TR-90-002

View Definition in an Object Oriented Database

TR-90-002

View Definition in an Object Oriented Database

Te-Son Kuo, J. T. Horng and B. H. Liu

TECHNICAL REPORT

Institute of Information Science

Academia Sinica

Taipei, Taiwan, Republic of China

July 1989

Abstract

In this thesis we address the problem of providing different ways to see objects in an object-oriented database system. The thesis first issues the notions of "derived object","virtual object", and "virtual schema", then provides a mechanism to construct user-specific views that simply and naturally extends the underlying object-oriented data model. User view is constructed by operators defined in this thesis, which are powerful enough to capture the essential feature of data abstraction and inheritance in the underlying data model. The method of mapping from the view definition to the conceptual schema is presented by examples.

Key Words and Phrases: object-oriented model, semantic data model, view, derived object.

Contents

Chapter 1. Introduction **1-1**

1.1 Introduction 1-1

1.2 Related Works 1-7

1.3 Organization of the Thesis 1-8

Chapter 2 Object-Oriented Data Model **2-1**

2.1 The (Pure) Relational Database System 2-1

2.2 Object-Oriented Programming 2-4

2.3. Object-Oriented Data Model Concepts 2-8

Chapter 3. Forming Virtual Classes And Views **3-1**

3.1 Definitions 3-1

3.2 Operator Approach 3-2

3.3 Taxonomy of Operators 3-3

3.3.1 Extension Operators 3-7

3.3.2 Generalization Hierarchy Operators 3-10

3.3.3 Aggregation Hierarchy Operators 3-19

3.3.4 Reference Operators 3-26

Chapter 4. Mapping of views **4-1**

4.1 Query Graph and Translation Method 4-1

4.2 Translation Retrieval Queries 4-5

4.3 Discussion 4-9

Chapter 5. Conclusion 5-1

Reference R-1

Chapter 1. Introduction

1. 1 Introduction

Relational systems have been successful in commercial business applications. However they are subject to the limitations of a finite set of data types and the need to normalize data. Thus, prevent them to be used in the area that are complex related, large scale, data intensive applications such as CAD, Office Automation. Since real world objects usually have to be decomposed onto 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 difficultly to interpreter the database by the database user, because it requires an intrinsic knowledge of the underlying schema definition.

The richer expressive capability of a data model will ease the work of database design and gives a natural model of real world. Object-orientation is getting ever popular in computer science as a concept to better capture the semantics of software systems and their applications. In recently years, there are some proposals for systems which merge the object-oriented programming language concepts and semantic data modeling capabilities to form an object-oriented environment supported by a database. Examples of this trend include several systems: (1).Gemstone [CM84], which incorporated class modulation and type hierarchy concepts from Smalltalk language[GR83] to support a set-theoretic data model in an object-oriented programming environment. (2).Iris [Fish87], which is based on a semantic data model that supports abstract data types. 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 [BCGK87], 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.

There are several important characteristics offered by object-oriented database systems, when comparing it with record-oriented database systems. First, information about "schema" and "data" is modeled using the single concept: object. Thus, management about meta-data and database itself is all the same. Second, object-oriented database systems support the direct representation of two essential data abstractions, namely aggregation and generalization. This two abstraction mechanism together with classification, which assembles the same concept of objects into classes, provides a natural,modular framework for modeling the basic structure of those applications like CAD/CAM or office automation. And third, object-oriented database systems model an object as an abstract data type with identity,. This allows the ability of easy sharing of objects and provides a mechanism to encapsulate data and updating operations. Finally, object-oriented database systems open a potential door to allow various kinds of changes to the conceptual database structure. This function is especially desirable for those applications whose environments are irregular,unpredictable,and evolving [LM88].

This research investigates an approach to provide user-oriented views of objects as a semantic hierarchy schema. A semantic hierarchy schema is composed of virtual classes which then are structured by the two abstraction mechanism: generalization and aggregation. This semantic hierarchy schema forms a virtual database about the objects of an application and the constraints on those objects and their relationships. With this approach, the system is able to:

(1) provide different user groups with their own subschema of the total conceptual schema,

(2) allow different user groups to have different perspectives of some aspects of the conceptual schema.

Another important goal of our work is that we expect to find the semantic operations of an object-oriented data model, that is, operations that manipulate the semantic components of a data model. This goal deeply affects the approach we adopt to define the user view.

Although several object-oriented data models have been proposed, to our knowledge, a few of papers cover the topic of views in the object-oriented data model. This motivates us to study the topic of this thesis in depth.

Now, we give a brief review of the object-oriented data model that we base on to design the view mechanism. The data model mainly borrows from the paper: "Object-oriented Database Approach to Multimedia Applications" [WKL86]. Objects in a multimedia database have various properties and participate in a number of relationships with other objects. The semantic data model concepts: attribute aggregation(an object contains other objects) and relation aggregation (an object is related to another object) are adopted to solve this complicated situation. The essential concepts of Smalltalk-80 language including object identity, abstract data type, generalization hierarchy among objects, and property inheritance are all important concepts in this model. The two abstraction mechanism: generalization and aggregation permit the user to model and view the data on many levels, which is consistent with the way people modeling the real world. The detailed description of object-oriented data model concepts is presented in Chapter 2.

Views have been formally treated in relational database. A view is defined in a relational model as a query over the base relations, and perhaps also over other views. In other words, view is implemented virtually, that is, view as a table has attributes but no data populates it. A mapping is available when transform user operations on views into operations over the base data. The final result is obtained by interpreting the combination of view definition and user query, using the description of the database stored in the database schema.

Following the idea of relational view when considering the problem we address. Some new research issues arise:

(a) Since view is a subset scope of the conceptual database. Objects in view should be treated as they are in the conceptual database. In other words, the concept of object identity must be preserved in the virtual database.

(b) For the same reason, classes defined in the virtual database should have the concepts of attributes, is_a relationship, and part_of relationship. Hence, the rule to define mappings between the conceptual database and the virtual database are needed.

(c) Since the schema in an object-oriented database is structured (based on is_a and part_of relationships). Rules to define the structured schema of the virtual database are needed.

1.2 Related Works

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"[TYI88] 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. The 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 also worth noting the following works:

(1). In Superview[Motro87], 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 restriction 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. (2).In the ORION project[BKK88] schema evolution was investigated, that is, the ability to dynamically make a wide variety of changes to the database schema. ORION provides a variety of operators to change a database schema and a set of rules to construct a valid database schema. However, operators in this paper do not concern the reconstruction of the data schema, nor support to define multiple schema virtually. But the taxonomy of over 20 schema changes gives the hint of the taxonomy of our operators.

1.3 Organization of the Thesis

Chapter 2 is a survey of object-oriented data model concepts. The data model provides the context to discuss the view facility in the object-oriented database system. In Chapter

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. In Chapter 4 , mapping of views is discussed. Chapter 5 gives a conclusion.

Chapter 2. Object-Oriented Data Model

In this chapter we first discuss the encountered problems in using the relational model to capture the semantics of applications that are complicatedly related, and highly structured. In the second section, the essential features of object-oriented programming are reviewed and advantages to use object-oriented approach are listed. At the last section, the modeling power of object-oriented data models is thoroughly discussed. This section is the fundamental theory we base on in order to introduce the view concepts in object-oriented models.

2.1 The (Pure) Relational Database System

In the commercial world, relational database systems have recently become popular for data processing applications. Relational model is more flexible and easier to use than previous database models. It is more flexible because inter-record relationships do not have to be pre-defined. The relational join operator allows a user to relate records dynamically based on attribute values. The relational model is easier to use because of its more intuitive metaphor of tabular data and because of fourth-generation query languages, which allows a user to pose ad hoc retrieval requests to the system. However, these DBMs are subject to the limitations of a finite set of data types and the need to normalize data. Thus, prevent it to be used in the area that are complex related, large scale, data intensive applications such like CAD, and Office Automation. Example to show the unnatural modeling behavior of the relational model is the following.

