) = \{m \mid m \in W_{behavior}(e_1) \wedge \exists x \in X \text{ with } x = m \ x_i\}$
- $W_{attributes}(e_1[X])$ is derived by Algorithm 4.1 as interpreted in Lemma 4.2. \square

Lemma 4.4 Messages and attributes of $e_1 \times e_2$:

case 1: If $\exists x_1 \in X_1, \text{len}(x_1) = 1 \wedge \exists x_2 \in X_2, \text{len}(x_2) = 1$, then

$$M_{expressions}(e_1 \times e_2) = (m_1 X_1) \cup (m_2 X_2) \Rightarrow$$

- $W_{behavior}(e_1 \times e_2) = \{m_1, m_2\}$
- $W_{attributes}(e_1 \times e_2) = \{iv_1, iv_2\}$,

$$\text{where } domain(iv_1) = 2^{W_{instances}(e_1)} \text{ and } domain(iv_2) = 2^{W_{instances}(e_2)}.$$

case 2: If $\forall x_1 \in X_1, \text{len}(x_1) > 1 \wedge \exists x_2 \in X_2, \text{len}(x_2) = 1$, then

$$M_{expressions}(e_1 \times e_2) = X_1 \cup (m_2 X_2) \Rightarrow$$

- $W_{behavior}(e_1 \times e_2) = messages(e_1) \cup \{m_2\}$
- $W_{attributes}(e_1 \times e_2) = W_{attributes}(e_1) \cup \{iv_2\}$,

$$\text{where } domain(iv_2) = 2^{W_{instances}(e_2)}.$$

case 3: If $\exists x_1 \in X_1, \text{len}(x_1) = 1 \wedge \forall x_2 \in X_2, \text{len}(x_2) > 1$, then

$$M_{expressions}(e_1 \times e_2) = (m_1 X_1) \cup X_2 \Rightarrow$$

- $W_{behavior}(e_1 \times e_2) = \{m_1\} \cup W_{behavior}(e_2)$
- $W_{attributes}(e_1 \times e_2) = \{iv_1\} \cup W_{attributes}(e_2)$,

$$\text{where } domain(iv_1) = 2^{W_{instances}(e_1)}.$$

case 4: If $\forall x_1 \in X_1, \text{len}(x_1) > 1 \wedge \forall x_2 \in X_2, \text{len}(x_2) > 1$, then

$$M_{expressions}(e_1 \times e_2) = X_1 \cup X_2 \Rightarrow$$

- $W_{behavior}(e_1 \times e_2) = W_{behavior}(e_1) \cup W_{behavior}(e_2)$
- $W_{attributes}(e_1 \times e_2) = W_{attributes}(e_1) \cup W_{attributes}(e_2)$.

Lemma 4.5 Messages and attributes of $e_1 \cup e_2$:

$$\begin{aligned}
 M_{expressions}(e_1 \cup e_2) &= X_1 \cap X_2 \Rightarrow \\
 &\cdot W_{behavior}(e_1 \cup e_2) = W_{behavior}(e_1) \cap W_{behavior}(e_2) \\
 &\cdot W_{attributes}(e_1 \cup e_2) = W_{attributes}(e_1) \cap W_{attributes}(e_2).
 \end{aligned}$$

Lemma 4.6 Messages and attributes of $e_1 - e_2$:

case 1: If $X_1 \subseteq X_2$, then

$$\begin{aligned}
 M_{expressions}(e_1 - e_2) &= X_1 \Rightarrow \\
 &\cdot W_{behavior}(e_1 - e_2) = W_{behavior}(e_1) \\
 &\cdot W_{attributes}(e_1 - e_2) = W_{attributes}(e_1).
 \end{aligned}$$

case 2: If $X_1 \not\subseteq X_2$, then

$$\begin{aligned}
 M_{expressions}(e_1 - e_2) &= X_1 - X_2 \Rightarrow \\
 &\cdot W_{behavior}(e_1 - e_2) = W_{behavior}(e_1) - W_{behavior}(e_2) \\
 &\cdot W_{attributes}(e_1 - e_2) = W_{attributes}(e_1) - W_{attributes}(e_2).
 \end{aligned}$$

Lemma 4.7 Messages and attributes of $e_1 \gg e_2$:

$$\begin{aligned}
 M_{expressions}(e_1 \gg e_2) &= X_1 \cup (m_2 X_2) \Rightarrow \\
 &\cdot W_{behavior}(e_1 \gg e_2) = W_{behavior}(e_1) \cup \{m_2\} \\
 &\cdot W_{attributes}(e_1 \gg e_2) = W_{attributes}(e_1) \cup \{iv_1\}, \text{ where } domain(iv_1) = 2^{W_{instances}(e_2)}.
 \end{aligned}$$

