Schema Design, Views and Incomplete Information in Object-Oriented Databases

KATSUMI TANAKA,* MASATOSHI YOSHIKAWA** and KOZO ISHIHARA*

"Object-oriented computing" is one of the most active research fields in computer science, and this notion has become an important paradigm in the areas of programming languages, artificial intelligence and databases. In the area of databases, although the relational approach achieved a great success, the weakness of this model has been also pointed out as the database application area extends to CAD/CAM application and software version control etc. As an alternative candidate of the relational approach, much attention has been focussed on "object-oriented database (abbreviated by OODB)", and the OODBs have been investigated by several researchers.

However, the study of the OODB has just begun, and there are few investigations about the database schema design in the OODB. In this paper, we show fundamental ideas and methods to achieve schema design, update propagation, view implementation and the treatment of incomplete information in the OODB. Especially, we show some methods written in Smalltalk-80 to perform these and also discuss the weakness of Smalltalk-80 from the viewpoint of the OODB.

1. Introduction

"Object-oriented computing" has become an important paradigm in several areas of computer science, such as programming languages, artificial intelligence and databases. Recently, in the area of database research, much attention has been focussed on "object-oriented database (abbreviated by OODB)" [1-3]. Although the relational approach achieved a great success in the area of business-oriented data processing, the weakness and the disadvantages of the relational approach have been identified in non-business data applications (CAD databases, office information systems, and multimedia databases).

Although there is neither a consensus nor a definition about what the OODB is, we can claim the following advantages of the OODB:

(1) *Direct representation of objects*

In the OODB, each "entity" or "object" is represented directly. Users can directly interact with and manipulate these objects, not by using the notion of tuples, attributes, and attribute values. Since everything can be regarded as objects in the OODB, the information about database schema and application programs can be treated as objects.

(2) *Operational semantics of database objects*

In the OODB, each database object is a mixture of data and operations which are possible to apply to the object itself. This is a powerful capability for expressing the semantics of data. That is, we can specify the 'operational' semantics of data, and encapsulate it into the database object.

(3) *Generalization hierarchy of database objects*

Database objects are classified and organized as a generalization ("is-a") hierarchy in the way similar to Smalltalk-80 [4]. Properties and operations given at an upper-level object are inherited by its lower-level objects. This is also a powerful capability for expressing structurally the semantics of data.

(4) *Realization of extensible databases [5, 6]*

By the approach of the OODB, there is a possibility to decompose a database schema and DBMS facilities into smaller parts (objects), and to combine necessary parts into an application-oriented database system. This possibility is very important to realize the notions of extensible DBMSs [5, 6].

Although there are a few studies about what is lacking in Smalltalk to construct the OODB systems [1-3], there are few investigations about the database schema design in the OODB. For example, in GemStone [2], no general methodology for schema design of OODB is given. In this paper, we discuss fundamental ideas and basic methods for achieving the schema design, the update propagation, the view implementation and the treatment of incomplete information in the OODB. Especially, we show several methods to perform these in Smalltalk-80 and also discuss the weakness of Smalltalk from the viewpoint of the OODB.

Section 2 discusses the concept of the OODB and some basic issues. In Section 3, we first introduce the

*Dept. of Instrumentation Engineering, Faculty of Engineering, Kobe University.
**Institute for Computer Science, Kyoto Sangyo University.

notion of "object database schema (abbreviated by ODS)". Then, we show how to design an ODS from Chen's ER model [7] with some additional semantic constraints. Also, a method to realize an ODS in Smalltalk is shown. Section 4 introduces the notion of "views" over the OODB. Intuitively, views are virtual objects. Here, we show some basic operations to construct "virtual" classes and "virtual" is-a hierarchies in the OODB, and methods to realize them in Smalltalk. In Section 5, we investigate the methods to handle incomplete information in the OODB and Smalltalk. Section 6 is a concluding remarks.

2. Object-Oriented Database and Smalltalk

The notions of "object-oriented database (OODB)" and "object-oriented data model" are still vague and their definitions depend on several researchers. In this section, we will clarify our standpoints for the OODB. Also, a brief introduction of Smalltalk is given.

