

A Query Algebra for an Extended Object-Oriented Database Model

George Loizou and Philippos Pouyioutas

*Department of Computer Science, Birkbeck College, University of London,
Malet Street, London WC1E 7HX, UK*

Abstract: In this paper we extend the *object-oriented database model* via the concept of *object inheritance*. We then define a *closed and safe object-oriented query algebra*. Our algebra is different from any other object-oriented query algebra found in the literature due to the extension of the model with object inheritance. The algebra defined deals with *homogeneous sets* of objects, and provides the model with efficient storage of persistent results of queries involving the *project* and *join* operators.

1. Introduction

Although *object-oriented database systems (OODBs)* have been researched for more than half a decade, there is still no formal definition of the *object-oriented database model*. Maier [Mai 89] argues that it is possible to precisely define what is an object-oriented database system, but it is not possible, for fundamental reasons, to define an object-oriented database model [Beer 89].

All the various database models (*relational, nested relational, complex objects, semantic, etc.*) separate the static properties of a database from the dynamic properties thereof, primarily by defining an essentially static database scheme, and by separately defining query and transaction languages. On the other hand an object-oriented database model provides a natural framework whereby dynamic properties of a database can be incorporated directly into a database scheme (in the form of *methods*). Thus the said model supports the notions of *data abstraction* and *encapsulation*. Moreover, the model supports the notion of *object identity* which provides the model with the concept of *object sharing* and thus avoids the update anomalies problem of *value-based* systems.

Another important concept of the object-oriented database model is the concept of *class inheritance*. This avoids redundant specification of data structures and operations within

the model. In this paper we extend this concept by introducing the novel concept of *object inheritance*.

In this paper we aim to formalize an extension of the object-oriented database model and thus provide the framework within which to define a query language for the said model. The rest of the paper is organised as follows. In Section 2 we introduce the object-oriented database model and we extend it via the concept of *object inheritance*. In Section 3 we formally define the extended object-oriented database model. In Section 4 we define our *object-oriented query algebra* for our extended object-oriented database model. Finally, concluding remarks and some research issues arising from our work are presented in Section 5.

2. The Object-Oriented Database Model

As it was mentioned in the introduction, there is no clear consensus on what an object-oriented database model is. There are still some arguments with regard to the basic principles and the main characteristics of the said model. Also, the model lacks a strong theoretical framework. In this section we introduce the main concepts of the object-oriented database model, emanating from object-oriented programming languages and some existing object-oriented database models, and in particular from the ORION object-oriented database model [Ban 87]. Moreover, we extend the said model with the concept of *object inheritance*.

2.1. Objects, attributes, and methods

In object-oriented database systems any entity is modelled by an *object*. Each object is associated with an *object identifier*. This identifier explicitly represents the object throughout all its life and is never re-used even when the object is deleted from the system. We discuss further the use of identifiers later in this section. The concept of an object also provides the model with the principles of *data abstraction* and *encapsulation* (data and operations are modelled together). Every object encapsulates a *state* and a *behaviour*. The *state* of an object consists of the values of the *attributes* of the object and the *behaviour* of the object consists of the set of *methods* which operate on the state of the object under consideration. The notion of attributes is similar to that of the relational model, but the *domain* of any attribute is not restricted to

atomic built-in values (integers, strings, etc). In the object-oriented database model the value of an attribute can also be an object (identifier of the object) as well as a set of values (set of built-in values or set of objects, etc.). Methods encapsulate the behaviour of an object. The idea here is not to operate on objects with general purpose operators (like the ones provided by a relational algebra, for instance), but rather to use *class* (discussed in the next subsection) specific operators (methods). The advantages emanating from this approach are:

- 1) the structure of objects is hidden away from the user who operates on objects only by methods;
- 2) the methods can be used to implement integrity checks that are specific to the object class under consideration;
- 3) operational independence is provided by the methods.

Thus an object-oriented database model provides more modelling capabilities on the operational side. Methods consist of code that manipulates or returns the state of an object. They are part of the definition of the *class* to which the object belongs. Methods, as well as attributes, are not visible from outside the object. Objects can communicate with one another through *messages*; messages constitute the public interface between objects. For each message understood by an object, there is a corresponding method that executes that message. An object reacts to a message by executing the corresponding method, and then returning a value.

2.2. Class, class hierarchy, and inheritance

All the objects which share the same set of attributes and methods are grouped together into a higher level object, called a *class*. An object must belong to only one class as an *instance* of that class. A class corresponds to a relational scheme in the relational model. All instances of a class respond to the same messages. A class describes the form (attributes) of its instances, and the operations (methods) applicable to instances thereof. Thus, when a message is sent to an instance, the method which implements that message is found within the definition of the class.

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a *class hierarchy* extends *information hiding* one step further. A class hierarchy is constructed by using the *is_a* relationship. *A is_a B* means that the class *A* is a *subclass* of the class *B* (*specialization*). *B* is a *superclass* of *A* (*generalization*). Any subclass has its own attributes and methods, but in addition it inherits the properties (attributes and methods) of its superclass(es). If, for example, class *A* is a subclass of class *B*, and *B* is a subclass of class *C*, then *A* inherits the properties of both *B* and *C*, since *B* inherits the properties of *C*. If we restrict our class hierarchy in the sense that a class can only have one *immediate* superclass, then we talk about *single inheritance* and we have a *class hierarchy*. If we allow a class to have two

or more immediate superclasses then we have *multiple inheritance* and we can represent it by a *class lattice*. To our knowledge all object-oriented programming languages and OODBs define a class *OBJECT* [Ban 87] as the root of the class hierarchy or class lattice. Such a system-defined class is a superclass of any other system-defined or user-defined class.