Example[KW87]:

real world objects usually have to be decomposed onto different relations (subject to simple data type and normalization) in the relational database model. Let us illustrate this on a relational BR(bracket-boundary representation) schema that is described below.

Mech_Part

ID	
cuboid	
cuboid	
pyramid	

FACES

ID	EDGES
f1	e1
f1	e2
...	...

EDGES

ID	VERTICES
e1	v1
e1	v2
e2	v1

VERTICES

ID	X	Y	Z
v1	0	0	0
v2	...	...	...
v3			

Table 2.1 Bracket Tables

We notice that this representation is broken up into four different relations, where the relationship of the tuples of the various relations is achieved via user-generated attribute values. This makes the model difficult to use by the database user, that is ,the engineer, in order to retrieve and manipulate the data because it requires an intrinsic knowledge of the underlying schema definition. In order to retrieve all the bounding vertices of the mechanical part "cuboid", one could formulate the following SQL queries:

```
select Mech_Part.ID,X,Y,Z
from Mech_Part,FACES,EDGES,VERTICES
where Mech_Part.FACES = FACES.ID
and FACES.EDGES = EDGES.ID
and EDGES.VERTICES = VERTICES.ID
and Mech_Part.ID = "cuboid".
```

These queries involve Joining the four relations Mech_Part, FACES, EDGES, and VERTICES. Adequately supporting such frequent join operations seems to be the major issue in extending the (pure) relational model.

2.2 Object-oriented Programming

Object-oriented languages have been available for many years. The productivity increases achievable through the use of such languages are well recognized. Although the object-oriented paradigm has many different uses and therefore many different definitions, the following aspects are recognized as being essential:

(1) Data abstraction and encapsulation

Every object comes with a set of operations which are used to operate upon and to change the object. Moreover, the object consist of a public interface and a private implementation part. This provide a way to reduce interdependence between software components. thus, modifiability and reliability is enforced.

(2) Object identity

The object identity is independent of (mutable) values of properties which makes a representation into a real-world entity. Once it has an object-id it can be referenced by it regardless any change.

(3) Messages

Contrary to high_level programming languages, objects communicate by passing message. Each message consists of a receiver object identity, the particular message name and the arguments for the message.

(4) Property inheritance

Objects are organized into classes, which in turn are organized in class hierarchies. The way of economy specification is achieved by giving a single description of the generic object properties and by automatic inheritance of properties defined at a lower level of abstraction.

(5) Overloading

Operators (function and procedure) can be overloaded. Both the operator name, the argument types, and the class of the receiver in the type hierarchy determines the specific operator definition.

(6) Late binding

Moving the binding of variables to runtime improves the expressiveness at the cost of loosing compile-time error checking capability.

(7) Graphics

The dialogue with modern graphic workstations require loose control over the input sequence. An object-oriented approach is best suited to model this manipulation of individual objects presented on a screen.

Since class hierarchy is a natural form to model the application domain. Object-oriented programming differs from conventional control-based programming by encouraging you to concentrate on the data to be manipulated rather than on the code that does the manipulation.

Object-oriented languages have many advantages over more traditional procedure-oriented languages. We list it below.

(a). Data abstraction is a way of using information hiding, which increase reliability and help decouple procedural and representation specification from implementation.

(b). Inheritance permits code to be reused. This has attendant advantage of reducing overall code bulk and increasing programmer productivity.

(c). Software economy is achieved. Programs are developed first for general cases, by defining methods for general classes, then these methods are refined by adding new subclasses only when strictly necessary.

2.3. Object-Oriented Data Model Concepts

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) of the database. Thus no artificial decomposition into simpler concepts is necessary unless the database designer decides to do so [Ditt87]. Generally speaking, an object-oriented data model is a kind of semantic data model having specific attractive points on modeling power and implementation aspects.

In the object-oriented data model, there are four components: Objects, Object Identities, Classes, and Relationships. We discuss it detailedly as follows:

(1). Object

An object is a real-world element or concept which can be distinctly identified. A person identified by his name (B. W. Liu) and a course identified by title (Compiler) are examples of objects. In object-oriented systems, an object is an encapsulated abstract data type, and as such its attributes need not be single values, but can be other entities of arbitrary complexity. This allows us to create a 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 manipulate or return 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 (collectedly called properties) 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 [KC86]. Once it has an object-id it can be referenced by it regardless 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.

The main concept of 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 others objects in which they appears as components.

(3) Class

Objects are classified into different classes in terms of their properties. All objects belonging 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.

An example of a class definition is given in table 2.2. This template defines a class whose name is Card. An instance of class Card can be used to represent a card in a game program.

Class Card has instance variables named suit,rank,and faceup. A new instance is created by sending class Cared the creation message suit:rank:. For example

```
| aCard |
aCard<- Card suit: 'heart' rank:7.
```

creates a new instance of Card that represents the seven of hearts. In the method for suit:rank:, the message new creates an uninitialized instance. The internal message setSuit:setRank: sets the suit and rank fields and initializes the new instance to be "face down". Given an instance of class Card, we can determine its suit,rank,and orientation, and change the latter using the external messages specified in the class definition. Because we do not want to be able to change the suit and rank of an instance once it has been created, we do not include a message for doing this operation in the set of external messages.

class name Card

superclass Object

instance variable names suit rank faceup

class messages and methods

```
    suit: aString rank: anInteger | aCard |
    aCard<- self new.
    aCard setSuit: aString setRank: anInteger.
```

instance messages and methods

external

```
suit | | † suit.
rank | | † rank.
turnFaceup | | faceUp<- true.
turnFacedown | | faceUp <- false.
turnOver | | faceUp <- faceUp not.
isFaceUp | | † faceUp.
isFaceDown | | † faceUp not.
```

internal

```
setSuit: aString setRank: anInteger | |
    suit <- aString. rank <- anInteger. self turnFaceDown.
```

Table 2.2 Class template for class Card.

(4) Relationships

There are two important type of relationships: generalization and aggregation. 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". For instance, a certain relationship between a person,a hotel,a room,and a date may be abstracted as the aggregate object "reservation". Thus, the name "reservation" may be used to identify the relationship, and hide all underlying semantic.

These two abstraction mechanism: 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.

As an example[SMF86], consider how a "computer" might be represented using aggregation and generalization. A computer is made up of several functional units (e.g. CPU, Storage, I/O). Thus, a computer is an aggregation of these sub-systems. Similarly, each sub-system is an aggregation of other units. The storage system, for instance, might be composed of a data register, error correction logic and a data array. Fig. 2.1 shows the aggregation hierarchy for computer.

we can also think of the term computer as a generalization of several types of machines. For example, an IBM 4341, a VAX11/780 and an MV10000 are all computers. Each of these computers, in tern, may be a generalization for several models in a family (the IBM4341 family includes the IBM4341-2, the IBM4341-11 and the IBM4341-12). Fig. 2.2 shows the generalization hierarchy for computer.

