Expanding the Notion of Operations in an Object-Oriented Database

Jorng-Tzong Horng* and Baw-Jhiune Liu**

*Department of Computer Science & Information Engineering, National Taiwan University
Taipei, Taiwan, R.O.C.

**Department of Electrical Engineering, National Central University, Chungli, Taiwan, R.O.C.

ABSTRACT

Research in databases, particularly in complex related, large scale, data intensive applications, poses particular challenges to specify the structure and the behavior of real-world entities. In this paper we address the problem of providing a variety of operators to specify the structure and the behavior of objects in object-oriented databases. We present the taxonomy of operators. Operators are classified into four categories: extension operators, generalization hierarchy operators, aggregation hierarchy operators and reference operators. The variety and quantity of operators which are needed pose challenges especially for definitions, functions and usage. We classify and discuss these requirements with illustrations of examples.

This work was partially supported by the National Science Council under grant 800408E15502.

1. Introduction

Relational database systems[6,7,8] have been successful in commercial business applications. However they are subject to the limitations of a finite set of data types and the need of data normalization. Thus, prevent them to be used in the area that are complex related, large scale, data intensive applications such as CAD, AI, and office automation[16,19]. Since real world objects usually have to be decomposed into different relations in relational database systems. Few of one to one correspondence between real world entities and database elements exists. This makes the database an unnatural model of real world. Further, the relationship of the tuples of the various relations is achieved via user-generated attribute values. In other words, the relationships among tables are not explicitly expressed in the schema level, instead are implicitly represented via user-generated attribute values. This makes it difficult to interpret the database by database users, because it requires an intrinsic knowledge of the underlying schema definition.

The richer expressive capability of a data model will simplify the work of database design and give a natural model of real world. Object-orientation has gained wide popularity in computer science as a concept to capture much more semantics of software systems and their applications. In recent years, there are some proposals for systems which merge the object-oriented programming language concepts and semantic data modelling capabilities[16] to form an object-oriented environment supported by a database. Examples of this trend include several systems:

(1).Gemstone [5,15], which incorporated class modulation and type hierarchy concepts from Smalltalk language[13] to support a set-theoretic data model in an object-oriented programming environment.

(2).Iris [10], which is based on a semantic data model[16] that supports abstract data types[13]. Iris contains three important constructs: objects, types and functions. Properties of objects, relationships among objects, and computations on objects are expressed in terms of functions.

(3).ORION [1,2,3], which incorporated the concept of composite objects as an enhancement to the standard Smalltalk-like object-oriented data model. A composite object is a collection of related instances that form a hierarchical structure that captures the is_part_of relationship between an object and its parent.

A semantic hierarchy schema is composed of virtual classes[4,20] which are structured by the two abstraction mechanisms: generalization and aggregation[19]. This semantic hierarchy schema forms a virtual database[17] about the objects of an application and the constraints on those objects and their relationships. With this approach, the system is able to provide different user groups with their own subschema of the total conceptual schema. Besides, the system allows different user groups to have different perspectives of some aspects of the conceptual schema.
An important goal of our work is to define operators in object-oriented databases. In addition, another important goal will also be achieved by utilizing these operators to construct user-oriented views of objects as a semantic hierarchy schema.