Lemma 4.8 Messages and attributes of $e_1![X]$: given $X \subseteq X_1$,

$$\begin{aligned}
 M_{expressions}(e_1![X]) &\text{ given in Definition 4.2 } \Rightarrow \\
 &\cdot W_{behavior}(e_1![X]) = \{m \mid \exists x \in M_{expressions}(e_1![X]) \text{ with } x = m x_j\} \\
 &\cdot W_{attributes}(e_1![X]) = \{iv \mid domain(iv) = d \wedge \forall o \in W_{instances}(e_1![X]) \\
 &\quad \exists m \in W_{behavior}(e_1![X]) \text{ with } (o m) \in d\}.
 \end{aligned}$$

Lemma 4.9 Messages and attributes of $e_1 \langle X, f, x_i \rangle$: given $X \subseteq X_1$ and $x_i \in X_1$,

$$\begin{aligned}
 M_{expressions}(e_1 \langle X, f, x_i \rangle) &\text{ given in Definition 4.2 } \Rightarrow \\
 &\cdot W_{behavior}(e_1 \langle X, f, x_i \rangle) = \{m_1, m_3\} \\
 &\cdot W_{attributes}(e_1 \langle X, f, x_i \rangle) = \{iv_1, iv_2\}, \\
 &\text{where } domain(iv_1) = W_{instances}(e_1) \text{ and } domain(iv_2) = \text{the domain of the result of } f.
 \end{aligned}$$

The proofs of Lemmata 4.3 to 4.9 are omitted as they are similar to the proof of Lemma 4.1. Informally, since every object algebra expression has a set of message expressions, then by considering message expressions of length one, the set of messages can be derived. Furthermore, the fact that, by definition, every attribute has a corresponding message leads to the derivation of the set of instance variables of an object algebra expression. This is done by collecting from the operand those attributes having a corresponding message in the already derived set of messages.

Combining Definition 4.2 and Lemmata 4.1 to 4.9, every object algebra expression has a set of objects, a set of messages and a set of attributes. The set of messages leads to the set of methods because every message has a corresponding method. Therefore, an object algebra expression has the characteristics of a class leading to the following corollary.

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

Proof:

$W_{instances}(e)$ are given by 4.2;

$W_{attributes}(e)$ and $W_{behavior}(e)$ are given by Lemmata 4.1 to 4.9;

Classes in $C_p(e)$ are determined by considering the relationship between $W_{instances}(e)$, and $M_{expressions}(e)$ and those of the operand(s) as explained in [11].

Therefore, since it has the characteristics of a class, e is, in fact, a class. □

4.4 Inheritance Relationship

One of the distinguishing features of object-oriented systems is inheritance. Inheritance leads to reusability where a class c_1 uses the facilities of its superclass c_2 as if those facilities were defined within the class c_1 .

In general, object-oriented database systems support multiple inheritance. Multiple inheritance is considered advantageous over and includes simple inheritance. In multiple inheritance, reusability is achieved to a greater degree than it is in simple inheritance. Therefore, supporting multiple inheritance in a data model helps to increase reusability because it provides flexibility in increasing the number of superclasses if required in a way to increase the facilities that a class c inherits, and, hence to decrease the facilities defined inside class c without being inherited. This is one of the reasons for supporting multiple inheritance in the data model described in section 3.

In the rest of this section, we will determine the inheritance relationship between *object algebra expressions*. Based on this relationship, we will prove that an object algebra expression e inheriting from other object algebra expressions e_1, e_2, \dots, e_n , enables the corresponding class c to inherit from classes c_1, c_2, \dots, c_n . This completes the proof of corollary 4.1, where $C_p(c)$ are determined based on Lemmata 4.10 to 4.17 and Theorem 4.1 given below.

Definition 4.3 Partial Ordering (\leq_e) among object algebra expressions

Given two object algebra expressions e_1 and e_2 , we say that e_1 inherits from e_2 , i.e., $e_1 \leq_e e_2$ iff:

1. $M_{expressions}(e_2) \subseteq M_{expressions}(e_1)$,
2. $W_{instances}(e_1) \subseteq W_{instances}(e_2)$. □

Notice that Definition 4.3 considers only message expressions and total instances which are the characteristics known for an object algebra expression based on Definition 4.2. Other characteristics leading to a class and given in Lemmata 4.1 to 4.9 are not considered.

Now that every object algebra expression corresponds to a class by Corollary 4.1, it is necessary to decide on the inheritance relationship between the class that corresponds to a given object algebra expression and other existing classes. It is enough to decide on the inheritance relationship between object algebra expressions because Theorem 4.1 leads to the inheritance relationship between the corresponding classes. Based on Definition 4.3, the following Lemmata 4.10 to 4.17 lead to the inheritance relationship between object algebra expressions. Based on this relationship, the inheritance relationship between the corresponding classes can be determined.