2.3. Object identity

Identity is that property of an object which distinguishes it from any other object. Identity has been investigated independently in general-purpose programming languages and database programming languages. Its importance has grown as these two environments evolve and merge [KhC 86]. Any real-world object is a unique object. So we need a notion of representing this uniqueness in a database model. In the relational model object identity has been represented by using user-defined *primary keys*. This *value-based* approach when implemented in relational database systems causes a lot of problems. One such problem is the update anomalies problem resulting from a change in the value of a primary key which is also a *foreign* key in another relation (*referential integrity*). Another problem is that objects with the same value are indistinguishable in the relational model. Furthermore, in any relation scheme each attribute or meaningful subset of attributes cannot have an identity. The reader is referred to [KhC 86] for further discussion on identity. In an object-oriented database model identity is dealt with by attaching a unique object identifier to each object. Thus the object has an existence which is independent of its value. Thus two objects with the same value can be two different objects. This approach to object identity avoids the problems mentioned above which occur in the relational model. In addition, it provides the model with the notion of *object sharing*. Any object in the database can be shared by other objects. This results in avoiding the storage of redundant data.

2.4. Object inheritance

Object inheritance extends the concept of *class inheritance* one step further. Consider, for example, an object-oriented database scheme where we have a class *person* and two subclasses of this class, namely *employee* and *student*. In conventional object-oriented databases when we create a new *employee* instance, a unique identifier is assigned to the instance value. This identifier is added to the set of *employee* instances of the database. However, since *employee* is a subclass of *person* any *employee* instance is also a *person* instance. Hence we add the aforementioned identifier to the *person* set of instances as well. So, basically the *employee* set of instances is a subset of the *person* set of instances. In our model, when a new instance of a class is created then the system automatically creates instances of all superclasses of the class (apart from *OBJECT*) to which the instance belongs. So in the particular example when we create an *employee* instance the system

automatically creates a *person* instance which is a *superobject* of the *employee* instance. This *person* object is *inherited* (included) in the *employee* instance. That is we have two objects created with two different object identifiers each belonging to the corresponding class set of instances. In classical object-oriented systems the new instance is a mapping from all the attributes of the state (explicitly or implicitly defined through inheritance) to appropriate values, whilst in our model this mapping is restricted to the explicitly defined attributes only; in addition, *superobjects* are created for each superclass of the class of the new instance, and the new instance is a *subobject* of these *superobjects*. Being a *subobject*, the new instance inherits all the values of the *superobjects*, and also all the methods that can be applied to them.

Object inheritance is thus concerned with inheritance of values. This approach necessitates the creation of extra object identifiers. The main reason for adopting this approach is to extend the notion of *object sharing*. For example, if a particular person is both an employee and a student, then we make use of the *person* object identifier which can be shared by both subclass instances, instead of duplicating unnecessarily the person information in each subclass instance. The said approach provides better semantics for our model. The two instances of *student* and *employee* include as a common *superobject* the *person* instance, thus referring to the same person. Hence our model provides a built-in version of *referential equality* [Mas 89]. In conventional object-oriented database systems this is not the case, since the two instances of *student* and *employee* have the same values for the *person* attributes but they do not specify that the person is the same. A solution to this problem would be to have a new class definition *employee-student* being a subclass of both the *employee* and *student* classes. This standard solution, however, clutters our class lattice with too many class definitions.

Our model supports more consistently the notion of *referential integrity*. One of the main advantages of the object-oriented database model is the existence of object identifiers which ensures that the *update anomalies* problem of *value-based* systems is avoided. Object inheritance incorporated into our model provides better semantics for referential integrity. For the said example when a particular *person* is both an *employee* and a *student*, and we want to update a particular *person* attribute, then in conventional object-oriented database systems we have to do so in both the *employee* and *student* instances, so as to keep the database consistent. In our model we only have to update the *person* instance.

Additionally our model provides better semantics for deletion operations. Generally deletion is not well defined within the context of object-oriented databases. If we delete a particular object there might be other objects referring to the identity of the deleted object, thus leading to *dangling* references. An implementation solution is to use *garbage collection*

mechanisms whereby an object is deleted and its storage space is restored to the system only when all references to this object are destroyed. So deletion semantics within the context of object-oriented databases is an implementation issue. Another problem that arises in deletion semantics is when we want to delete a class instance, but we wish to retain information on any *superobjects* of the class instance to be deleted. Referring again to our example, suppose that we want to delete a particular *employee* instance from our database. An explicit deletion of the instance results in its deletion from all its superclasses as well. This is obviously not always desirable, since, for our particular example, we would like to keep the personal information for the particular employee who is to be deleted. A *removal* operator in a garbage collection mechanism can remove a particular instance from a class whilst still keeping this information in the superclasses. However, this is not explicit deletion. Another way of dealing with this is to use *mutability* mechanisms if they are provided by the system. That is to *demote* the *employee* instance to a *person* instance. Our model provides abstract semantics for explicit delete operations, due primarily to object inheritance. Thus the deletion of the *employee* instance, does not affect the *person* instance since we have two distinct object-identifiers for the two instances under consideration. In this way we still have the personal information of the employee who is to be deleted.

Finally, another advantage of our model is that it provides a framework for defining precisely a closed object-oriented query algebra, which deals with *homogeneous* sets and provides precise and efficient definitions for the *project* and *join* operators when we wish to save their query results as persistent objects.

3. Formalization of the Object-Oriented Database Model

In this section we formalize the concepts introduced in Section 2 by giving formal definitions to the concepts of *class*, *type*, *value*, *object-oriented database scheme*, and *object-oriented database instance*, and by providing some primitive operators that manipulate the structure of classes.

3.1. Class

Definition 3.1.1. A *class* is defined as a triple $\langle \text{state}, \text{behaviour}, \text{inheritance} \rangle$.

When each class is defined, it is given a unique name and we use this name and the *dot* notation to access the three components of the triple. The *class lattice* is built indirectly by the triple definition and more specifically by the inheritance component.