1.1 Related Works

Related works may be divided into two parts: one is relative to object-oriented database operators and the other is relative to the concepts of view. We first introduce the works of object-oriented database operators. In Iris database[10], database is a collection of abstract data types[13], with operations defined on the types, and with objects which are instances of the types. Objects can be accessed and manipulated only by invoking operations defined on their types. All operations in this database are functions. Properties of objects and relationships between objects are expressed in terms of functions, which are defined over their types. An operation which accepts parameters of one type will also accept parameters which belong to subtypes of this type. This work emphasizes some aspects of operations in an object-oriented database. Particularly, how to specify and store operations in the database of view. We first introduce the works of object-oriented database operators. In Iris database[10], database is a collection of abstract data types[13], with operations defined on the types, and with objects which are instances of the types. Objects can be accessed and manipulated only by invoking operations defined on their types. All operations in this database are functions. Properties of objects and relationships between objects are expressed in terms of functions, which are defined over their types. An operation which accepts parameters of one type will also accept parameters which belong to subtypes of this type. This work emphasizes some aspects of operations in an object-oriented database. Particularly, how to specify and store operations in the database of view. We first introduce the works of object-oriented database operators. In Iris database[10], database is a collection of abstract data types[13], with operations defined on the types, and with objects which are instances of the types. Objects can be accessed and manipulated only by invoking operations defined on their types. All operations in this database are functions. Properties of objects and relationships between objects are expressed in terms of functions, which are defined over their types. An operation which accepts parameters of one type will also accept parameters which belong to subtypes of this type. This work emphasizes some aspects of operations in an object-oriented database. Particularly, how to specify and store operations in the database of view. 

In the following we describe the works that are relative to the concepts of view. Intuitively, it is natural to extend the relational view and accommodates it to the object-oriented database system. With this basic idea, the notion of "schema virtualization"[20] had been issued. In that, a virtual class is defined by associating a conceptual or virtual class with a predicate, and the is-a relationships between classes are declared by the virtual schema designer. In a specification, a virtual class is a set of objects subject to a predicate, induces the number of attributes of a virtual class to be unfixed. Thus, leads the way to contradict with the definition of a class in the conceptual database. The introduction of is-a relationship provides users to specify their own virtual schema, however it does not consider the attribute inheritance aspects of a virtual schema. Related to the work of schema virtualization, it is also worth noting the following work. In superview[14], Motro provides an integration language to construct user views that access multiple databases. The database model is based on the functional approach. It allows users to integrate data with the same concept on different databases. The emphasis is largely on database integration, but less on restricting or exposing the data that user is favorite. Since the underlying data model is a semantic data model, some of the integration language are suited for our work.

1.2 Organization of the Paper

Section 2 is a survey of object-oriented data modelling concepts. The data model provides the context to define operators in object-oriented database systems. In section 3, operators are defined by carefully exposing the relationships among the basic components of the data model. These operators shield the complexity of programming in deriving views of the database and serve as a specification of derivate views. Section 4 gives a conclusion.

2. Object-Oriented Data Modelling Concepts

In this section, the modelling power of object-oriented data models is described. This section is the fundamental theory we base on, in order to introduce the view concept in object-oriented database systems. There is an informal definition of an object-oriented database system. It is based on a data model that allows to represent one real-world entity (whatever its complexity and structure) by exactly one object (in terms of the data model concepts) in a database. Thus no artificial decomposition into simpler concepts is necessary unless the database designer decides to do so[9]. Generally speaking, an object-oriented data model is a kind of semantic data models having much abundance of modeling power and implementation aspects.

In an object-oriented data model, there are four components: objects, object identities, classes, and relationships. We described them separately as follows:

(1) Object

An object is a real-world element or concept which can be distinctly identified. In object-oriented systems, an object is an encapsulated abstract data type, and as such its attributes need not be single values, but may be other entities of arbitrary complexity. This allows us to create an one-to-one mapping between objects and the entities we are trying to model.

The behavior of an object is encapsulated in methods. Methods consist of code that manipulates or returns the state of an object. Methods are analogous to procedures and functions, and represent the external interface to objects. So, in these systems, attributes and methods completely define the semantics of objects.
(2) Object identity

An object has an existence and an identity which is independent of its value. Identity is that property of an object which distinguishes each object from all others [12]. Once it has an object-id, it can be referenced by other object regardless of its any change. Objects are different from their identity, they are all distinct and they might not have an external reference, such as key, that stands for them.

Object identity allows objects to be shared, and associations among entities can be modeled by relating the corresponding objects and not external references, such as user-defined attributes. When objects are updated, their modification is reflected in all other objects in which they appear as components.