Given two object algebra expressions e_1 and e_2 , let $M_{expressions}(e_1) = X_1$ and $M_{expressions}(e_2) = X_2$. Lemmata 4.10 to 4.17 give the inheritance relationship between object algebra expressions. This method is used because the proofs of Lemmata 4.10 to 4.17 follow straightforwardly from Definitions 4.2 and 4.3, as illustrated by the proofs of Lemmata 4.10 and 4.11. From Lemmata 4.10 to 4.17 and based on Theorem 4.1, $C_p(c)$ for class c that corresponds to a given object algebra expression e can be determined.

Lemma 4.10 Inheritance relationship of $e_1[p]$ with e_1 , where p is a predicate expression:

$$e_1[p] \leq_e e_1.$$

Proof: (By definition)

$$\begin{aligned} M_{expressions}(e_1[p]) &= M_{expressions}(e_1) \text{ and } W_{instances}(e_1[p]) \subseteq W_{instances}(e_1) \text{ (by Definition 4.2)} \\ \Leftrightarrow e_1[p] &\leq_e e_1 \text{ (by Definition 4.3).} \quad \square \end{aligned}$$

Lemma 4.11 Inheritance relationship of $e_1[X]$ with e_1 , where $X \subseteq X_1$, $e_1 \leq_e e_1[X]$.

Proof: (By definition)

$$\begin{aligned} M_{expressions}(e_1[X]) &\subseteq M_{expressions}(e_1) \text{ and } W_{instances}(e_1[X]) = W_{instances}(e_1) \text{ (by Definition 4.2)} \\ \Leftrightarrow e_1 &\leq_e e_1[X] \text{ (by Definition 4.3).} \quad \square \end{aligned}$$

Lemma 4.12 Inheritance relationship of $e_1 \times e_2$ with e_1 and e_2 :

- if $\exists x_1 \in X_1, \text{len}(x_1) = 1 \wedge \exists x_2 \in X_2, \text{len}(x_2) = 1$, then
 $(e_1 \times e_2) \not\leq_e e_1$ and $(e_1 \times e_2) \not\leq_e e_2$;
- if $\forall x_1 \in X_1, \text{len}(x_1) > 1 \wedge \exists x_2 \in X_2, \text{len}(x_2) = 1$, then
 $(e_1 \times e_2) \leq_e e_1$
- if $\exists x_1 \in X_1, \text{len}(x_1) = 1 \wedge \forall x_2 \in X_2, \text{len}(x_2) > 1$ then
 $(e_1 \times e_2) \leq_e e_2$;
- if $\forall x_1 \in X_1, \text{len}(x_1) > 1 \wedge \forall x_2 \in X_2, \text{len}(x_2) > 1$, then
 $(e_1 \times e_2) \leq_e e_1$ and $(e_1 \times e_2) \leq_e e_2$. □

Lemma 4.13 Inheritance relationship of $e_1 \cup e_2$ with e_1 and e_2 :

$$e_1 \leq_e (e_1 \cup e_2) \text{ and } e_2 \leq_e (e_1 \cup e_2). \quad \square$$

Lemma 4.14 Inheritance relationship of $e_1 - e_2$ with e_1 and e_2 :

- if $X_1 \subseteq X_2$, then
 $(e_1 - e_2) \leq_e e_1$;
- if $X_1 \not\subseteq X_2$, then
 $e_1 \leq_e (e_1 - e_2)$. □

Lemma 4.15 Inheritance relationship of $(e_1 \gg e_2)$ with e_1 :

$$(e_1 \gg e_2) \leq_e e_1 \quad \square$$

Lemma 4.16 Inheritance relationship of $e_1![X]$ with e_1 , where $X \subseteq X_1$,

$$e_1![X] \not\leq_e e_1 \text{ and } e_1 \not\leq_e e_1![X]. \quad \square$$

Lemma 4.17 Inheritance relationship of $e_1 \langle X, f, x_i \rangle$ with e_1 , where $X \subseteq X_1$ and $x_i \in X_1$:

$$e_1 \langle X, f, x_i \rangle \not\leq_e e_1 \text{ and } e_1 \not\leq_e e_1 \langle X, f, x_i \rangle. \quad \square$$

Although omitted, the proofs of Lemmata 4.12 to 4.17 follow from Definitions 4.2 and 4.3 in the same way as the proofs of Lemmata 4.10 to 4.11 do. After deciding on the inheritance relationship between object algebra expressions, Theorem 4.1 leads to the inheritance relationship among the corresponding classes. First, the OBJECT class is assumed to be the superclass of any class c that corresponds to an object algebra expression resulting from a query; then, other user defined classes can be included in $C_p(c)$ by applying the appropriate lemma out of Lemmata 4.10 to 4.17 followed by application of Theorem 4.1. In other words, given an object algebra expression, at the start, I of the corresponding class is assumed to be empty. However, the following theorem is considered as a first step towards maximizing I and, hence, minimizing L by deriving the inheritance relationship between classes based on the inheritance relationship between the corresponding object algebra expressions, according to Lemmata 4.10 to 4.17.