Definition 3.1.2. The *state* of a class is a *tuple* type of the form $[A_1 : a_1, \dots, A_n : a_n]$, where A_i is an attribute name and a_i is the *type* of the *value* that the attribute can have in a particular instance of a class.

Types and values are discussed in the next subsection. According to their types, attributes are divided into *atomic* and *compound* attributes. An *atomic* attribute is an attribute of *atomic* type. A *compound* attribute is either an attribute of *class* or *tuple* or *set* type.

Definition 3.1.3. The *behaviour* of a class is a set of *methods*, explicitly defined for this class.

Definition 3.1.4. The *inheritance* of a class consists of a set of *immediate superclasses* of the class under consideration. The common superclass OBJECT is not included in this set if no immediate superclasses exist for the class.

3.2. Class instances, types, and values

3.2.1. Class instance

A *class instance* is a mapping from a class to a particular value. More precisely, it is a mapping from the state attributes to values in the domain of each attribute, and from the inheritance superclasses to corresponding class instances.

A class instance is associated with a unique identifier (*object identifier*). Hereinafter object and class instance will mean the same thing. Also, we remark that the mapping from superclasses to instances is actually a mapping from superclasses to object identifiers. More formally a class instance of a class C with immediate superclasses $\{C_1, C_2, \dots, C_k\}$ is defined as follows.

Definition 3.2.1. Given a class definition $C = \langle [A_1 : a_1, \dots, A_n : a_n], \text{methods}, [C_1, \dots, C_k] \rangle$, a *class instance* is an association of an *object identifier* with a tuple $\langle [v_1, \dots, v_n], \{i_1, \dots, i_k\} \rangle$, where $v_1, \dots, v_n$ are *values* for the attributes $A_1, \dots, A_n$, and $i_1, \dots, i_k$ are the object identifiers of the immediate *superobjects* of the class instance.

The two components of the tuple referred to in the above definition are known as *state* and *inheritance* of the object. They are accessed by using the *dot* notation. Furthermore, their individual components (attribute values and superobjects) are similarly accessed.

We now introduce the concept of *types* which allow the construction of the domains of attributes.

3.2.2. Types

Definition 3.2.2. A *type* is one of the following : *atomic type*, *class type*, *set type*, *tuple type*.

An *atomic type* is one of the following : *integer*, *string*, *boolean*. A *class type* is basically a type of a class definition. A *tuple type* is of the form $[A_1 : a_1, \dots, A_n : a_n]$, where A_i is an attribute name and a_i is the type of the value of the attribute. A *set type* restricts the elements of a particular set to a certain type.

3.2.3. Values

As it was mentioned earlier an object is a mapping from attributes and superclasses within a class definition to corresponding values. We now define the various kinds of values.

Definition 3.2.3. A *value* is one of the following : *atomic value*, *object value*, *tuple value*, *set value*.

An *atomic value* is any value over an atomic type, i.e. a particular integer, or a particular string, or a particular boolean. An *object value* is an object identifier (*o_id*) of a class instance. A *tuple value* is a mapping from the attributes of the tuple to corresponding values of the appropriate type. A *set value* is a set of values of a particular type.

One of the main advantages of the object-oriented database model is the notion of object identity. As we have seen in our model, only class instances are considered as objects, i.e. only class instances are associated with object identity. That is object sharing is provided only for class instances. The introduction of the tuple type and tuple value in our model was first addressed in [LeR 89]. The main reason for this is that sometimes values are not to be shared and are explicitly owned by a particular object [Ban 87, LeR 89]. Moreover, the introduction of the tuple type avoids the need of creating extra classes in our database scheme. Finally, the tuple type combined with the set type gives a more flexible database model that can also represent the *relational* and *nested relational* database models [LeR 89].

3.3. The object-oriented database model

Definition 3.3.1. An *object-oriented database scheme* is a set of class definitions.

Definition 3.3.2 A *class set* is a set of class instances.

Definition 3.3.3. An *object-oriented database instance* is a set of class sets.

3.4. Operations on classes

The inheritance concept incorporated into Definition 3.1.1 of class avoids the explicit specification of additional attributes and methods. Instead it provides indirectly these additional properties. We provide some primitive operators that operate on the structure of a class.

3.4.1. The *all_superclasses* operator

In the inheritance component of the class we only consider the immediate superclasses of the class under consideration. We therefore need an operator to return all superclasses of a specific class (propagate upwards in the class lattice). Hence we define the following operator.

Definition 3.4.1. The *all_superclasses* operator is given by

$$\text{all_superclasses}(C:\text{class}) = (C_1(\text{all_superclasses}(C_1)), \dots, C_n(\text{all_superclasses}(C_n))),$$

where n is the cardinality of $C.\text{inheritance}$ (denoted by $|C.\text{inheritance}|$), and $\forall i: 1 \leq i \leq n : C_i \in C.\text{inheritance}$.

More specifically this operator returns a portion of the class lattice, from the given class C to the class OBJECT. We also assume a primitive operator *flatten* which converts such a portion of the class lattice to the corresponding *set* of superclasses.

3.4.2 The all_attributes operator

This operator returns all the attributes of a given class (directly or indirectly defined due to inheritance) at the top level.

Definition 3.4.2.1. The *all_attributes* operator is given by

$$\text{all_attributes}(C:\text{class}) = \text{Att}(C) \cup \left(\bigcup_{i=1}^{|C.\text{inheritance}|} \text{all_attributes}(C_i) \right),$$

where $\text{Att}(C)$ is the set of attributes in $C.\text{state}$, and $\forall i: 1 \leq i \leq |C.\text{inheritance}| : C_i \in C.\text{inheritance}$.

We refer to the attributes $\text{Att}(C)$ as the set of *explicitly* defined attributes, whereas we refer to all other attributes as *implicitly* defined attributes.