There seems to be the following two promising approaches towards costructing an "object-oriented database":

(a) "Semantic data model" approach

This approach is based on the semantic data models such as ER model [7] and IFO model [8], and is to provide several semantic constructs for modeling objects and object-relationships directly, for example, entities, complex objects, attributes, relationships, and generalization/aggregation hierarchies.

(b) "Object-oriented programming" approach

This approach tries to model the application domains by encapsulating data and the associated procedures with the notion of types and inheritance structures.

We believe it is important to integrate these two approaches naturally to realize the ultimate "object-oriented database". However, it is not known completely how this integration can be achieved. In this paper, we mainly took the approach (b), and try to clarify how several constructs in (a) corresponds to those of object-oriented programming languages. As a representative of object-oriented programming languages, we adopted Smalltalk-80. However, we believe many of the ideas and results of this paper can be applied not only to other object-oriented programming languages but also to exploiting fundamental constructs for realizing the ultimate object-oriented database systems.

Smalltalk [4] is a general purpose programming language, and its basic components are as follows:

(1) classes
(2) instances
(3) class/instance variables
(4) class/instance methods
(5) subclass-superclass relationships

Each class is an object to describe a set of instances of the same type. (Here, each instance is also an object.) The description of a class C contains (a) definition of instance variables, each of which is owned by its instance, (b) definition of class variables that are referred by all the instances of C, (c) definition of instance methods, each of which is a Smalltalk program and actually applied to its instance, (d) definition of class methods, each of which is also a Smalltalk program and applied to the class C itself. Each variable contains an object as its value. Between two classes, the sub-super class relationship can also be defined. This is an "IS-A" relationship between two classes, and if C_1 IS-A C_2, then the variables and methods of class C_2 is automatically inherited to the class C_1. Basically, a Smalltalk program is a collection of message expressions, each of which consists of an object (receiver) followed by an appropriate message. According to the message sent to a receiver object, an appropriate method of the receiver or an inherited method is executed. Furthermore, a Smalltalk program by itself can also be an object called 'block', whose execution is done by sending the message 'value' to it. In order to refer to a unique object in Smalltalk system, the "global variable" is also used.

3. Schema Design of OODBs

3.1 Problems of Schema Design in OODBs

As described in Section 2, several semantic database models are already offering some constructs to represent objects and relationships between objects, for example, Chen's ER model [7] and the IFO data model [8]. In designing a database schema in the OODB, furthermore, the following problems should be coped with:

(a) Although Smalltalk offers a mechanism to realize entity sets, attributes and is-a relationships between entity sets of the ER model, no general framework is given to represent relationships between entity sets in ER model and to maintain semantic constrains about relationships, such as 1-1, many-1, many-many.

(b) In representing some data semantics, Smalltalk is superior to conventional semantic database models. For example, the notions of "class variable" and "block" in Smalltalk [4] is powerful to represent some data semantics. The former notion can represent a value which is common to all the entities in an entity set. The latter one allows users to store a procedure (program) as a value of an attribute. However, no framework is known to design a database schema using these constructs in OODBs.

(c) Although Smalltalk is a candidate for realizing OODBs, it is not known to what extent Smalltalk offers enough capabilities to construct the database schemata of OODBs.

(d) We need some general methodology to design 'methods' (instance methods and class methods in Smalltalk) contained in the OODB schema.

To discuss and cope with the above problems, we first introduce the notion of Object Database Schema (abbreviated by ODS). Then, we show a method to transform a schema described by the ER model together

with some additional semantic constraints into an ODS. This is because we wish to investigate how several constructs of a typical semantic data model (ER model, here) corresponds to those of our ODS. Also, we show how to realize a given ODS on Smalltalk.

3.2 Object Database Schema

In this section, we define the "object database schema". Intuitively, an object database schema consists of classes and several relationships. First, we define the components of the object database schema as follows.