(3) Class

Objects are classified into different classes in terms of their properties. All objects belong to the same class are described by the same set of attributes and methods. Objects are said to be instances of their classes. Classes are also used for object creation and for determination whether a request to apply a particular method to particular objects is legal. In object-oriented systems, classes provide a natural basis for modularization because they are commonly used to model entities in the application domain.

(4) Relationships

There are two important type of relationships: generalization and aggregation [17]. All relationships existing in database construct the 2-D hierarchies: generalization hierarchy and aggregation hierarchy.

The generalization hierarchy permits the inheritance of properties between different entities in the system. A generalization hierarchy is a hierarchy of classes in which an edge between a pair of classes represents the is-a relationship; that is, every element in the lower level class is also an element of the higher level class. For a pair of classes on a generalization hierarchy, the higher level class is called a superclass, and the lower level class is called a subclass. The attributes and methods specified for a class are inherited by its subclasses. Further, multiple inheritance is allowed by having a class be a subclass of more than one class, thus inheriting the properties of all its parents.

Aggregation is a user-defined relationship. An edge in the aggregation hierarchy represents the concept of "participate-in", "an_attribute_of", or "a_part_of". A certain relationship among objects may be abstracted as an aggregate object. In this aggregation, details about the participating objects are ignored and the relationship is named as a whole.

These two abstraction mechanisms: generalization and aggregation permit the user to model and view the data on many levels, and is consistent with the way people modeling the real world.

In summary, In an object-oriented database objects are organized into classes, which in turn are organized in class hierarchies. Hierarchies of classes express the semantics of "is-a", "is_instance_of", and "has_part_of" between classes or objects. Classes are therefore logically organized by these explicitly stated semantic relationships rather than being a collection of independent data types.

3.2 A Taxonomy of Operators

Object-oriented database operators, like relational algebra, can be defined for the manipulation of data. But in the semantic manipulation, the former provide more operational support than the latter. This because of the variety of objects, their relationships, and more sophisticated data modeling concepts such as aggregation and generalization are provided by object-oriented data models. In this context, the spirit behind defining "operator" is to expose the two basic semantic concepts in object-oriented data models, i.e. object identity and abstraction hierarchy.

Operators are defined by carefully exposing the relationships among the basic components of the object-oriented data model. The basic constructs of an object-oriented data model are objects, object identities, classes, and relationships. The functionalities of operators include that restrict instances of a class, change the structure of hierarchies, construct the is-a and part-of relationships between classes, change attributes of a class, move class references along the generalization structure, and reorder the attributes of a class. The user's view is constructed by issuing a sequence of operators over the conceptual schema. These operators not only are used as an interface to the system but also serve as a specification of derive views and will shield much tedious programming efforts in deriving views from a database.

Operators are divided into the following 4 categories:

(A). extension operators which create a virtual class by extracting instances from classes.

(B). generalization hierarchy operators which create sub/super virtual class by exposing the is-a relationship implied or declared in the conceptual schema.

(C). aggregation hierarchy operators which scope the attributes a virtual class has.

(D). reference operators which specify where the attributes or participants of a class come from.

In general, database schema consists of database extension domain and database intension domain. The former is the database itself. In the specification, it focuses on restricting or selecting instances from specific classes. Operators in (A) and (D) are this kind of operator. In the latter, problems appear to change the definition of the schema, or modify
the structure of abstraction hierarchies. In the specification, it place emphasis on reordering or hiding the attributes of objects, or changing the structure of abstraction hierarchies. Operators in (B) and (C) are used for these purposes. Since the data schema is changed after restructuring the intension domain of a database, the variation of the extension domain cannot be forbidden.

We describe notations used in the following sections as follows:

attribute(aClass): all attributes of aClass.

all_inst(aClass): all instances of aClass.