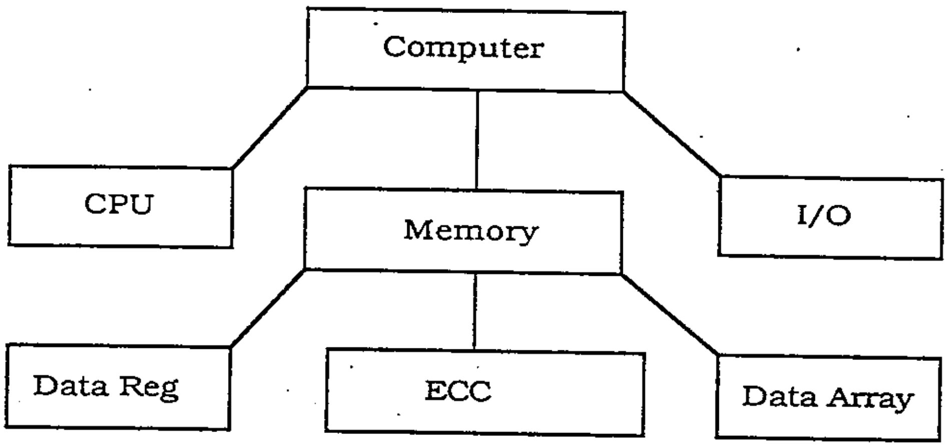

Fig 2.1 Aggregation Hierarchy for Computer

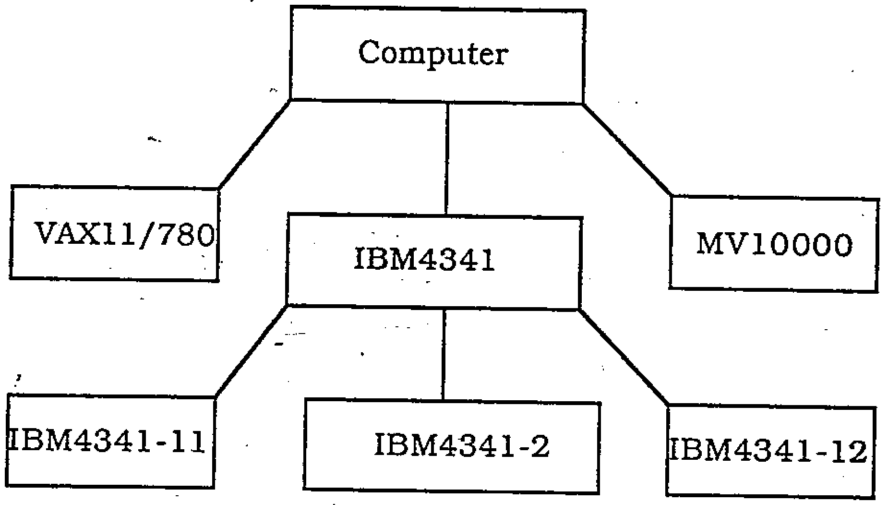

Fig 2.2 Generalization Hierarchy for Computer

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.

In conceptual modeling, an object-oriented database differs from a relational database is noted below.

(1) Class extensibility

With a relational database, there is only a single parameterizable type, Relation. The operations on all relations are limited to get-field-value and set-field-value. In the object-oriented approach, each object is associated with a class. User defined types, or classes, are at the same semantic level as the built-in types. Providing user defined data type, make it natural to model the complexities and variations that occur in real data.

(2) Object is a semantic unit

In the relational model, to reflect that two people have the same set of children requires either a relation representing named sets of children, or a rather complicated data dependency. In the first case, the indirect reference to the set is visible to users. In the second case, the set of children cannot be referenced as a unit. Object-oriented data model provide object identity and user defined data type ease the modeling problem.

(3) Semantic Enhancement

Hierarchies of abstract data type express the semantics of "is a", "is instance of", and "has part of" between classes or objects. The abstract data type are therefore logically organized by these explicitly stated semantic relationships rather than being a collection of independent data types.

Chapter 3.

Forming Virtual Classes And Views

The approach to developing user-oriented views of abstract objects as presented in this Chapter is centered around the notion of virtual class. A virtual class is the window through which a user sees a set of abstract object existing in the conceptual database. The virtual classes of a particular user group is structured by the two abstraction mechanism: generalization and aggregation to form a semantic virtual schema. This semantic virtual schema is the user view of the conceptual database.

3.1 Definitions

[definition 1] A virtual class is a class, but no instance that populates this class. A mapping is available from this class into other classes to extract the properties. in other words actual class are <property,instances> pairs, virtual class are <property,mapping> pairs.

[definition 2] A virtual schema is defined as a schema in which one or more classes are virtual classes [TYI88]. We consider every valid virtual schema should satisfy the is-a condition in generalization plane and the part-of condition in aggregation plane.

[definition 3] Actual objects are real-world individuals. The concept of derived object which is the different view of an object, however provides a second way to see an actual object. A derived object still has object identity and attributes but both are extracted. Clearly speaking, an actual object and its corresponding derived objects denote the same real-world individual but hold different attributes.

[definition 4] A virtual object is defined as a new object, which is created by join or cartisian_product operator. Intuitively, virtual objects are dependent objects. We can denote it by the system_generate surrogate or just the set of identities involved.

3.2 Operator Approach

In our work,we use "operator" approach to derive virtual class and virtual schema. This approach is the same as using relational algebra to derive user view in relational database systems, but the "operator" presented here keeps more semantic_oriented than the relational operator does. It is mainly because the based data model has semantic enhancement. 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.

3.3 Taxonomy of Operators

Operators are defined by carefully exposing the relationships among the basic components of the object-oriented data model. The basic components include 4 elements, namely objects, object identities, classes, and relationships. The functionality of operators include that restrict instances of a class, change the structure of two hierarchics, construct the is_a and part_of relationships between classes, change attributes into a class, move class references along the generalization structure, and do subset or reorder the attributes of a class. The user 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 derivate views and will shield much tedious programming efforts in deriving views from a database.

Since "operator" can be combined to create higher level operators, we discuss only basic operators. Ignoring relation join and cartisian_product operators, there are four types of basic operators.

(A). extension operators which create a virtual class by extracting instances from actual or virtual 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 restructuring a database schema, it is better to divide the database, from the beginning, into two separate domains: database extension domain and database intension domain. In the former, problems appear to care the database itself. In the specification, it becomes to find ways to restrict or select instances from specific classes. Operators in (A) and (D) are dedicated to this kind of problems. In the latter, problems appear to change the definition of the schema, or modify the structure of the abstraction hierarchy. In the specification, it becomes to find ways to reorder or hide the attributes of objects, or change the structure of abstraction hierarchies. Operators in (B) and (C) are addressed to these problems. Since the data schema is changed after restructuring the intension domain of a database, the variation of extension domain cannot be forbidden after we modify the data schema.

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 (Fig1) which contains four classes: person,student,advisor and assistant. In that, person is a generalization of student and advisor. Assistant,student and advisor form a multiple inheritance hierarchy. The instances and the corresponding attributes of each class are listed as follows:

attribute(person)=pid,age,sex,faculty.

(*all attributes of the person class*).

all_inst(person)=O1--O7.

(* all instances that belong to the person class*).

attribute(student)=pid,age,sex,faculty,s#,sname,degree.

all_inst(student)=O3--O5.

attribute(advisor)=pid,age,sex,faculty,a#,aname.

all_inst(advisor)=O5--O7.

attribute(assistant)=

pid,age,sex,faculty,s#,sname,degree,a#,aname.

all_inst(assistant)=O5.

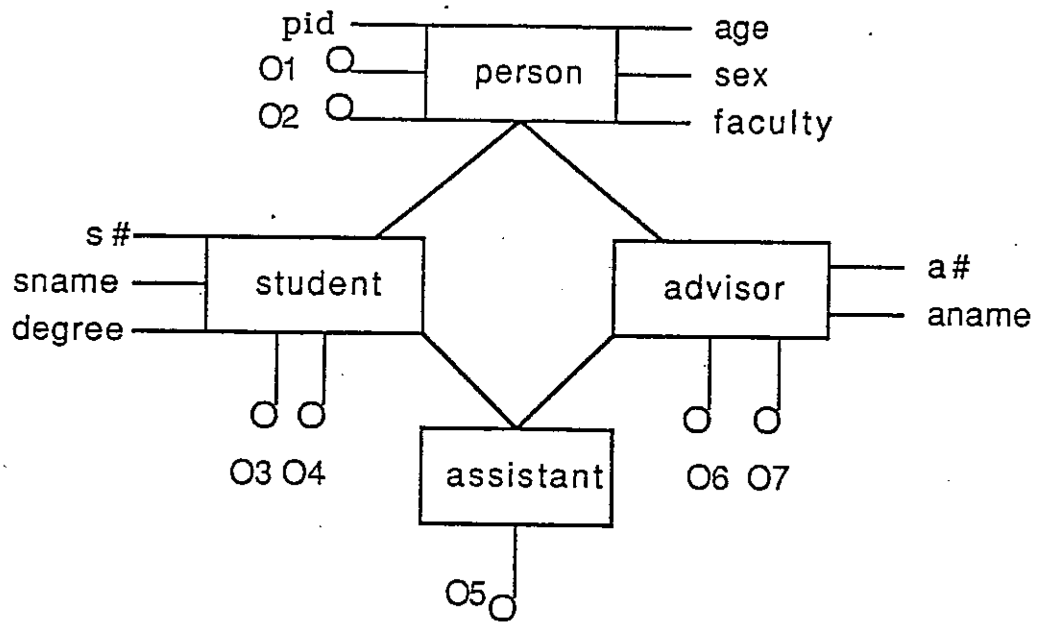

Fig. 1 A Conceptual Database

3.3.1 Extension Operators

These operators apply on the instance domain of an actual/virtual class. They are the familiar set operators, and are used finding the extension domain of virtual classes.