Besides the *all_attributes* operator, we also need an operator to navigate within the structure of the class and to return any *nested attribute*. We use the *dot* notation to navigate through nested attributes. The operator to be defined will return not only the top level attributes, but also all the nested attributes. The set of all nested attributes is referred in [Wed 89] as the *set of property induced functions*. Any attribute in this set corresponds to an *induced property function* [Wed 89].

We define the length, $\text{len}(A)$, of an attribute A to be the number of *dots* occurring in an attribute. We also assume that the operator $\text{type}(A)$ returns the type of the said attribute. Finally, we assume that the operators $\text{atomic_attr}(B)$ and $\text{compound_attr}(B)$, where B is a *tuple* type, return the corresponding sets of atomic and compound attributes in B .

Given a type B the *all_level_attr*(B) operator returns all attributes of B at all levels.

Definition 3.4.2.2. Given a type B , the *all_level_attr*(B) operator is given by

$$\text{all_level_attr}(B) =$$

$$\text{atomic}(B) \text{ then } \{ \}$$

$$\text{tuple}(B) \text{ then } \text{atomic_attr}(B) \cup \text{compound_attr}(B) \cup \left(\bigcup_{i=1}^{|\text{compound_attr}(B)|} \{ B_i, B'_j \mid B'_j \in \text{all_level_attr}(\text{type}(B_i)) \} \right)$$

$$\text{class}(B) \text{ then } \left(\bigcup_{i=1}^{|B.\text{inheritance}|} \text{all_level_attr}(B_i) \right) \cup \text{all_level_attr}(B.\text{state})$$

$$\text{set}(B) \text{ then let } A \text{ be the type of the set attribute, then } \{ A'_j \mid A'_j \in \text{all_level_attr}(A) \}.$$

The above operator may not terminate for a particular *class* type or *tuple* type or *set* type when the object-oriented database scheme under consideration is *cyclic* [Wed 89]. To avoid getting into infinite loops we modify this operator by adding a control parameter d , the *depth*, that restricts to d the maximum nested level of the returned attributes. Thus we redefine the operator to *all_level_attr*(B, d), which *unnests* our object-oriented database scheme to a maximum depth of d .

3.4.3. The all_methods operator

This operator returns all the methods of a given class (directly defined or indirectly defined due to inheritance).

Definition 3.4.3. The *all_methods* operator is given by

$$\text{all_methods}(C:\text{class}) = C.\text{behaviour} \cup \left(\bigcup_{i=1}^{|C.\text{inheritance}|} \text{all_methods}(C_i) \right)$$

where $\forall i: 1 \leq i \leq |C.\text{inheritance}| : C_i \in C.\text{inheritance}$.

Correspondingly we refer to all methods in $C.\text{behaviour}$ as *explicitly* defined and to all the other methods due to inheritance as *implicitly* defined.

3.5. The UNFOLD operator

We next provide a restructuring operator to "UNNEST" the structure of a value to atomic values. This operator is an extension of the UNFOLD operator encountered in the FAD database programming language [BBKV 87].

Definition 3.5. The *UNFOLD* operator is given by

$$\text{UNFOLD}(\text{atomic value}) = \text{atomic value}$$

$$\text{UNFOLD}(o_id) = \left(\bigcup_{i=1}^n \text{append UNFOLD}(o_id_i) \right) \text{append} (\text{UNFOLD}(o_id.\text{state}))$$

$$\text{UNFOLD}([A_1 : \text{value}_1, \dots, A_k : \text{value}_k]) = [A_1 : \text{UNFOLD}(\text{value}_1), \dots, A_k : \text{UNFOLD}(\text{value}_k)]$$

$$\text{UNFOLD}(\{\text{value}_1, \dots, \text{value}_k\}) = \{\text{UNFOLD}(\text{value}_1), \dots, \text{UNFOLD}(\text{value}_k)\},$$

where $o_id.\text{state}$ is a tuple value, n is the number of immediate superobjects of o_id , o_id_i is a superobject of o_id and *append* is the usual LISP operator.

4. The Object-Oriented Query Algebra

The development of object-oriented database systems can be divided into two approaches. One approach is to extend a *computationally complete object-oriented programming language* to incorporate database concepts. The other is to

extend a database model incorporating concepts from object-oriented programming languages. In either approach a *query language* can be incorporated into the system.

The main advantages of query languages that support our decision to define such a language for our model are :

1. Query languages are easy to formulate.
2. Optimization strategies can be applied to these languages.
3. Programs in general (database) programming languages do not terminate in general, and it is undecidable whether a program will do so : programs are not *safe*. On the other hand, queries formulated in a query language always generate a result in a finite number of steps.
4. Most programming languages are *navigational* whereas most query languages are *descriptive*.

Some attempts have been made to define query algebras for the object-oriented database model [Kim 89, ScS 89, ShZ 89]. The main problems to be solved when designing such an algebra are :

1. The structure of the resulting class of the query.
2. The location of the resulting class in the class lattice.

The main properties we require from a query language for the object-oriented database model is *closure* and *safety*. A language is *closed* under an underlying structure if the input and output structures in any query are the same. For example, the relational algebra is *closed* under the *relation* structure. The *object-oriented query algebra* defined herein is *closed* under the *class* structure. Also our algebra is *safe* (i.e. all queries terminate). This is guaranteed by the definition of the operators.

The main contribution of our algebra is the definition of the *project* and *join* operators which are unique in the literature in the sense that the new resulting class is appropriately positioned in the class lattice. Also the introduction of our *object inheritance* concept in combination with the above operators minimizes data redundancy when we want to save query results as persistent objects. Finally, our algebra deals with *homogeneous* class sets instead of with *heterogeneous* ones like all the other query algebras, at least to our knowledge. This obviously enhances the semantics of our model, since any member of any class set is known to be of a particular class type, and not of any other subclass type.

A final remark on query algebras for the object-oriented database model is that the definition of such a language is antinomic with the concept of *data abstraction* and *encapsulation* since the language operates directly on the structure of the objects thus violating these two concepts.