In the following examples, we assume a database (Fig.1) which contains four classes[14]: PERSON, STUDENT, ADVISOR and ASSISTANT. In that, PERSON is a generalization of STUDENT and ADVISOR, STUDENT and ADVISOR form a multiple inheritance hierarchy. The instances and the corresponding attributes of each class are listed below:

attribute(PERSON)=pid, age, sex, faculty
(* the attributes of the PERSON class*)
all_inst(PERSON)=01--07
(* all instances that belong to the PERSON class*)
attribute(STUDENT)=pid, age, sex, faculty, s#, sname, degree
all_inst(STUDENT)=03--05
attribute(ADVISOR)=pid, age, sex, faculty, a#, aname
all_inst(ADVISOR)=05--07
attribute(ASSISTANT)=pid, age, sex, faculty, s#, sname, degree, a#, aname
all_inst(ASSISTANT)=05

3.2.1 Extension Operators

These operators apply on the domain of instances of a class. The operation for the manipulation of instances of a class is provided below:

(a) C select: all_inst

The definition of this operator consists of the specification of a class. This operation returns a virtual class whose instances belong to the given class C. The instances and the corresponding attributes of the virtual class (if named V) are listed as follows:

attribute(V): the same as C.
all_inst(V): the same as all_inst(C).

The result of the following example:

PERSON select: all_inst
returns a class whose instances belong to the PERSON class.

(b) C select: direct_inst

The definition of this operator consists of the specification of a class. This operation returns an unnamed virtual class. The instances and the corresponding attributes of the virtual class (if named V) are listed as follows:

attribute(V): the same as C.
all_inst(V): all_inst(C) - all_inst(subclasses of C).

Where "-" represents the difference operator.

The following example returns a class whose instances belong to class PERSON but do not belong to class STUDENT and ADVISOR:

PERSON select: direct_inst.
Similarly, the following example returns a class whose instances belong to class STUDENT but do not belong to class ASSISTANT:

STUDENT select: direct_inst.

(c) C select: aQualification

The definition of this operator consists of the specification of a class and a predicate. The result of this operation returns a virtual class whose instances are satisfied the aQualification. The following example returns a class whose instances are person and their age are less than 30:

PERSON select: [:x1 x age < 30].

In a similar manner, the following example returns a class whose instances belong to class STUDENT but do not belong to class ASSISTANT:

STUDENT select: [:x1 x not in ASSISTANT].

The instances and the corresponding attributes of the virtual class (if named V) are listed as follows:

attribute(V): the same as STUDENT.
all_inst(V): all_inst(STUDENT) - all_inst(ASSISTANT).
Another example is illustrated as follows:

\[
\text{PERSON select: } \{x:x \text{ in ASSISTANT}\}.
\]

The example returns a class whose instances play two roles: student and assistant. The instances and the corresponding attributes of the virtual class (if named \(V\)) are listed as follows:

- attribute(\(V\)): the same as STUDENT.
- all_inst(\(V\)) \(\cap\) intersect( all_inst(STUDENT), all_inst(ASSISTANT)).

Note that operator (c) is more general than operators (a) and (b). Note also set operators such as intersect, difference are simulated individually by "in" and "not in" operators in the aQualification expression.

(c) rename A to B

This operator renames class A to B.

3.2.2 Generalization Hierarchy Operators

Since generalization hierarchy is a fundamental conceptual structure in object-oriented data models. It is important to identify basic operations for restructuring a generalization schema such that the same information contained in the generalization schema can be represented differently. Hence, redundancy is inhibited and data integrity is preserved.

In the aspects of semantic manipulation, this kind of operator is more important and meaningful than relational algebra. This because of an important mechanism-abstraction is supported in object-oriented data models. By utilizing these operators, objects in a generalization schema can have different levels of perspective, that is, objects can be perceived by a user at different levels of detail at different time.