Theorem 4.1 Let e_1 and e_2 be two object algebra expressions with c_1 and c_2 being their corresponding classes by Corollary 4.1, respectively:

$$e_1 \leq_e e_2 \Leftrightarrow c_1 \leq_c c_2.$$

Proof: First of all, Lemmata 4.1 to 4.9 give $W_{behavior}(c_1)$ and $W_{behavior}(c_2)$;

$$\begin{aligned} e_1 \leq_e e_2 &\Leftrightarrow M_{expressions}(e_2) \subseteq M_{expressions}(e_1) \\ &\text{and } W_{instances}(e_1) \subseteq W_{instances}(e_2) \text{ (by Definition 4.3)} \\ &\Leftrightarrow W_{behavior}(c_2) \subseteq W_{behavior}(c_1) \text{ (by Lemma 3.1)} \end{aligned}$$

and

$$\begin{aligned} iv \in W_{attributes}(c_2) &\Leftrightarrow \exists m \in W_{behavior}(c_2) \text{ such that given } o \in W_{instances}(c_2), \\ & o \text{ satisfies } o = m = value(iv) \\ & \text{(but } W_{behavior}(c_2) \subseteq W_{behavior}(c_1), \text{ already proved)} \\ &\Rightarrow iv \in W_{attributes}(c_1) \\ &\Rightarrow W_{attributes}(c_2) \subseteq W_{attributes}(c_1) \end{aligned}$$

Therefore, $e_1 \leq_e e_2 \Leftrightarrow W_{behavior}(c_2) \subseteq W_{behavior}(c_1) \wedge W_{attributes}(c_2) \subseteq W_{attributes}(c_1)$.

Hence, $c_1 \leq_c c_2$. \square

Theorem 4.1 determines any classes which should constitute $C_p(c)$ for a given class c , which corresponds to an object algebra expression e .

5. ILLUSTRATIVE EXAMPLES

In this section, several examples are included to illustrate the distinguishing aspects of the query model presented in this paper. The examples given in this section are based on the classes presented in section 3.

Example 5.1 Select 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. More than one variable may range over the objects of an operand. For example, $student\%s_1\%s_2$ indicates that s_1 and s_2 range over the objects of the *student* class. The use of = calls for an evaluation of this query on a temporary basis. Thus, the resulting pair S_1 consists of the sets $W_{instances}(S_1) = \{o_5, o_9\}$ and $M_{expressions}(S_1) = M_{expressions}(student)$. Notice that the student with object identity o_7 is not included in $W_{instances}(S_1)$ because he/she does not attend the course “CS565”.

We differentiate between temporary and persistent evaluations of a query, where an assignment free query is always evaluated on a temporary basis. 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 finding additional class characteristics of the determined pair using Lemmata 4.1 to 4.9.

Example 5.2 Find brothers of ‘Adams’:

$$person\%p_1[p_1\ sex() = “M” \wedge \exists p_2 \in W_{instances}(person) \wedge p_2\ name() = “Adams” \wedge \exists p_3 \in W_{instances}(person) \wedge \{p_1, p_2\} \subseteq p_3\ children()].$$

The predicate expression in Example 5.2 guarantees that the objects in the result will be solely those who are sons of the parent of “Adams”, hence, the output is the pair $\langle \{o_6\} M_{expressions}(person) \rangle$, while the operand is the pair $\langle W_{instances}(person), M_{expressions}(person) \rangle$.

Example 5.3 Assume that the student class is not present in the lattice and define the *research-assistant* class is as:

$$research-assistant \langle \{staff\}, year: integer, courses: course \rangle$$

To derive the *student* class as a persistent class and assuming that a student attends the department (s)he works in, the *research-assistant* class is projected with respect to a set of messages as follows:

$$student := research-assistant[\{name(), age(), sex(), children(), year(), courses(), works-in() \rightarrow student-in()\}].$$

The subset $\{name(), age(), sex(), children()\}$ of the projection set can be replaced by $W_{behavior}(person)$, which is its implicit representation. Consequently, the query can be coded as

$$student := research-assistant[\{year(), courses(), works-in() \rightarrow student-in()\} \cup W_{behavior}(person)].$$

By definition, $W_{instances}(student) = W_{instances}(research-assistant)$, and by Lemma 4.3,

$$\begin{aligned} W_{behavior}(student) &= \{name(), age(), sex(), children(), year(), courses(), student-in()\}, \\ W_{attributes}(student) &= \{name: string, age: integer, sex: ["M", "F"], children: person, year: integer, courses: course, student-in: department\}. \end{aligned}$$

In [11], we showed how the derived *student* class will be recognized as a subclass of the *person* class and naturally placed in the lattice so that $C_p(student) = \{person\}$ with $W_{attributes}(c)$ and $W_{behavior}(c)$ is adjusted accordingly. Also, the *student* class is added to the superclasses of the *research-assistant* class and, hence, is changed to, $C_p(research-assistant) = \{student, staff\}$.