4.1. The set operators

In all object-oriented query algebras known, the set operators *Union*, *Intersection* and *Difference* are defined to be identical to Codd's [Cod 70] corresponding relational operators. The difference is that object equality is defined over the object identifiers of objects. Also in the relational model the two operand relations in a relational operation are required to be over the same relation scheme. In the object-oriented database model there is no such restriction. *Heterogeneous* class sets are allowed to be operands and/or results of a set operator, with a particular common supertype defining the class set type. In our model the introduction of *object inheritance* allows us to deal with *homogeneous* class sets. This is advantageous since any member of a set is of a specific class type. In *heterogeneous* sets a member is of a particular class type (the common supertype of the class set) but it may also be of any other subclass type.

In order to illustrate the difference between the conventional object-oriented set operators and those of our model, let us consider our *person*, *student* and *employee* database scheme discussed in the previous sections. Suppose that we have a set of *employee* class instances and a set of *student* class instances, and we wish to take the *union* of the two sets. In conventional object-oriented database systems this is the set union of the two sets. The result is a *heterogeneous* set of *employee* and *student* object identifiers. The common class type of the resulting set is the *person* class. In our model the resulting set is a *homogeneous* set of *person* object identifiers. In order to get the result we first convert the *employee* and *student* class sets to *person* class sets by extracting from each *employee* and *student* instance the corresponding *person* superobject. This is done by using the *extract* operator to be defined next. Finally, we perform the union of the resulting sets.

4.1.1. The extract operator

Definition 4.1.1. Let c be a class set over class C , and A a superclass of the class C . The *extract* operator is given by

$$\text{extract}(A, c) = \{ \text{pick}(A, i) \mid i \in c \},$$

where $\text{pick}(A, i)$ returns the *superobject* of type A in the class instance i , by navigating in the class structure and by using the *dot* notation for accessing.

4.1.2. The set operators

Definition 4.1.2. Let $*$ denote a set operator, c_1 a class set over class C_1 , c_2 a class set over class C_2 and A the *least* common superclass of C_1 and C_2 . Then

$$c_1 * c_2 = \text{extract}(A, c_1) * \text{extract}(A, c_2).$$

The use of object identifiers in the model restricts the definition of the set operators to *minimal* definitions [LeL 89]; in *minimal* operators the objects are considered as indivisible

units. Hence we cannot have the advantages of *maximal algebras* [RKS 89].

4.2. The project operator

The *project* operator defined herein is different from any other one found so far in the literature. It is different from that in [Kim 89] in the sense that we adopt a version of the first of the two options discussed in that paper. Our definition can also be seen as an extension of the project operator found in [ScS 89]. We prefer not to have the projected class scheme as a direct subclass of the root of the class lattice. Instead we return a new class definition which inherits attributes from as many superclasses as possible, retains the methods of all these superclasses, and finally has a state with all the remaining attributes in the list of attributes to be projected that cannot be inherited. Also the methods that can be defined over the said remaining attributes are retained. Thus the user can only project a set of attributes (NOT methods). The methods that are available to the projected class are those implicitly defined in all the superclasses of the projected class together with those that can be defined on the said remaining attributes. This option is adopted to avoid extra storage requirements and takes advantage of the concept of object inheritance introduced in this paper. As a final remark we note that the definition of the project operator violates the concept of encapsulation since the user distinguishes between attributes and methods, and thus knows the structure of the class.

Prior to giving the formal definition of the project operator, we give an example to demonstrate how our operator works.

4.2.1. Example

Consider the following class lattice:

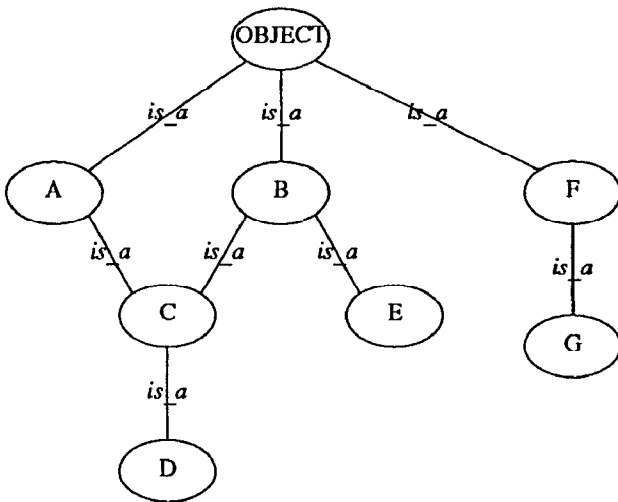


Figure 4.2

where OBJECT is the common superclass as defined in all object-oriented database systems, and A, B, C, D, E, F, G are

class definitions with corresponding state attributes $Att(A)=\{A_1, \dots, A_n\}$, $Att(B)=\{B_1, \dots, B_m\}$, $Att(C)=\{C_1, \dots, C_p\}$, $Att(D)=\{D_1, \dots, D_k\}$, $Att(E)=\{E_1, \dots, E_r\}$, $Att(F)=\{F_1, \dots, F_u\}$ and $Att(G)=\{G_1, \dots, G_s\}$.

In our notation the above structure is defined as follows:

```

class A = < [A1 : a1, ..., An : an], mA, { } >
class B = < [B1 : b1, ..., Bm : bm], mB, { } >
class C = < [C1 : c1, ..., Cp : cp], mC, {A, B} >
class D = < [D1 : d1, ..., Dk : dk], mD, {C} >
class E = < [E1 : e1, ..., Er : er], mE, {B} >
class F = < [F1 : f1, ..., Fu : fu], mF, { } >
class G = < [G1 : g1, ..., Gs : gs], mG, {F} > .
  