(a) C select: all_inst

(*select all instances belong to the class C*).

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): the same as C.

Example:

person select: all_inst.

(*return a class whose instances are the same as the instances in the person class*).

(b) C select: direct_inst

(*select all instances belong to C but does not belong to subclasses of C*).

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).

Example 1:

person select: direct_inst.

(* return a virtual class whose instances are the same as the instances in the person class *).

Example 2:

student select: direct_inst.

(* return a class whose instances are students and not assistants *).

(c) C select: aQualification

(*return a subcollection each of whose elements satisfies a Qualification*).

Example 1:

person select: [:x| x age < 30].

(*return young person*)

Example 2:

student select: [:x| x not in assistant].

(* return a class whose instances are students but not assistants *).

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 student.

all_inst(V): all_inst(student) - all_inst(assistant).

Example 3: (*employee is a class*).

student select: [:x| x in employee].

(* return a class whose instances have two roles: student and employee *).

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 student.

all_inst(V): intersect(all_inst(student), all_inst(employee)).

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

(d) rename A to B

(* change the name of class A*)

3.3.2 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 in the same data model. Hence, redundance is inhibited and data integrity is preserved.

Another important reason of providing generalization view is that abstraction is supported. Thus objects in a virtual generalization schema can have different levels of perspective, that is, objects can be perceived by a user at different levels of detail in different time, or may be looked at by a user from different angles in different situations.

Now, we begin the description of the individual restructuring operator, then provide examples to show the construction of virtual schemata from Fig1.

(a) partition aClass into (classes) {with discard} as select: Qualifications.

(*according to Qualifications partition aClass*).

This operation creates virtual classes:(classes) by partitioning aClass according to some attributes. The optional expression "with discard" means to delete Qualification attributes.

Example 1:

partition person into (Young,Old) as select: [x:| x age<30,x 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.

(*rank is a catalog attribute which value is some name of subclasses. In this example rank value may be "student", "advisor", or both depending on the class to which the instance belongs.*).

all_inst(Young)={young person}.

attribute(Old)=pid,age,sex,faculty,rank.

all_inst(Old)={old person}

Example 2:

partition Young into (Youngman,Youngfemale) with discard as select: [x:| x sex=man, x sex=female].

resulted virtual classes descriptions:

attribute(Youngman)=pid,age,faculty,rank.

(* the qualification attribute: sex is discarded*).

all_inst(Youngman)={young man}.

attribute(Youngfemale)=pid,age,faculty,rank.

all_inst(Youngfemale)={young female}.

(b) Gen (subclasses) into Superclass

(*create a generalization schema*)

This operation creates a virtual class: Superclass, which is a generalization of subclasses and declares is_a relationships between Superclass and subclasses. The instances and the corresponding attributes of Superclass are listed as follows:

attribute(Superclass): intersect attribute(subclasses).

all_inst(Superclass): union all_inst(subclasses).

Example:

Gen, (student,advisor) into person.

Result: as Fig.1.

(c) Object-join (superclasses) into Subclass

(*setup a multiple inheritance hierarchy*)

This operation creates a virtual class: Subclass which is a specialization of superclasses, and declares is_a relationships among superclasses and Subclass. The instances and the corresponding attributes of Subclass are listed as follows:

attribute(Subclass): union attribute(superclasses).

all_inst(Subclass): intersect all_inst(superclasses).

Example:

Object-join (student,advisor) into assistant.

Result: as Fig.1.

Resulted assistant descriptions:

attribute(assistant):

pid,age,sex,faculty,s#,sname,degree,a#,aname.

all_inst(assistant):

{instances | those are both student and advisor}.

(d) merge (classes) into C

(*merge (classes) which has the same attributes *).

Resulted virtual class C descriptions:

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

all_inst(C): union all_inst(classes).

(e) subtyping A to B

(* B is the super class of A*)

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

(f) specialize Superclass into (subclasses) {withdiscard} as select: Qualifications.

(*according to qualifications create a generalization schema*).

Note that specialize is a composite operator, which can be realized by partition and Gen operator.

In Fig.1, person is divided into student, advisor, and assistant. If we are interested in the faculty that every person participates in. We can partition person (according to the attribute "faculty") into CSfaculty,EEfaculty,and Linquistic as shown in example 1.

Example 1:

(1). partition person into (CSfaculty,EEfaculty,Linquistic) with discard as select: [x: | x faculty=CS, x faculty=EE,
x faculty=Linquistic],

(2). Gen (CSfaculty,EEfaculty) into Engineer,

(3). Gen (Engineer,Linquistic) into person.

apply (1),(2),(3) we get a generalization virtual schema (Fig. 2) from Fig. 1

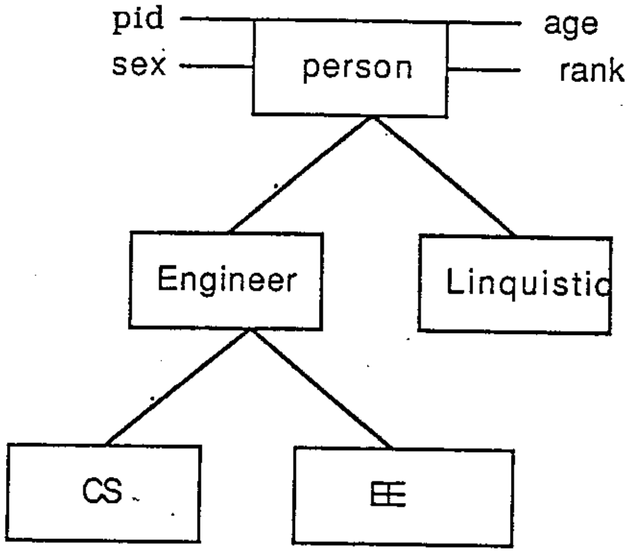

Fig. 2 A Generalization View of the Conceptual Database

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 as shown in example 2:

Example 2:

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

(3). Gen (Young,Old) into person.

apply (1),(2),(3) we get another generalization virtual schema (Fig. 3) from Fig. 1.

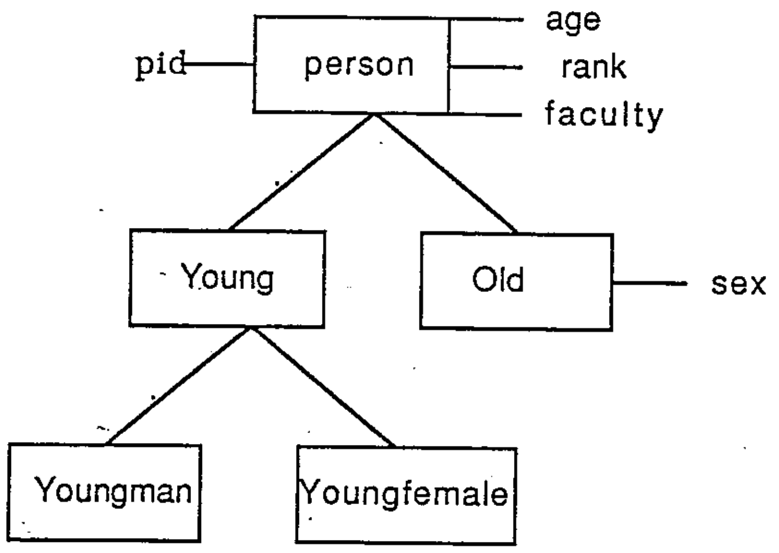

Fig. 3. Another Generalization View of the Conceptual Database

From example 1&2 we acquire different virtual schemas from the same source of conceptual schema by applying a sequence of operators on Fig1.