Example 5.4 Find the names and net salaries of staff members by deducting taxes at the rate $t = 0.1$:

$$staff ![\{name(), net-salary(0.1) \rightarrow m()\}]$$

Each object in the output includes a value which is returned by the message $m()$ and derived by $net-salary(0.1)$ from the stored value *salary* in the corresponding object of the operand. The operand is the pair $\langle W_{instances}(staff), M_{expressions}(staff) \rangle$ while the output is the pair $\langle \{ \langle "Smith", 45K \rangle, \langle "Adams", 37.8K \rangle, \langle "George", 13.5K \rangle \}, \{name(), m()\} \rangle$.

Example 5.5 Find pairs of students attending the same courses

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

The result of this query is the pair $\langle \langle o_5, o_9 \rangle, (m_1() M_{expressions}(student)) \cup (m_2() M_{expressions}(student)) \rangle$

Here, $m_1()$ and $m_2()$ are two new messages added to return the first and second components of a receiving object in the resulting pair, respectively.

Notice that the result of the query in Example 5.5 will be a direct subclass of the root because the *student* class has some attributes with atomic domains. However, using a 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 appends to every student a set of identities of related students, the cross-product operation forms new values, each consisting of the identity of a student together with the identity of a related student.

Example 5.6 Assume that the *person* class is not present in the lattice and define the *student* and the *staff* classes are defined as follows:

$$\begin{aligned} \textit{student} &<\phi, \textit{name:string}, \textit{age:integer}, \textit{sex}:[\textit{"M"}, \textit{"F"}], \textit{children:person}, \textit{year:integer}, \\ &\quad \textit{courses:course}, \textit{student-in:department}>, \\ \textit{staff} &<\phi, \textit{name:string}, \textit{age:integer}, \textit{sex}:[\textit{"M"}, \textit{"F"}], \textit{children:person}, \textit{salary:integer}, \\ &\quad \textit{works-in:department}>. \end{aligned}$$

The person class is derived as $\textit{person} := \textit{student} \cup \textit{staff}$.

$$\begin{aligned} \text{By definition, } W_{\textit{instances}}(\textit{person}) &= W_{\textit{instances}}(\textit{student}) \cup W_{\textit{instances}}(\textit{staff}), \\ M_{\textit{expressions}}(\textit{person}) &= M_{\textit{expressions}}(\textit{student}) \cap M_{\textit{expressions}}(\textit{staff}). \end{aligned}$$

$$\begin{aligned} \text{By Lemma 4.5, } W_{\textit{attributes}}(\textit{person}) &= \{\textit{name:string}, \textit{age:integer}, \textit{sex}:[\textit{"M"}, \textit{"F"}], \textit{children:} \\ &\quad \textit{person}\}, \\ W_{\textit{behavior}}(\textit{person}) &= \{\textit{name}(), \textit{age}(), \textit{sex}(), \textit{children}()\}. \end{aligned}$$

We showed how the *person* class is recognized as a superclass of both the *student* and *staff* classes in [11].

Example 5.7 Find students who are not research assistants:

$$\textit{student} - \textit{research-assistant}.$$

Thus, the output pair of this query is $\langle \{o_5, o_7\}, M_{\textit{expressions}}(\textit{student}) \rangle$, according to Definition 4.2, since $M_{\textit{expressions}}(\textit{student}) \subseteq M_{\textit{expressions}}(\textit{research-assistant})$.

Example 5.8 In Example 5.4, it was assumed that $t = 0.1$ is fixed for all staff members. In this example, we assume that $t = 0.1$ for *research-assistants*, and that $t = 0.15$ for other staff members. To find the names and net salaries of staff members, we write

$$\begin{aligned} &(\textit{staff} - \textit{research-assistant}) ![\{\textit{name}(), \textit{net-salary}(0.15)\}], \\ &\cup \textit{research-assistant} ![\{\textit{name}(), \textit{net-salary}(0.1)\}]. \end{aligned}$$

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

Example 5.9 Find staff members earning more than the average salary in their department:

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

where $m()$ and $\text{avsalary}()$ are the two messages in the result of the aggregate function application operation used to return the set of object identities of staff members working in the same department and the corresponding calculated average salary, respectively. The message $\text{avsalary}()$ is a concatenation of the first two letters of the applied function, average , with the last message in the used message expression, here $\text{salary}()$. The staff class is nested with the result of the application of the aggregate function average on staff members grouped by $\text{works-in}()$. In other words, first, the set $W_{\text{instances}}(\text{staff})$ is partitioned into equivalence classes based on the result of the message expressions in the set $\{ \text{works-in}() \}$ by collecting the same equivalence class staff members working for the same department. The second step is application of the message expression $\text{salary}()$ to every object to get the corresponding salary; then, the aggregate function average is applied to get the average salary for objects in every equivalence class, which leads to the pair

$$\langle \{ o_{17} < \{ o_6, o_8, o_9 \}, 41.667K \rangle, \{ m(), \text{avsalary}() \} \rangle.$$