The operation for restructuring schema is provided as follows:

(a) partition aClass into (classes) \(\{\text{with discard}\}\) as select: Qualification

This operation creates a variety of virtual classes by partitioning aClass according to some attributes. The optional expression "with discard" means that the attribute specified in the Qualification does not appear in the virtual classes. An example of this operation is given as follows:

\[
\text{partition PERSON into (Young,Old) as select: } \{x:x \text{ age}<30,x \text{ age}>30\}.
\]

This operation returns two virtual classes: Young and Old. The instances and the corresponding attributes of each virtual class are listed as follows:

- attribute(Young)=pid,age,sex,faculty,rank
- all_inst(Young)= (young person)
- attribute(Old)=pid,age,sex,faculty,rank
- all_inst(Old)= (old person)

Where rank is a catalog attribute whose value may be the names of some subclasses of the given class. It indicates one of them to which the instances belongs. In this example, the value of the rank may be "STUDENT", "ADVISOR", or both depending on the class to which the instance belongs.

Another example of using this operation is shown below:

\[
\text{partition Young into (Youngman,Youngfemale) with discard as select: } \{x:x \text{ sex='man', x sex='female'}\}.
\]

The attributes and instances of each class are listed below separately:

- attribute(Youngman)=pid,age,faculty,rank
  all_inst(Youngman)= (young man)
- attribute(Youngfemale)=pid,age,faculty,rank
  all_inst(Youngfemale)= (young female)

Due to the 'with discard' is specified, the 'sex' attribute in the virtual classes is discarded.

(b) Gen (subclasses) into Superclass

This operation creates a virtual class, Superclass, which is a generalization of the specified subclasses and creates is-a relationships between the Superclass and each one of the subclasses. The operator is useful in creating a superclass from a set of given subclasses. The instances and the corresponding attributes of the Superclass are listed as follows:

- attribute(Superclass)=intersect (attribute(subclasses))
- all_inst(Superclass)=union (all_inst(subclasses)).

Consider the following definition:

\[
\text{attribute(STUDENT)=pid,age,sex,faculty,s#,sname,degree}
\text{all_inst(STUDENT)=03--05}
\text{attribute(ADVISOR)=pid,age,sex,faculty,c},
\text{all_inst(ADVISOR)=05--07}.
\]

Example of Gen

\[
\text{Gen (STUDENT,ADVISOR) into PERSON}
\text{attribute(PERSON)=pid,age,sex,faculty}
\text{all_inst(PERSON)=03--07}.
\]

(c) Object-join (superclasses) into Subclass

Entities in real-world may have different roles in different positions. For example, a person may be an employee by day and a student by night. In object-oriented databases, this situation can be easy modeled by declaring some objects in different classes having the same object identities. The Object-join operator can achieve this function. Moreover, if the conceptual model does not support the multiple inheritance function, the operator will serve this work.

This operation creates a virtual class, Subclass, which is a specialization of superclasses, and creates is-a relationships between
each one of the superclasses and the Subclass. Usually this operation is 
used to create multiple inheritance hierarchies. The instances and the 
corresponding attributes of Subclass are listed as follows:

attribute(Subclass): union attribute(supercIasses)
all_inst(Subclass): intersect all_inst(superclasses).

Consider the definition of classes STUDENT and ADVISOR in (b) 
again. Example of Object-join

Object-join (STUDENT, ADVISOR) into ASSISTANT 
attribute(ASSISTANT)=pid,age,sex,faculty, #.sname,degree, #.aname, 
all inst(ASSISTANT)= {instances | those are both in STUDENT and in 
ADVISOR}.

(d) merge (classes) into C 
The operation merges the given classes into a virtual class C. The 
merge operator causes multiple similarly structured classes to become 
one. The precondition of this operation is all classes which must have 
the same attributes. The attributes and instances of the class C is shown 
below:

attribute(C): the same as (classes).
all_inst(C): union all_inst(classes).

(e) subtyping A to B 
The subtyping operation creates the is-a relationship between 
subclass A and superclass B. The operation must satisfy the following 
precondition:

Precondition: every instance of A must also exist in B.

(f) specialize Superclass into (subclasses) (with discard) as select: Qualification

Given class Superclass and predicate Qualification, the operation 
first partitions the Superclass into some subclasses according to the 
Qualification expression and then generalizes these subclasses to the 
Superclass. The specialize operator is a composite operator, which can 
be implemented by partition and Gen operators.