In summary, Partition and specialize operators are needed when users want to consider objects which have certain properties as a whole new class. For example, the person class can be divided by marriage_status, by age, by sex,or by profession depending on the user's intention. Note that they are composite operators and may be simulated by extension operators and Gen operators.

It is useful to discover thoroughly the power of Object-join in finding the relationship among objects. We know entities in real world may have different roles in different situations. For example: a Person may be an Employee in broad day and is a Student at night. In the database, we can model this situation easily by the declaration of some objects in Employee and Student having the same object identities. The Object-join operator eases the way to relate these same identity objects. Moreover, if the conceptual model does not support the multiple inheritance function, this operator will serve the work, that is, we can use the object_join operator to construct the desired multiple inheritance schema in the virtual database.

3.3.3 Aggregation Hierarchy Operators

It is well known that the real world can be carefully modeled in terms of two types of primitives: entities and relations. Relations mentioned in this section has the same meaning as the relationships in Entity-Relationship models. Entities correspond to real-world objects 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, relation is a higher-level object that has objects as its attributes. 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. For convenience sake, the participants of an abstract object may be regarded as the attributes of that abstract object.

The remainder section is the description of the individual aggregation hierarchy operator and examples to show the construction of aggregation virtual schemata.

(a). Typing $S(a_1...a_m)$ into C.

(* apply attribute aggregation to form C class, then remove $a_1...a_m$ from S, and then construct a relationship between S and C *).

This operation creates a virtual class C by grouping some attributes:$(a_1...a_m)$ of S, (assume in the beginning attribute(S)=$a_1...a_m, a_{m+1}...$) and declares the "part_of" relationship between S and C. The instances and the corresponding attributes of the virtual class are listed as follows:

attribute(C): $a_1...a_m$.

all_inst(C):

{instances | those derived objects that come from S class }.

attribute(S): C,$a_{m+1}...$

(*C is an attribute which value is some instance of the Class C*).

all_inst(S): no change.

Example:

thesis(s#,sname,degree,a#,aname,title) is a class.

Apply

(1) typing thesis(s#,sname,degree) into student,

(2) typing thesis(a#,aname) into advisor.

on class thesis, we get an aggregation virtual schema as follows:

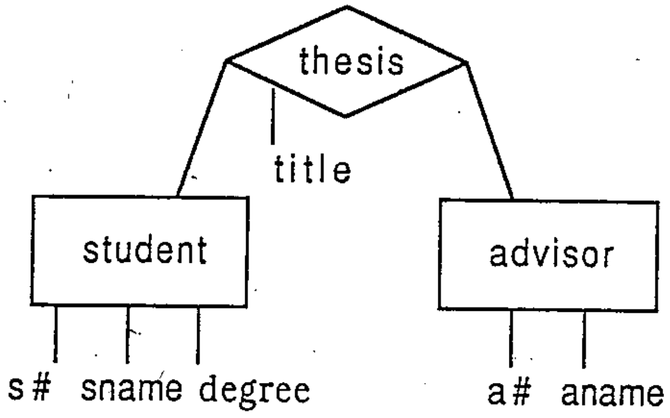

Fig. 4 An Aggregation Schema

(b). expand S(A)

(*transform the relationship between S and A into an attribute aggregation*)

This operation modifies the S class into a new S by replacing the object identities of A with the attributes of A, and the relationship among S and C are also deleted. The instances and the corresponding attributes of the resulted S are listed as follows:

attribute(new S): (attribute(old S)-A) union attribute(A).

all_inst(new S): the same as (old S).

Example:

apply

(1) expand thesis(student)

(2) expand thesis(advisor)

on Fig4.

we get the initial class: thesis(s#,sname,degree,a#,aname,title).

Since there are attribute aggregation and relation aggregation involved in the typing operator. Two different usages appear when we only care partial functionality of a typing operator. First, consider only attribute aggregation which has the functionality of forming attributes into a virtual class. Hence, we can use it to hide the unnecessary attributes of a class. Second consider only relation aggregation which constructs a relationship among classes. We can use it to introduce a new relationship that is not recorded in the original schema. For example, attribute(student)=s#, sname, course, teacher, grade. This statement says that class student has 5 attributes:

s#,sname,course,teacher,and grade. If we issue case1 and case2 as follows to the student class. It will create two virtual classes: enrollment and course_taken in that some undesired attributes of student are hidden.

case 1. typing student(sname, course,grade) into enrollment.

case 2. typing student(sname,course) into course_taken.

case 3. rename student to record,
typing record(s#,sname) into students.

In case 3. a virtual class students is created by grouping attributes: s# and sname as a class. More of this, a relationship between students and attributes of record is added. The final schema of case 3 states that record is a class, which associates students with 3 attributes: course,teacher, and grade.

Since relationships are abstract objects, the above mentioned operators can apply on them. In the paper of "Object Integration in Logical Database Design" [EN84] the author provide a figure that shows a few relationship transformations in an extended E-R model. In Fig. 5 we do the same transformations by operators defined in this thesis, and show

how such restructure equivalent views may arise.

In Fig. 5, a uni-direction edge between two schemata represents a kind of transformation. For each edge, a sequence of operators which does the task of transformation is presented aside. As for label [1] marked in Fig.5 the presented operator sequence changes the attribute "a" of class R into a virtual class. As for label [2], splitting a ternary relationships to two binary relationships is done. As for label [3] the presented operators will merge the splitted schema to the original tenary relationships. In label [4] hiding a certain attribute is done. In label [5] the relationship is absorted. As for the last label[6] is-a relationship is represented by aggregation relationship.