The staff class is nested with this pair resulting from the aggregate function application operation to get the pair:

$$\begin{aligned} & \langle \langle \langle \text{“Smith”}, 45, \text{“M”}, \{ o_1, o_7 \}, 50K, o_{10}, o_{17} \rangle, \langle \text{“Adams”}, 40, \text{“M”}, \phi, 60K, o_{10}, o_{17} \rangle, \\ & \langle \text{“George”}, 22, \text{“M”}, \phi, 15K, o_{10}, o_{17} \rangle, \}, M_{\text{expressions}}(\text{staff}) \cup \{ m_1(), m(), m_1(), \text{avsalary}() \} \rangle, \end{aligned}$$

where $m_1()$ is a message added to the result of the nest operation to facilitate reaching the related objects in the pair resulting from the aggregate function application operation. Then those staff members satisfying the given predicate expression are selected, and finally, projection on $\{ \text{name}() \}$ is performed. The overall result of this query is the pair

$$\begin{aligned} & \langle \langle \langle \text{“Smith”}, 45, \text{“M”}, \{ o_1, o_7 \}, 50K, o_{10}, o_{17} \rangle, \\ & \langle \text{“Adams”}, 40, \text{“M”}, \phi, 60K, o_{10}, o_{17} \rangle, \}, \{ \text{name}() \} \rangle. \end{aligned}$$

On the other hand, a recursive query is coded by allowing an object variable bound to a resulting object in the evaluation of a query to also appear in a predicate in that query as illustrated in the following examples.

Example 5.10 Find all descendants of “Smith”:

$$\begin{aligned} D \% d = \text{person} \% p [& (\exists p_1 \in W_{\text{instances}}(\text{person}) \wedge p_1 \text{name}() = \text{“Smith”} \wedge p \in p_1 \text{children}()) \\ & \vee (p \in d \text{children}())] \end{aligned}$$

In Example 5.10, the first part of the predicate expression, i.e., $(\exists p_1 \in W_{instances}(person) \wedge p_1 name() = "Smith" \wedge p \in p_1 children())$, is responsible for adding the children of "Smith" to the result D , which is prior to that empty. After that, the first part is ignored, and only the second part of the predicate expression, i.e., $(p \in d children())$, is recursively added to the result for the children of a person already added, until no more persons can be added. In this way, the obtained result D includes from the person class those members whose are the descendants of "Smith". The set of message expressions of the operand is retained in the result.

Example 5.11 Find all prerequisites of the course "CS450":

$$P\%p = course\%c[(\exists c_1 \in W_{instances}(course) \wedge c_1 code() = "CS450" \wedge c \in c_1 prerequisites()), \\ \vee (c \in p prerequisites())].$$

Example 5.12 Find all relatives of "Tom":

$$R\%r = person\%p[(p name() = "Tom") \vee (p \in r children()) \vee (r \in p children()) \\ \vee (\exists p_1 \in W_{instances}(person) \wedge \{p, r\} \subseteq p_1 children())]$$

6. CONCLUSIONS

In this paper, we have formally described a query model for object-oriented database systems. Our query model is not restricted to handling existing objects. The introduction of new relationships and the creation of new objects are also supported. We enable a new relationship to have a stored value by extending objects in the operand to include new values for the new attributes. We have also made it 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 so as 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 can itself be an operand; hence, the closure property is naturally maintained without introducing non-object-oriented constructs into the model.

The operators of our object algebra subsume those of the relational and nested relational algebras; hence, it is more powerful than either one. Uniform handling of objects as well as their behavior is an important requirement of an object algebra, and we have satisfied this requirement 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.

A message expression causes evaluation of the underlying methods in the same sequence as would occur if they all together formed a single method invoked by that message expression. Furthermore, message expressions are used in the invocation of behavior and

behavior constructors. Also, message expressions facilitate accessing of stored and derived values, thus achieving computational completeness without having an embedded query language with impedance mismatch.

We started by defining a set of objects and a set of message expressions for a class. With such a pair, a class was then shown to be an operand. Based on this, some operands are defined as 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, we proved that from such a pair, other class characteristics can be derived. Having the characteristics of a class, the output from a query is, in fact, a class. Thus, we can decide on the proper placement of such a class in the lattice.

Finally, we noted the possibility of supporting linear recursion without any need for a particular operator to serve this purpose. This significantly enhances the power of the described object algebra and is also very practical since many actual recursive queries are linear in nature. A future extension of the described object algebra could support general recursive queries. Currently, we are working on a visual query language on top of the presented related algebra. We believe that a visual query language is very important for multi-media applications.