```

Now suppose that we have a class set of class instances of type D, and we want to perform a projection on this class set as a result of a simple query involving only the project operator. We would like to compute the scheme of the result of the query. Let X be the set of attributes to be projected in the query, and let D' be the resulting projected class. In [Kim 89] the projected class D' is a new class, which explicitly contains all the attributes in X and is made an immediate subclass of the common superclass OBJECT. In [ScS 89] a more elaborate approach is adopted by considering whether the new class can be an immediate superclass of D. If not then the new class is made an immediate subclass of OBJECT.

Herein we adopt a more general solution than that in [ScS 89] by taking into consideration the whole structure of the class lattice as it is implicitly defined in the inheritance component of Definition 3.1.1.

In our example, suppose that $X = \{A_1, \dots, A_n, B_1, \dots, B_m, C_1, D_2\}$. Following both the above approaches [Kim 89, ScS 89] the resulting projected class, D', has a state with attributes $Att(D') = X$ and is an immediate subclass of the superclass OBJECT. In our approach we construct a new class, which is an immediate subclass of both A and B, since all attributes of A and B are included in the projected set of attributes, and we explicitly define only the remaining attributes. More formally the resulting projected class is of the form :

$$D' = \langle [C_1 : c_1, D_2 : d_2], m_{D'}, \{A, B\} \rangle,$$

where $m_{D'}$ is the set of methods that are defined over C_1 and/or D_2 and any other attributes implicitly defined through inheritance.

Thus in any projected instance of the class set, we do not store explicitly the attribute values corresponding to the attributes in X, but only those values of the attributes explicitly defined in the state of the projected class. The object identifiers corresponding to the immediate superclasses defined in the inheritance component are retained.

We now give an algorithm that, given a class definition, C, and a set of attributes to be projected, X, computes the projected class C'. More specifically the algorithm computes the inheritance component of the projected class.

4.2.2. An algorithm for the project operator

Let C be a class definition and X a set of attributes to be projected. The projected class is of the form $C' = \langle \text{new_attributes}, \text{new_methods}, \text{new_inheritance} \rangle$, where new_inheritance is the set of immediate superclasses of the projected class, and is computed by the *projected_inheritance* function given hereafter; new_attributes is the set of the said remaining attributes that cannot be inherited by superclasses, and new_methods is the set of methods that can be defined over the attributes new_attributes and any implicitly defined attributes from the superclasses.

Let $m = \max \{ \text{len}(X_i) \mid X_i \in X \}$ and

let $X' = X \cup \left(\bigcup_{i=1}^{|X|} \{ X_i, Z \mid Z \in \text{all_level_attr}(\text{type}(X_i), m) \text{ and } X_i \in X \} \right)$.

projected_inheritance(C, X') : returns *new_inheritance*

begin

let $C.\text{inheritance} = \{C_1, \dots, C_k, C_{k+1}, \dots, C_n\}$

where $(\forall i : 1 \leq i \leq k \quad \exists A : A \in (\text{all_level_attr}(C_i, m) - X') \text{ and } (\text{atomic}(\text{type}(A)) \text{ or } \text{len}(A)=m))$

and $(\forall i : k+1 \leq i \leq n \quad \exists A : A \in (\text{all_level_attr}(C_i, m) - X') \text{ and } (\text{atomic}(\text{type}(A)) \text{ or } \text{len}(A)=m))$

$\text{new_inheritance} := \{C_1, \dots, C_k\}$;

$\text{temp_inheritance} := \bigcup_{i=k+1}^n \text{projected_inheritance}(C_i, X')$;

$R := \text{temp_inheritance} - \{ S \mid S \in \text{temp_inheritance} \text{ and } \exists C_j \in \text{new_inheritance} : S \in \text{flatten}(\text{all_superclasses}(C_j)) \}$;

$\text{new_inheritance} := \text{new_inheritance} \cup R$;

return *new_inheritance*

end (*projected_inheritance*).

$\text{new_attributes} = X - \bigcup_{i=1}^n \text{all_attributes}(C_i)$,

where n is the cardinality of *new_inheritance* and $C_i \in \text{new_inheritance}$.

Before defining the project operator we first define the *restriction* of a class instance i to a set of attributes X denoted by $i[X]$.

Definition 4.2.1. Given a class definition C and the corresponding projected class on the set of attributes X , $C' = \langle A'_1 : a'_1, \dots, A'_r : a'_r \rangle$, new_methods , $\{C'_1, \dots, C'_j\}$, the *restriction* of a class instance i of C to the attributes in X is given by

$$i[X] = \langle [v'_1, \dots, v'_r], \{i'_1, \dots, i'_j\} \rangle,$$

where $\forall k : 1 \leq k \leq r$, v'_k is the value of the class instance i on attribute A'_k (A'_k is either explicitly or implicitly defined within the class C), and $\forall g : 1 \leq g \leq j$, i'_g is the object identifier of the superobject of i corresponding to the appropriate class C'_g in the projected class inheritance. More formally $\forall g : 1 \leq g \leq j$, i'_g

$= \text{pick}(C'_g, i)$.

Definition 4.2.2. Let c be a class set over class C and let X be the set of attributes to be projected. Then the *projection* of c on X is given by

$$\pi_X(c) = \{ i[X] \mid i \in c \},$$

where $i[X]$ is assigned to a unique newly created object identifier.