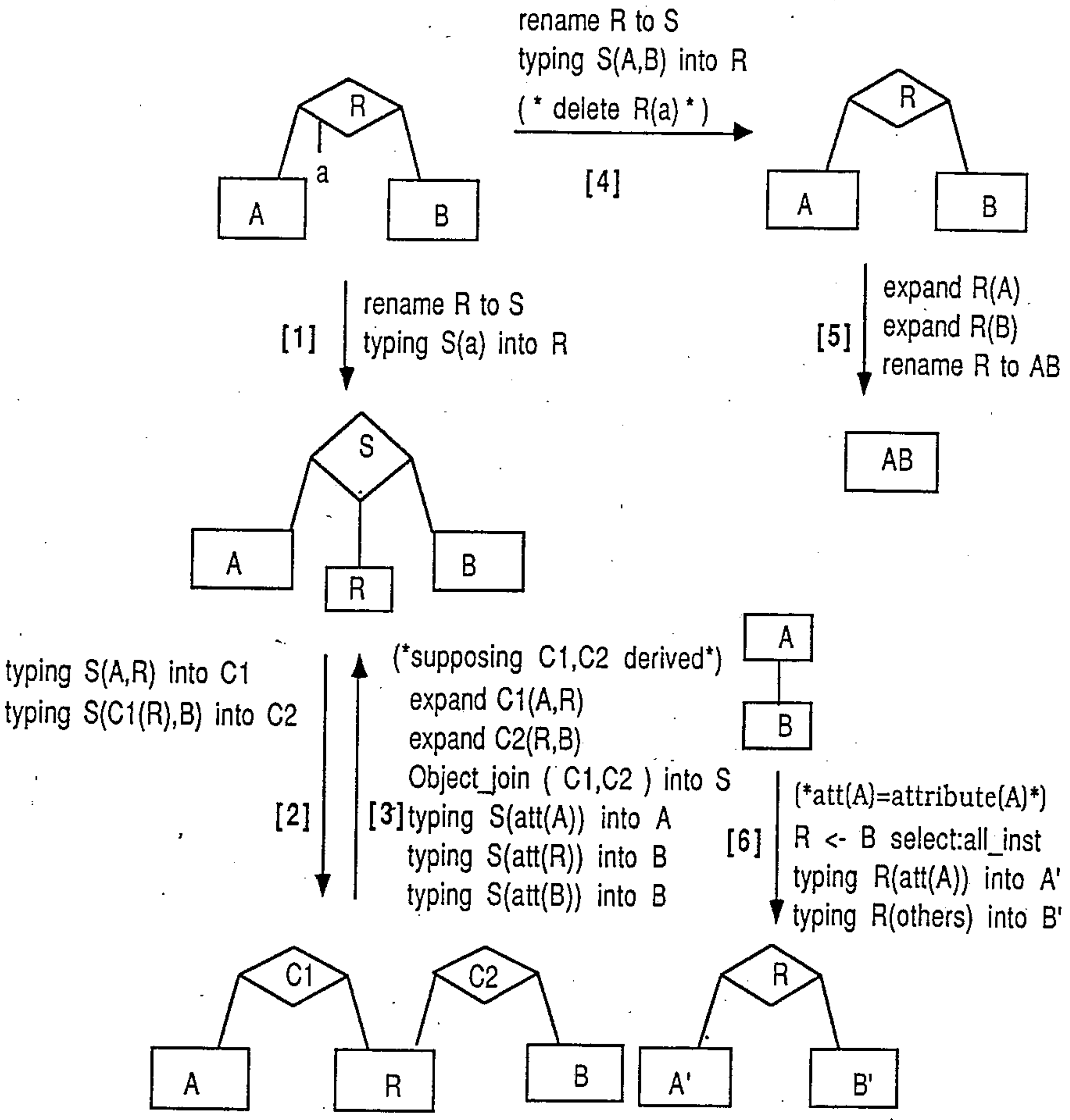

Fig. 5 Relationship Transformation

3.3.4 Reference Operators

In generalization hierarchy an object may have many levels of abstraction, so objects will hold different number of attributes when reference from the different abstraction level of the generalization plane. This motivates us to discover operators to easy the task of using an object up/down the generalization plane. In the following description, changing the scope of reference means changing the reference of objects on different levels of abstraction, and this will affect the number of attributes of an object's own when we display it.

(a). dot operator: "."

(*denote the aggregation component which may be an object or a whole class depending on the context*).

Example:

thesis.student denotes the students who are participating in the thesis.

(b). super_ref

(*change scope of reference from subclass to superclass*).

(c). sub_ref

(*change scope of reference from superclass to subclass and restrict all_inst of superclass to subclass*).

Example: In Fig.4 classes: thesis, student, advisor form an aggregation hierarchy. If we want to create a virtual schema in which virtual classes: phd_thesis, phd_student, phd_advisor form an aggregation hierarchy (Fig.6). We can apply operations (1)--(6) listed below on Fig1&Fig4.

(1). phd_student <- student select:[:x| x degree="phd"]

(2). subtyping phd_student to student

(3). phd_thesis'<- thesis select:[:x| 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:[:x| x.advisor sub_ref phd_advisor]

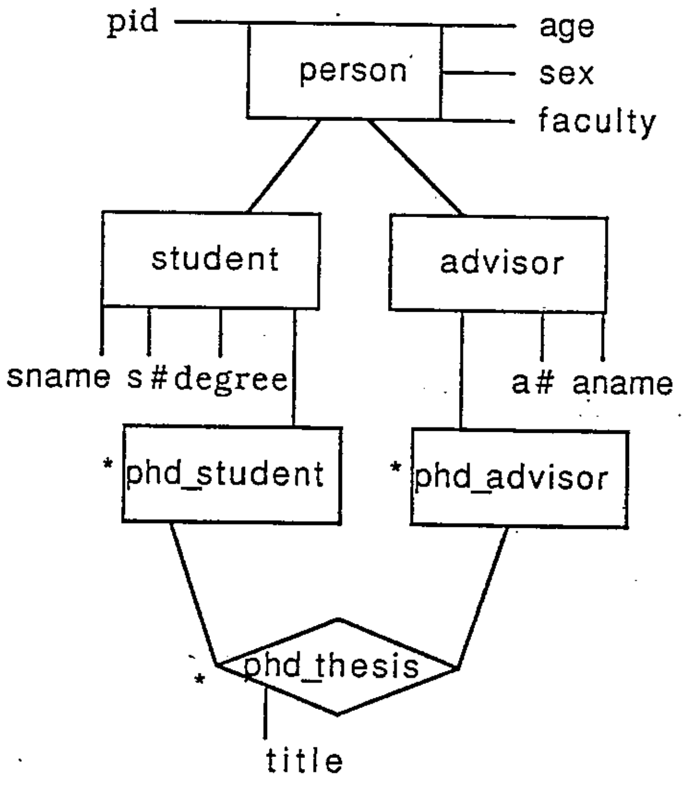

Fig. 6 A 2-D Schema

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 is required when we display the object. Since objects in an generalization hierarchy have different levels of view. Change the reference class of a participant/component object to its sub/super class will change the number of attributes being expanded.

For example: A schema include 4 class: Part,Price,NT$,US$.

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$.

case 1. (a) Part_nt$_charge <- Part select: [:x| x price.cost sub_ref NT$].

(b) expand part_nt$_charge(price).

(* show part in NT$*).

case 2. (a) Part_us$_charge <- Part select: [:x| x price.cost sub_ref US$].

(b) expand part_us$_charge(price).

(* show part in US$*).

In user's perspective, case1 displays Part as a class has attributes:(p#,unit,nt$), but in case2 it shows Part as a class has attributes:(p#,unit,us$).

Chapter 4. Mapping of views

In this Chapter we provide methods to translate user's query on the virtual database into query operations on the conceptual database. The query translator first constructs the derivation of virtual schema as a query graph, then query on the virtual database will impose on this graph.

4.1 Query Graph and Translation Method

We represent a class and the domains of all its attributes involved in a query in a form of a directed graph, which we will call a query graph. Each node on a query graph represents a class-set, and an edge from a node A to a node B means that the class-set B is derived from the class-set A. For each edge in a query graph there is an operator adhered with, which we call a derivation operator. An example of a query using the Partition operator is in the following:

(partition Person into (Young,Old) as select: [x: | x age < 30, x age >= 30])

The query graph corresponding to this derivation will have only two nodes, the class Person as the leaf, and the class-set (Young,Old) as the root. It states that the virtual class-set (Young,Old) is derived from the class Person, and the derivation operator is (partition Person into (Young,Old) as select: [x: | x age < 30, x age >= 30]).

Since for each edge in a query graph there is a derivation operator adhered with. During query processing, each edge in the query graph is associated with an individual derivation procedure. The individual derivation procedure is obtained by interpreting the combination of user's query and the derivation operator on the query graph. The resulted query program is synthesized by concatenating the associated derivation procedures in the query graph. Two examples of retrieval queries using select_all_inst are illustrated in the following:

Example 1: (Young select:all_inst).
This statement retrieves all the instances belonging to Young (see the above example to get the derivation operator of the virtual class Young). Obviously, the derivation procedure between (Person,Young) for this select_all_inst query is (select:[x:| x age <30]). The result of this query is the same as doing the

operation: (Person select:[x: | x age < 30] select:all_inst).

Example 2: (phd_thesis select:all_inst).

This statement retrieves all the instances of the virtual class: phd_thesis in Fig6.