REFERENCES

1. S. Abiteboul and C. Berri, "On the power of languages for the manipulation of complex objects," *VLDB Journal*, 1995, Vol. 4, No. 4, pp. 727-788.
2. 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, 1988, pp.389-417.
3. J. Banerjee, et al., "Data model issues for object-oriented applications," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987, pp. 3-26.
4. M. J. Carey, D. J. DeWitt, and S. L. Vandenberg, "A data model and a query language for EXODUS," in *Proceedings of ACM-SIGMOD Conference on Management of Data*, 1988, pp. 413-423.
5. D. H. Fishman, et al., "IRIS: an object-oriented database management system," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987, pp. 48-69.
6. M. F. Hornick and S. B. Zdonik, "A shared segmented memory system for an object-oriented database," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987, pp. 70-95.
7. D. Maier and J. Stein, "Development and implementation of an object-oriented DBMS," in B. Shriver and P. Wegner (eds), *Research Directions in Object-Oriented Programming*, MIT Press, MA, 1987.
8. E. Neuhold and M. Stonebraker, "Future directions in DBMS research," Technical Report, TR-88-001, Intl. Computer Science Inst., Berkeley, CA, 1988.
9. W. Kim, "A model of queries for object-oriented databases," in *Proceedings of the International Conference on Very Large Databases*, 1989, pp. 423-432.
10. R. Agrawal, "Alpha: an extension of relational algebra to express a class of recursive queries," *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, 1988, pp. 879-885.
11. R. Alhajj and F. Polat, "Proper handling of query results towards maximizing reusabil-

- ity in object-oriented databases,” *Information Sciences: An International Journal*, Vol. 107, No. 1-4, 1998, pp. 247-272.
12. R. Alhajj and M. E. Arkun, “A data model for object-oriented databases,” in *Proceedings of the International Symposium on Computers and Information Sciences*, 1991, pp. 203-213.
 13. R. Alhajj and M. E. Arkun, “A formal data model and object algebra for object-oriented databases,” *Applied Mathematics and Computer Science*, Vol. 2, No. 1, 1992, pp. 49-63.
 14. R. Alhajj and F. Polat, “Reusability and schema evolution in an object-oriented query model,” in *Proceedings of the ASME European Conference on Systems Design and Applications*, 1996, pp. 21-29.
 15. R. Alhajj and F. Polat, “Closure maintenance in an object-oriented query model,” in *Proceedings of the ACM International Conference on Information and Knowledge Management*, 1994, pp. 72-79.
 16. R. Alhajj and M.E. Arkun, “Queries in object-oriented database systems,” T. Finin, Y. Yesha and K. Nicholas (eds.), in *Information and Knowledge Management, Expanding the Definition of “Database”*, Vol. 752, Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1993, pp. 36-52.
 17. R. Alhajj and M. E. Arkun, “Object-oriented query language,” *Journal of Information and Software Technology*, Vol. 35, No. 9, 1993, pp. 519-529.
 18. R. Alhajj and M. E. Arkun, “A query model for object-oriented database systems,” in *Proceedings of the IEEE International Conference on Data Engineering*, 1993, pp. 163-172.
 19. A. Klug, “Equivalence of relational algebra and relational calculus query languages having aggregate functions,” *Journal of the ACM*, Vol. 29, No. 3, 1982, pp. 699-717.
 20. M. Stonebraker, et.al., “Third generation on database system manifesto” in *Proceedings of IFIP DS-4 Workshop on Object-Oriented Databases*, 1990, pp. 495-511.
 21. W. Kim, “Object-oriented databases: definition and research directions,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, 1990, pp. 327-341.
 22. J. Orenstein, et.al, “Query processing in the object store database system,” in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1992, pp. 403-412.
 23. L. A. Rowe and M. R. Stonebraker, “The postgres data model,” in *Proceedings of the International Conference on Very Large Databases*, 1987, pp. 83-96.
 24. S. Cluet, et. al., “Reloop, an algebra based query language for an object-oriented database system,” in *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1989, pp. 313-332.
 25. D. Florescu, L. Rachid, and P. Valduriez, “Answering queries using OQL view expressions,” in *Proceedings of the Workshop on Materialized Views, in cooperation with ACM-SIGMOD*, 1996, pp. 84-90.
 26. S. L. Vandenberg and D. J. DeWitt, “Algebraic support for complex objects with arrays, identity and inheritance,” Technical Report CS-TR-987, Department of Computer Science, University of Wisconsin-Madison, 1990.
 27. J. Banerjee, W. Kim, and K. C. Kim, “Queries in object-oriented databases,” in *Proceedings of IEEE International Conference on Data Engineering*, 1988, pp. 31-38.
 28. A. Alashqur, S. Y. Su, and H. Lam, “OQL: a query language for manipulating object-oriented databases,” in *Proceedings of the International Conference on Very Large*