4.3. The join operator

The *join* operator has not yet been fully researched within the area of object-oriented query algebras. This operator as defined in existing query algebras is incomplete. We also note that in all existing such algebras the join operator is actually a *theta join* operator, i.e. the user specifies the *joined* condition formulae. In [Kim 89] the join operator returns a new class with two attributes being the two operand classes in the join. This new resulting class is made a direct subclass of the superclass OBJECT. This definition of the join operator does not preserve the required associativity property. In [ShZ 89] the type of the result of the join operator is a *parameterized tuple*, with two components, namely the two operand classes (*abstract data types*), with a newly created object identifier assigned to this tuple. Hence a join between two tuples, an n -tuple and an m -tuple, results in an $(n+m)$ -tuple; similarly a join between an n -tuple and another class results in an $(n+1)$ -tuple. Thus the algebra defined in [ShZ 89] is not *closed* under the class definition. The main motivation for incorporating this tuple approach into the algebra is to obtain algebraic optimization properties, and more specifically associativity of the join operator. In [ScS 89] the join operator is defined so as to return a new class type, which basically has two attributes, one attribute of the type of the class of the first operand in the join operator, and the other attribute of the type of the class of the second operand. Thus their algebra preserves the *closure* property since the result of the join operator is a class type, and associativity of the join is obtained by using the *extend* operator defined within their algebra.

As far as we know there is no other different definition of the join operator apart from those mentioned above. We claim that both the above definitions are not clear and precise, and also have some disadvantages. Neither of these definitions positions the resulting class in the appropriate position in the class lattice. In our algebra the introduction of *object inheritance* requires that the *join* operator apart from the *theta join* should also be partially a *natural join*. If, for instance, we want to join two classes which have a common superclass, then apart from the *user-specified condition formulae* we must also satisfy the requirement that for any two instances to be joined the common *superobject* instance should be the same. Considering again our *person*, *employee*, *student* database scheme, suppose that we want to find all persons that are both employees and

students, and who study and work at the same city (assuming that we have as an employee attribute the city where the employee works, and similarly as a student attribute the city where the student studies). This query can be expressed as a *theta* join, with the user-specified joined condition being that the college city is the same as the job city. In addition, our join operator performs a *natural* join over the superobject *person* since the above user-specified condition by itself is not sufficient to give the correct result. Hereafter we define the *natural join* operator of our model. The definition preserves the required associativity property of the operator, which is essential for query optimization.

4.3.1. The natural joined class

Definition 4.3.1. Given two class sets, c_1 of class type C_1 , and c_2 of class type C_2 , the *natural joined class*, C' , is the class type of the result class set $c' = c_1 \bowtie c_2$. The joined class C' is computed as follows:

- (i) $C' = C_1 = C_2$
- (ii) $C' = C_1$ if $C_2 \in \text{flatten}(\text{all_superclasses}(C_1))$.
- (iii) $C' = C_2$ if $C_1 \in \text{flatten}(\text{all_superclasses}(C_2))$.
- (iv) $C' = \langle C_1.\text{state} \text{ append } C_2.\text{state},$
 $C_1.\text{behaviour} \cup C_2.\text{behaviour},$
 $\{ C \mid (C \in C_1.\text{inheritance and}$
 $(\forall C'' : C'' \in C_2.\text{inheritance}$
 $C \notin \text{flatten}(\text{all_superclasses}(C'')))) \text{ or}$
 $(C \in C_2.\text{inheritance and}$
 $(\forall C'' : C'' \in C_1.\text{inheritance}$
 $C \notin \text{flatten}(\text{all_superclasses}(C'')))) \} \rangle$

Definition 4.3.2. The *natural join* operator for the above four cases is defined by

- (i) $c_1 \bowtie c_2 = \{ i \mid i_1 \in c_1 \text{ and } i_2 \in c_2 \text{ and } i = i_1 = i_2 \}$
- (ii) $c_1 \bowtie c_2 = \{ i_1 \mid i_1 \in c_1 \text{ and } i_2 \in c_2 \text{ and } \text{pick}(C_2, i_1) = i_2 \}$
- (iii) $c_1 \bowtie c_2 = \{ i_2 \mid i_1 \in c_1 \text{ and } i_2 \in c_2 \text{ and } \text{pick}(C_1, i_2) = i_1 \}$
- (iv) $c_1 \bowtie c_2 = \{ i \mid$
 $i_1 \in c_1 \text{ and } i_2 \in c_2 \text{ and}$
 $\forall j: 1 \leq j \leq |C'.\text{inheritance}| \text{ and } C_j \in C'.\text{inheritance} :$
 $(a) C_j \in C_1.\text{inheritance and } C_j \in C_2.\text{inheritance}$
 $\text{ then } i.\text{inheritance}.C_j = \text{pick}(C_j, i_1) = \text{pick}(C_j, i_2)$
 $(b) C_j \in C_1.\text{inheritance and } C_j \notin C_2.\text{inheritance}$
 $\text{ then let } X = \{ C_k \mid C_k \in \text{flatten}(\text{all_superclasses}(C_2))$
 $\text{ and } C_k \in \text{flatten}(\text{all_superclasses}(C_j)) \}$
 $\text{ and } \forall k: 1 \leq k \leq |X|: \text{pick}(C_k, i_1) = \text{pick}(C_k, i_2)$
 $\text{ then } i.\text{inheritance}.C_j = \text{pick}(C_j, i_1)$
 $(c) C_j \in C_2.\text{inheritance and } C_j \notin C_1.\text{inheritance}$
 $\text{ then let } X = \{ C_k \mid C_k \in \text{flatten}(\text{all_superclasses}(C_1))$
 $\text{ and } C_k \in \text{flatten}(\text{all_superclasses}(C_j)) \}$
 $\text{ and } \forall k: 1 \leq k \leq |X|: \text{pick}(C_k, i_2) = \text{pick}(C_k, i_1)$
 $\text{ then } i.\text{inheritance}.C_j = \text{pick}(C_j, i_2) \}.$

We now give an example to illustrate how our join operator works at the scheme level, showing the positioning of the resulting class in the class lattice.