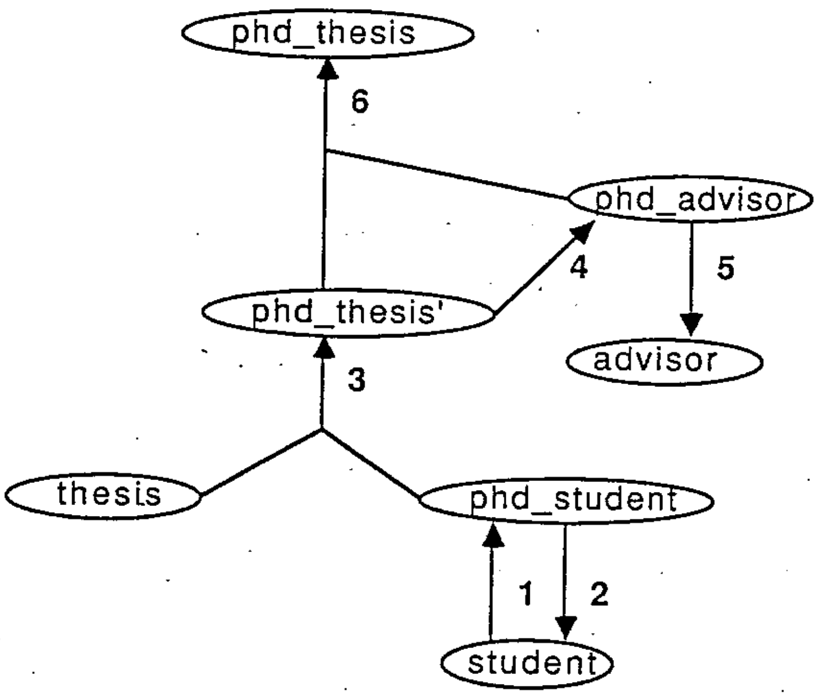

Fig. 7 a Query Graph

The query graph of this virtual schema is shown in Fig.7. The individual derivation operator for each edge(see example about Fig.6) are listed, for convenience, as follows:

(1). phd_student <- student select:[:x| x degree='PHD']

(2). subtyping phd_student to student

(3). phd_thesis' <- thesis select:[:x| 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:[:x| x.advisor sub_ref phd_advisor]

The corresponding derivation procedures are listed below:

1. select:[:x| x degree='PHD']
2. see below
3. select:[:x| x.student sub_ref phd_student]
4. select:all_inst
5. see below
6. select:[:x| x.advisor sub_ref phd_advisor]

Note that there are not any derivation procedures associated to operators (2) and (5), for they are only declaration operators and would be transformed into internal schema constraints. Results of this example is the same as doing the operation:

```
thesis select:[:x| x.student sub_ref (student select:[:x| x
degree='PHD'] )] select:[:x| x.advisor sub_ref
{
(thesis select:[:x| x.student sub_ref (student select:[:x| x
degree='PHD'])]
).advisor select:all_inst
} ]
```

4.2 Translation of Retrieval Queries

We can distinguish retrieval queries into three basic queries:

case 1. select:all_inst.

case 2. select:[:x|x attribute op a_value].

case 3. display:[:x| x attribute].

In case1, all the instances (including attributes) belonging to the receiver class is returned. In case 2, the operation restricts the set of objects to be returned by some qualified attributes. In case 3, the operation displays the specified attribute values from the receiver class.

A complex retrieval query may be regarded as the combination of these basic retrieval queries. For example, a query to list the names of students who received the grade A in the course CS100 may be regarded as the combination of case 2 and case3 queries. The combined statement is shown as follows:

```
(enrollment select:[:x|x course="CS100" and x grade="A"]
display:[:x| x
student name]).
```

According to the above query mapping method, when we apply retrieval queries on virtual classes which was defined by a sequence of operators as presented in Chapter 3, then for each combination of derivation operator and retrieval query there is a derivation procedure needed to be developed. Since the formation of derivation procedures are obviously in most cases. We only describe the formation of derivation procedures on Gen and Object-join in the following:

(1). Gen (A,B) into C.

case 1. C select:all_inst.
the derivation procedure is:
C1 union C2. (* union on object_identity *).
C1,C2 is defined as follows:

typing(A select:all_inst)(att(C)) into C1.

typing(B select:all_inst)(att(C)) into C2.

Example: assume that person is the Gen of student and advisor. The answer to show all_inst(person) is the union of the answer to show all_inst(student) and all_inst(advisor).

case 2. C select:[:x|x attribute op a_value].

the derivation procedure is:

C1 union C2. (* union on object_identity *).

C1,C2 is defined as follows:

typing (A select:[:x|x attribute op a_value])(att(C)) into C1.

typing (B select:[:x|x attribute op a_value])(att(C)) into C2.

Example: person select:[:x| x age = 30]. This statement will select someone whose age is 30 from person. The answer is obtained by union the qualified person from student and advisor.

case 3. C display:[:x| x attribute].

the derivation procedure is:

(A display:[:x| x attribute]) union (B display:[:x| x attribute]).

(* union on object_identity *).

(2) Object-join (A,B) into C

case 1. C select:all_inst.

the derivation procedure is:

(A select:all_inst) object_join (B select:all_inst).

(*object-join means first to perform intersect on object_identity, then perform union on attributes*).

Example: assume that assistant is the subtype of student and advisor. The answer of all assistants will come from instances who are both student and advisor.

case 2. C select:[:x|x attribute op a_value].

the derivation procedure is:

(A select:[:x|x attribute op a_value]) object-join (B select:[:x|x attribute op a_value]).

case 3. C display:[:x| x attribute].

the derivation procedure is:

(A display:[:x| x attribute]) object-join (B display:[:x| x attribute]).

4.3 Discussion

We have talked retrieval queries in the above section. In general, retrieval queries are easier to be treated than insert or delete queries. Because ambiguities may occur in the translation of delete or insert queries. For example: Gen(student,advisor) into person. This statement creates a virtual class: person by generalizing student and advisor. If we insert an instance into person ,then some ambiguity occurs. We do not know where to insert the instance. The place may be student, advisor or both. An ambiguity also appears in the delete operation, If the instance to be deleted has both role of student and advisor. The ambiguity is that we do not know which one to be deleted.

Note that the mapping method described in section 4.1 also serves for delete and insert queries. Now, We give a brief discussion on the updating problems of virtual database. From the definition of basic operators we know an one to one mapping between a derived object and its corresponding real object is well preserved(except Gen). Due to this fact It is no doubt to update (including retrieve,insert,and delete) a virtual database created by the four kinds of basic operators except Gen as mentioned above. The main reason is that each object in the virtual class,which was defined by basic operators, is a derived object. In the following examples we show some update operations on this kind of virtual classes. Example1: Object_join(student,advisor) into assistant. This statement creates a virtual class: assistant, and specifies there is a multiple inheritance hierarchy among assistant, student and advisor. If we insert an instance into assistant, it would be translated to insert the instance into student and advisor. Example 2: Young <-- person select:[x:| x age <30]. This statement creates a virtual class: Young by placing restriction on the instances of person. If we insert an instance whose age larger than thirty into Young, it would be rejected by the system because instances in Young must have age under 30.

For those so called virtual objects, the associated updating problem is the same as in the relational model, but something is preferred in object-oriented approach. Recall that an object is a well defined ADT(abstract data type). We can use ADT approach [RS79] to define the set of allowed updates and their translation operations on virtual objects so that update is no longer anomalous and ambiguity is avoided.

Chapter 5. Conclusion

In this thesis operators are defined by carefully exposing the relationships among the basic components of an object-oriented data model. The functionalities of operators include restrict instances of a class, change the structure of two hierarchies, construct the is_a and part_of relationships between classes, change attributes into a class, move class references along the generalization structure, and do subset or reorder the attributes of a class. The user view is constructed by issue 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 derivate views and will shield much teadious programming efforts in deriving views from a database.