- Databases*, 1989, pp. 433-442.
29. A. Alashqur, S. Y. Su, and H. Lam, "A rule based language for deductive object-oriented databases," in *Proceedings of IEEE International Conference on Data Engineering*, 1990, pp. 58-67.
 30. U. Dayal, "Queries and views in an object-oriented data model," in *Proceedings of the International Workshop on Database Programming Languages*, 1989, pp. 80-102.
 31. F. Manola and U. Dayal, "PDM: an object-oriented data model," in *Proceedings of the International Workshop on Object-Oriented Databases*, 1986, pp. 18-25.
 32. G. Shaw and S. Zdonik, "A query algebra for object-oriented databases," in *Proceedings of IEEE International Conference on Data Engineering*, 1990, pp. 154-162.
 33. A. Albano, L. Cardelli, and R. Orsini, "Galileo: a strongly-typed interactive conceptual language," *ACM Transactions on Database Systems*, Vol. 10, No. 2, 1985, pp. 230-260.
 34. F. Bancilhon, et.al., "FAD: a powerful and simple database language," in *Proceedings of the International Conference on Very Large Databases*, 1987, pp. 97-105.
 35. P. A. Boncz, A. N. Wilschut, and M. L. Kersten, "Flattening an object algebra to provide performance," in *Proceedings of IEEE International Conference on Data Engineering*, 1998, pp. 568-577.
 36. D. K. C. Chan, P. W. Trinder, and R.C. Welland, "Evaluating object-oriented query languages," *Computer Journal*, Vol. 37, No. 10, 1995, pp. 858-872.
 37. G. Gardarin and P. Valduriez, "ESQL2: an extended SQL2 with F-logic semantics," in *Proceedings of the 10th IEEE International Conference on Data Engineering*, 1992, pp. 320-327.
 38. S. Hibino and E. A. Rundensteiner, "Processing incremental multidimensional range queries in a direct manipulation visual query," in *Proceedings of IEEE International Conference on Data Engineering*, 1998, pp. 458-465.
 39. M. Kifer, W. Kim, and Y. Sagiv, "Querying object-oriented databases," in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1992, pp. 393-402.
 40. W. C. Lee and D. L. Lee, "Path dictionary: a new access method for query processing in object-oriented databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 3, 1998, pp. 371-388.
 41. L. Fegaras, "Query unnesting in object-oriented databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998, pp. 49-60.
 42. M. Chavda and P. T. Wood, "Towards an ODMG-compliant visual object query language," in *Proceedings of the International Conference on Very Large Databases*, 1997, pp. 456-465.
 43. U. Röhm and K. Böhm, "Working together in harmony – an implementation of the CORBA object query service and its evaluation," in *Proceedings of IEEE International Conference on Data Engineering*, 1999, pp. 238-247.
 44. E. A. Rundensteiner and L. Bic, "Set operations in object-based data models," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 3, 1992, pp.382-398.
 45. M. H. Scholl and H. J. Scheck, "A relational object model," in *Proceedings of the International Conference on Database Theory*, 1990, pp. 89-105.
 46. H. T. Siegelmann and B. R. Badrinath, "Integrating implicit answers with object-oriented queries," in *Proceedings of the International Conference on Very Large Databases*, 1991, pp. 15-24.

47. S. Y. W. Su, S. J. Hyun, and H. H. M. Chen , “Temporal association algebra: a mathematical foundation for processing object-oriented temporal databases,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 3, 1998, pp. 389-408.
48. D. Taniar and W. Rahayu, “Object-oriented collection join queries,” in *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, 1996, pp. 115-125.
49. H. Korth and X. Peltier, “Query algebra for object-oriented databases,” in *Proceedings of the Schlumberger Software Conference*, 1990.
50. M. Stonebraker, et. al., “The design and implementation of INGRES,” *ACM Transactions on Database Systems*, Vol. 1, No. 3, 1976, pp. 189-222.
51. S. L. Osborn, “Identity equality and query optimization,” in *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1988, pp. 346-351.
52. S. Cluet and C. Delobel, “Classification and optimization of nested queries in object Bases,” in *Proceedings of BDA '94*, 1994.
53. E. Bentino, et.al., “Object-oriented query languages: the notion and the issues,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 3, 1992, pp. 223-237.
54. R. Busse and P. Fankhauser, “Declarative and procedural object-oriented views,” in *Proceedings of IEEE International Conference on Data Engineering*, 1999, pp. 260.



Reda Alhadj received his BS degree in computer engineering in 1988 from Middle East Technical University, Ankara, Turkey. Latter, he obtained his MS and Ph.D. degrees in computer engineering and information sciences from Bilkent University, Ankara in 1990 and 1993, respectively. In 1995, Dr. Alhadj was promoted to associate professor by the Turkish Institute for Higher Education. He served as an assistant professor in the College of Computers and Information Sciences at King Saud University and two years at Sultan Qaboos University in Oman. Currently, he is an associate professor in the Department of Math & Computer Science at the American University of Sharjah, U.A.E. He published over 40 papers in refereed international journals and conferences. Dr. Alhadj's primary work and research interests are in the areas of object-oriented databases, data warehouses and view maintenance, multi-agent based query processing, schema unification and re-engineering of legacy databases, parallel algorithms, and pattern recognition of hand-written Arabic characters and Hindi numerals.