Example 4.3.2

Consider Figure 4.2 and suppose that we have a class set d , over the class D , and a class set e over the class E , and we want to compute the join $d \bowtie e$. A join in any of the other algebras discussed previously would only rely on the condition formulae provided by the user and *would ignore the fact that the class B is a common superclass of both D and E* . Also the resulting class would be an immediate subclass of the class OBJECT. The resulting class set would basically be a set of newly created objects with new identifiers. Each object would be a pair consisting of the two old identifiers from the two instances of the two class sets. Our join definition ensures that apart from the user-specified formulae to be satisfied, any two instances to be joined should also have the same common superobject over the class B . Finally, our resulting class, J , is appropriately positioned in the class lattice as shown in Figure 4.3.2.

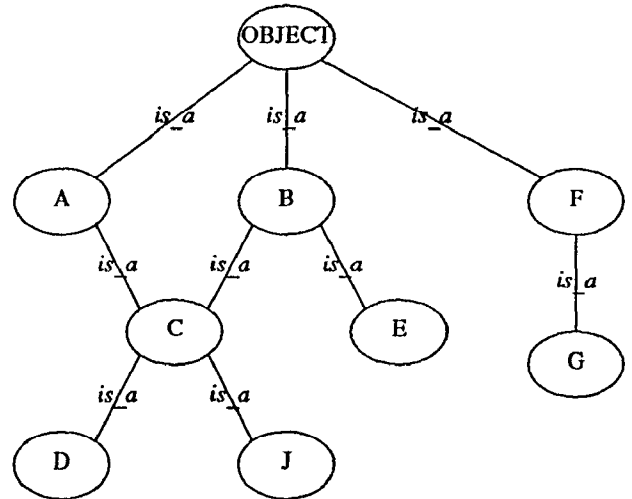


Figure 4.3.2

Using our notation the resulting class J is written:
 $J = \langle D.\text{state} \text{ append } E.\text{state}, D.\text{behaviour} \cup E.\text{behaviour}, \{ C \} \rangle.$

Finally, we leave as an exercise for the reader to compute the join of $(d \bowtie e) \bowtie g$, where g is a class set over G , and to verify that this is equivalent to $d \bowtie (e \bowtie g)$, i.e. our join operator is associative.

Conclusions

In this paper we have proposed an extension of the object-oriented database model. The extension incorporates the concept of *object inheritance*. This results in an increase of the number of object identifiers within the database. However, this

ensures that the model provides better semantics, incorporating *referential equality* and keeping the database more consistent in terms of updates and deletions. It also provides the framework for defining a *closed* and *safe* object-oriented query algebra. The algebra defined herein is more complicated than any of the other algebras proposed to date. This is due to *object inheritance* but also to the fact that our *project* and *join* operators are precisely defined so as to return the new resulting class in the appropriate position in the class lattice. This results in a more efficient representation of query results. Finally, we believe that our model and more particularly the *object inheritance* concept would probably be useful in researching the *mutability* of objects in object-oriented database systems.

References

- [Ban 87] Banerjee, J., Chou, H-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., Kim, H-J. "Data Model Issues for Object-Oriented Applications". *ACM Trans. on Office Information Systems*, Vol. 5, No 1, pp. 3-26, 1987.
- [Ban 88] Banerjee, J., Kim, W., Kim, H-J. "Queries in Object-Oriented Databases". *Proc. 4th Int'l. Conf. on Data Engineering*, pp. 31-38, Los Angeles, California, 1988.
- [Banc 88] Bancilhon, F., "Object-Oriented Database Systems". *Proc. of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 152-162, Austin, Texas, 1988.
- [BBKV 87] Bancilhon, F., Briggs, T., Khoshafian, S. N., Valduriez, P. "FAD, a Powerful and Simple Database Language". *Proc. 13th Int'l Conf. on Very Large Data Bases*, pp. 97-105, Brighton, England, 1987.
- [Beer 89] Beeri, C. "Formal Models for Object Oriented Databases". *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, pp. 370-395, Kyoto, Japan, 1989.
- [Cod 70] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks". *Comm. ACM*, Vol. 13, No. 6, pp. 377-387, 1970.
- [GoR 83] Goldberg, A., Robson, D. "SMALLTALK-80 : The Language and its Implementation". Addison-Wesley, 1983.
- [KhC 86] Khoshafian, S. N., Copeland, G. P. "Object Identity". *Proc. 1st Int'l Conf. Object Oriented Programming Systems and Languages*, pp. 406-416, Portland, Oregon, 1986.
- [Kim 89] Kim, W. "A model of Queries for Object-Oriented Databases". *Proc. 15th Int'l. Conf. on Very Large Data Bases*, pp. 423-432, Amsterdam, 1989.
- [LeL 89] Levene, M., Loizou, G. "The Null Extended Nested Relational Algebra". *To appear*.
- [LeR 89] Lecluse, C., Richard, P. "Modelling Complex Structures in Object-Oriented Databases". *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 360-367, Philadelphia, Pennsylvania, 1989.
- [Mai 89] Maier, D. "Why Isn't There an Object-Oriented Data Model?" *Proc. IFIP 11th World Computer Conference*, pp. 793-798, San Francisco, 1989.
- [Mas 89] Masunaga, Y. "Object Identity, Equality and Relational Concept". *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, pp. 170-187, Kyoto, Japan, 1989.
- [RKS 89] Roth, M. A., Korth, H. F., Silberschatz, A. "Null Values in Nested Relational Databases". *Acta Informatica*, Vol. 26, pp. 615-642, 1989.
- [ScS 89] Scholl, M. H., Scheck, H. J. "A Relational Object Model". *To appear*.
- [ShZ 89a] Shaw, G. M., Zdonik, S. B. "An Object-Oriented Query Algebra". *Data Engineering*, Vol. 12, No. 3, pp. 29-36, 1989.
- [Ull 87] Ullman, J. D. "Database Theory: Past and Future". *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 1-10, San Diego, 1987.
- [Wed 89] Weddell, G. E. "A Theory of Functional Dependencies for Object-Oriented Data Models". *Proc. 1st Int'l Conf on Deductive and Object-Oriented Databases*, pp. 150-169, Kyoto, Japan, 1989.