Constructing the semantic virtual database (user view) as a 2-D hierarchy structure, we obtain several advantages 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 works. Second, in a semantic virtual database, the basic construct: virtual class provides a second way to see user-specific objects without costly data redundancies. Hence, updating a database is simplified and data integrity can be enhanced. And third, the 2-D hierarchy of virtual schemata structures user's data with levels of abstraction. This gives user a wholly concise view of a virtual database.

In previous chapters, we do not mention any inheritance of methods between view objects and real objects, for there are not a straight inheritance rule if it is not obvious, and occasionally the inherited operations are not well suited for the new view objects. If inherited methods are required, It is no doubt we can cover it using the abstract data type approach by declaring methods to be inherited or even including the newly encapsulated operations in the definition of virtual classes.

In summary, we conclude the works in our research as follows: (a.) User views is defined as a semantic virtual schema, which no data populate, but instead mappings are available from the virtual schema into the conceptual schema. The mapping of the virtual database into the conceptual database is transparent to user, so the usage of user views is the same as it being in the conceptual schema. The main reason is that the structure of the virtual schema is designed as close as the structure of the conceptual data schema. (b.) 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 develop or process a high level query language such like OSQL in an object-oriented database system. (Object SQL was developed in the Iris project.) (c.) Query translation and update method is discussed.

Further researches are needed for coping with the following problems:

1. Operators defined in this thesis have it's syntax in low level. A high level syntax is needed for supporting declarative query language like Object SQL. Nevertheless, the data model of Object SQL based upon is on functional approach. Research is required when we want to design the high level database language for the model we adopt.

2. As mentioned in the Introduction, it is another goal of this

thesis to find the basic operations for the underlying data model. Operators defined in this thesis are all central around the virtual class concept. Changing the target from classes to objects that operators apply on, we can find some useful operators that help to manipulate objects in the database.
for example:

a. Reference a class to its sub/super type will introduce the issue of up/down operators, which change the abstraction level of an object to its corresponding sub/super object.

b. Typing operator will be useful in hiding the undesired attributes of a class. it introduces the issue of scope operator ,which only binds the interested attributes to an object.

c. Object_join operator helps to define the unite operator, which merges all attributes of an object who has many types.

However, more operations to manipulate objects, or operations to act on object identities, and operations to define integrity constraints are still needed.

References

[BCGK87] J. Banerjee, H. Chou, J. Garza, W.Kim, D. Woelk, N. Ballou, and H.Kim. "Data Model Issues for Objected-Oriented Applications", ACM Transactions on Office Information Systems, 5(1), pp.3-26, January 1987.

[BK88] J. Banerje, and W. Kim. "Semantics and Implementation of Schema evolution in Object-Oriented Databases", In Proceeding of the ACM SIGMOD International Conference on Management of Data, pp. 311-322, May 1987.

[BKK88] J. Banerjee, W.Kim, and K.C. Kim. "Queries in Object-Oriented Databases", in Proceeding of the Fourth International Conference on Data Engineering, pp. 31-38, February 1988.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. "A comparative analysis of methodologies for database schema integration", ACM Computing Surveys, 18(4), pp.323-364, December 1986.

[BLS82] C. Batini, M. Lenzirini and G. Santucci. "A Computer-Aided Methodology for Conceptual Database Design", Information Systems, 7(3), pp. 265-280, 1982.

[CM84] G. Copeland and D. Maier. "Making Smalltalk a Database Syatem", In Proceedings of the ACM SIGMOD

Conference, pp. 316-325, June 1984.

[Codd70] E. F. Codd. "A relational model of data for large shared data banks", CACM 13(6), pp. 377-387, June 1970.

[Codd79] E. F. Codd. "Extending the database relational model to capture more meaning", ACM Transactions on Database Syatems", 4(4), pp. 397-434, 1979.

[Date83] C.J. Date. " An Introduction to Database Systems, Volume II.", Addison-Wesley, Reading, Massachusetts, 1983.

[Date86] C.J. Date. " An Introduction to Database Systems, Volume I.", Fourth Edition, Addison-Wesley, Reading, Massachusetts, 1986.

[Ditt86] K.R. Dittrich. "Object-Oriented Database Systems - A Workshop Report", in Proceeding of the Fifth Entity Relationship Approach Conference, North Holland Publishing Groups, pp.51-66, 1986.

[EN84] R. Elmasri and S. Navathe. "Object Integration In Logical Database Design", In Proceeding of the 1st International Conference on Data Engineering, Los Angeles, pp. 426-433, April 1984.

[FC85] A. L. Furtado and M. A. Casanova. "Updating Relational Views", in Query Processing in Database Systems, W.Kim, D.S.Reiner, and D.S.Batory, eds., Springer-Verlag, pp. 127-142, 1985.

[Fish87] D. Fishman, et. al., "Iris: An Object-Oriented Database Mangement System", ACM Transactions on Office Information Systems, 5(1), pp. 48-69, January 1987.

[GR83] A. Goldberg, and D. Robson. "Smalltalk-80, The Language and Its Implementation", Addison-Wesley, 1983.

[HK87] R. Hull, R. King. "Semantic Database Modeling: Survey, Applications, and Research Issues", ACM Computing Surveys, 19(3), pp. 201-260, September1987.

[KC86] S. N. Khoshafian and G. P. Copeland. "Object Identity", In OOPSLA Proceedings, ACM SIGPLAN NOTICES, 21(11), pp. 406-416, Nov. 1986.

[KM85] R. King, and D. Mcleod. " Semantic Data Models", In "Principle of Database Design", S. Yao (et.), Prentice-Hall, pp. 115-150, 1985.

[KS86] M. L. Kersten, and F. H. Schippers. "Towards an Object-Centered Database Language", In Proceeding of the 1st International Workshop on Object-Oriented Database Systems, IEEE, pp.104-112, 1986.

[KW87] A. Kemper, and M. Wallrath. "An Analysis of Geometric Modeling in Database Systems", ACM Computing Surveys, 19(1), pp. 47-91, March 1987.

[LM88] Q. Li and D. McLeod. "Object Flavor Evolution in an Object-Oriented Database System", In Proceeding of the Conference on Office Information System, ACM, pp. 265-275, March 1988.

[Motro87] A. Motro. "Superviews: Virtual Integration of Multiple Databases", IEEE Transactions on Software Engineering, 13(7), pp. 785-798, July 1987.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. "Development of an Object-Oriented DBMS", In OOPSLA Proceedings, pp. 472-482, October 1986.

[PM88] J. Peckham and F. Maryanski. "Semantic Data Models", ACM Computing Surveys, 20(3), pp. 153-189, September 1988.

[RS79] L. Rowe, and K.A. Schoeus. "Data Abstractions, Views and Updates in RIGEL", In Proceeding of ACM SIGMOD, pp. 71-81, May 1979, also available in "The INGRES Paper", Chapter 15, Addison-Wesley, Reading, 1986.

[Rumb87] J. Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language", In OOPSLA Proceeding, pp. 466-481, December 1987.

[Shen85] S. Shen. "Design of a Virtual Database", Information Systems, 10(1), pp. 27-35, 1985.

[SMF86] D.L. Spooner, M.A. Milicia, and D.B. Faatz. "Modeling Mechanical CAD Data with Data Abstraction and Object-Oriented Techniques", In Proceeding of the International Conference on Data Engineering, IEEE, pp. 416-424, 1986.

[SS77] J.M. Smith and D.C.P. Smith. "Database abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, 2(2), pp. 105-133, March 1977.

[TYI88] K. Tanaka, M. Yoshikawa and K. Ishihara. "Schema Virtualization in Object-Oriented Databases", In Proceeding 4th International Conference on Data Engineering, Los Angeles, Califorina, pp. 23-30, 1988.

[WKL86] D. Woelk, W. Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases", In Proceeding ACM SIGMOD Conference on the Management of Data, pp. 311-325, May 1986.