The above operators are defined for specifying the generalization 
structure of objects. By invoking different sequence of operations, we 
can obtain different generalization schemata.

In Fig.1, in general PERSON is divided into STUDENT, 
ADVISOR, and ASSISTANT. If we are interested in the faculty of a 
person. We can partition PERSON (according to the attribute "faculty") 
into CSfaculty,EEfaculty,and Linguistic. The example is shown below:

1. partition PERSON into (CSfaculty,EEfaculty,Linguistic) with 
discard as select: [x.1 x faculty='CS', x faculty='EE', x 
Faculty='Linguistic'),

2. Gen (CSfaculty,EEfaculty) into Engineer,
3. Gen (engineer,Linguistic) into PERSON.

Apply operations (1),(2) and(3) we can get a generalization virtual 
schema (Fig. 2) from Fig. 1.

Fig. 2 A Generalization View of the Conceptual Database

In a similar manner, if we are interested in the age of a person and 
the sex of a person. We can partition PERSON (according to the 
attributes "sex" and "age") into Young, Old, Youngman, and 
Youngfemale. We could have operations as follows:

1. partition PERSON into (Young,Old) as select: [x:1 x age<30, x 
age>=30],
2. specialize Young into (Youngman,Youngfemale) with discard as 
select: [x:1 x sex='man', x sex='female'],
3. Gen (Young,Old) into PERSON.

Apply operations (1),(2) and (3) we get another generalization virtual 
schema (Fig. 3) from Fig. 1.

Fig. 3 Another Generalization View of the Conceptual Database
From the above examples, we acquire different virtual schemata from the same source of conceptual schema by applying a sequence of operators to Fig. 1.

3.2.3 Aggregation Hierarchy Operators

In object-oriented databases, real-world entities can be modelled in terms of two basic constructs: objects and relations. Objects correspond to real-world entities which can be regarded as individual for modeling purpose. Relations are a formalization of relevant real-world relationships in which the entities participate. Aggregation abstraction allows a relationship between named objects to be thought as a higher-level named object, that is, the relationship can be abstracted as a single object. We distinguish two kinds of aggregation: attribute aggregation and relation aggregation. Attribute aggregation is the activity to associate attributes in a class template. Relation aggregation is the activity to expose relationships among objects. The former provides the ability to model composite objects, and the latter represents the relationship among objects as an abstract object.

The operators for the manipulation of aggregation is defined as follows:

(a). Typing $S(a_1, \ldots, a_m)$ into $C$

This operation constructs an aggregation hierarchy. The steps are described below:

(i) The attributes $a_1, \ldots, a_m$ of $S$ are moved from class $S$ to class $C$, that is, create an attribute aggregation $C(a_1, \ldots, a_m)$.

(ii) Declare the part-of relationship between class $S$ and class $C$.

The instances and the corresponding attributes of the virtual class are listed as follows:

attribute($C$): $a_1, \ldots, a_m$.

all-inst($C$): {instances of those derived objects that come from $S$ class}.$C(a_1, \ldots, a_m)$.

attribute($S$): $a_1, \ldots, a_m$.

all-inst($S$): no change.

Where $C$ is an attribute whose value is some instance of the class $C$.

As an example, consider the class

$\text{THESIS}(s\#, sname, degree, a\#, aname, title)$

Apply the following two operations

1. Typing $\text{THESIS}(s\#, sname, degree)$ into $\text{STUDENT}$.
2. Typing $\text{THESIS}(a\#, aname)$ into $\text{ADVISOR}$.

We get an aggregation virtual schema as follows:

(b). expand $S(A)$

This operation appends the attributes of class $A$ to class $S$, and then deletes the relationship between class $A$ and class $S$. In other words, this operation creates an attribute aggregation.

The instances and the corresponding attributes of the resulted $S$ are listed as follows:

attribute($S$): (attribute($S$) union attribute($A$)),

all-inst($S$): the same as all-inst($S$).

Consider the schema of Fig. 4, the following operations

1. expand $\text{THESIS}($STUDENT$)$
2. expand $\text{thesis}($ADVISOR$)$

will change the class $\text{THESIS}(title)$ into $\text{THESIS}(s\#, sname, degree, a\#, aname, title)$.

3.2.4 Reference Operators

In a generalization hierarchy, an object may have many levels of abstraction, so objects will associate different number of attributes when references are done from different abstraction levels of the generalization plane. This motivates us to define some operators for going up/down to the desired abstraction level. In the following description, changing the scope of reference means changing the reference of objects on different levels of abstraction.
(a). dot operator: "." 

The dot operator specifies an aggregation component which may be an object or a class depending on the context. As an example 

THESIS.STUDENT 

denotes the students who participate in the thesis.

(b). super_ref 

The operation changes the scope of reference from subclass to superclass.

(c). sub_ref 

The operation changes the scope of reference from superclass to subclass.

As an example, in Fig.4 classes THESIS, STUDENT and ADVISOR form an aggregation hierarchy. If we want to create a virtual schema that contains virtual classes phd_thesis, phd_student and phd_advisor. We can use the following operations (1)–(6) to create an aggregation hierarchy (Fig. 5). The definition comes from Fig. 1 and Fig. 4.

1. phd_student <= STUDENT select: [:x1 x degree="phd"],
2. subtyping phd_student to STUDENT,
3. phd_thesis <= THESIS select: [:x1 x.STUDENT sub_ref
phd_student],
4. phd_advisor <= phd_thesis'.ADVISOR select:all_inst,
5. subtyping phd_advisor to ADVISOR,
6. phd_thesis <= phd_thesis'. select: [:x1 x.ADVISOR sub_ref
phd_advisor]

In the next example we show the effect of using sub/super reference when expanding a composite object. Expand operator will be used if all the participant's attributes are required. Since objects in an generalization hierarchy have different levels of view. Change the reference from a participant/component object to its sub/super class will change the number of attributes being expanded.

Consider a schema include 4 classes: Part, Price, NT$, US$. The definition of these classes is given below:

Part(p#, price) is a class has attributes: p#, price.
Price is a composite object whose class is Price.
Price has attributes: unit, cost.
Cost is an attribute whose class is Cost.
Cost has two subtypes: NT$ and US$.
nt$ and us$ are the individual attribute of NT$ and US$.
The following operations generate class Part which has attributes p#, unit and nt$.

(a) Part.nt$-charge <= Part select: [:x1 x.price.cost sub_ref
NT$).
(b) expand part.nt$-charge(price).
(* show part in NT$*).

In a similar manner, the following operations generate class Part which has attributes p#, unit and us$.

(a) Part.us$-charge <= Part select: [:x1 x.price.cost sub_ref
US$).
(b) expand part.us$-charge(price).
(* show part in US$*).

![Fig. 5 A 2-D Schema](image)

4. Conclusions

On levels of abstraction, a given real-world can be modeled as a set of objects of various classes and relationships among them. In order to precisely specify the structure and behavior of objects, some primitive operators are defined. In addition, these operators provide more versatile viewpoints of real-world entities for users. The operator, which we define to construct a virtual schema, exposes the semantics between data explicitly at the schema level. This will ease the control and management of user's favorite data. The operator in this respect serves as a low level definition language in the construction of user's view. Like the relational algebra, the operator will be the central components when a high level query language is developed or processed in an object-oriented database system.

Constructing the semantic virtual database (user view) as a 2-D hierarchy structure, we obtain several advantages. These advantages are described as follows. First, in user's perspective a semantic virtual
database has the same architecture as the underlying conceptual database, so the same query language will work. Second, in a semantic virtual database, the basic construct - virtual class provides another way to see user-specific objects avoiding data redundancies. Hence, the process of updating a database is simplified and data integrity can be enhanced. Finally, user's data are modelled as a 2-D hierarchy of virtual schema with levels of abstraction. This provides users with a wholly concise view of a virtual database.

Other expansion of these operations are under consideration: for example, expand schema from two dimension to three dimension and parallel processing for these operations.

References