A *class* is a 7-tuple (class-name, V_{class}, V_{instance}, V_{derived}, $V_{\text{derived-object}}$, M_{class}, M_{instance}), where

(1) "class-name" is the name of the class,

(2) V_{class} is the set of *class variables*,

(3) V_{instance} is the *instance variables*,

(4) V_{derived} is the set of *derived instance variables*, where "derived" means that the value of the instance variable for each instance object is computed from other objects,

(5) $V_{\text{derived-object}}$ is the set of *derived-object included instance variables*, where "derived-object included" means that some instance of this class is allowed to have a block (program) to compute the value of the instance variable,

(6) M_{class} is the set of *class methods*, and

(7) M_{instance} is the set of *instance methods*.

Except the notion of V_{derived} and $V_{\text{derived-object}}$, the other notions are the same as those in Smalltalk. For simplicity, if we say a class C, then C means both the class name and the class itself. In our diagrammatic notion, a box □ denotes a class.

A *subclass-superclass relationship* from a class C_1 to a class C_2 is an ordered pair (C_1, C_2), which means that C_1 is-a C_2, and that each instance of C_1 is a C_2. This notion is also the same as that of Smalltalk. In our diagrammatic notion, $C_1 \Rightarrow C_2$ denotes the relationship "C_1 is-a C_2".

An *instance-of relationship* from an instance variable V of a class C_1 to a class C_2 is a triple (C_1, V, C_2) (C_1 and C_2 are not always distinct), which means that each value of the instance variable V of class C_1 is an instance of class C_2. In our diagram, $V \xrightarrow{\text{instance-of}} C_2$ denotes the relationship. An *association relationship* for a class C is an ordered pair (C, C^*), where C^* denotes a class whose instance is the set of instances of class C. In our diagram, $C \Rightarrow C^*$ denotes the association relationship.

By these constructs, we define an object database schema as follows:

An *object database schema* (abbreviated by ODS) is a 4-tuple $(C, R_{\text{isa}}, R_{\text{instance}}, R_{\text{association}})$, where C is the set of classes, R_{isa} is the set of subclass-superclass relationships defined over C, R_{instance} is the set of instance-of relationships defined over C, and $R_{\text{association}}$ is the set of association relationships defined over C.

For each class C, its instance variable (or the set of instance variables) is said to be a *key* of class C if the value of the instance variable uniquely identifies its instance.

3.3 Transformation from ER schema to ODS

Here, we discuss a method to transform a database schema described by ER model into our ODS. Also, we show some methods to support an ODS by Smalltalk.

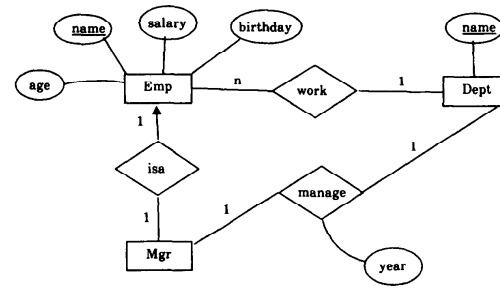
First, we illustrate the process of the transformation by an example. Suppose that the ER schema in Fig. 1 is given. Also, we add the following semantic constraints:

(a) If an employee belongs to 'toy' department, then his salary is equal to the salary of the department manager multiplied by 0.6.

(b) Every manager's salary is fixed at $60000.

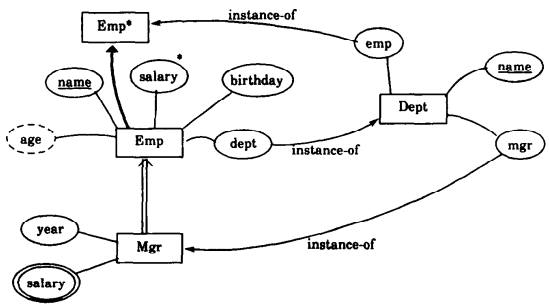
(c) The department which each manager manages is the same as the department which he belongs to as an employee.

First, each entity set is transformed into a class with the same name. (see Fig. 2). The attributes name and birthday are transformed into the instance variables of class Emp. Since some employees' (employees belonging to 'toy' department) salaries are automatically computed according to (a), we regard "salary" as a "derived-object included" instance variable of Emp. Also, since every employee's age can be computed from his



□ : entity set , ◇ : relationship , ○ : attribute
Emp : Employee , Dept : Department , Mgr : Manager

Fig. 1 An Example ER schema.



□ : Class, ○ : instance variable, : derived instance variable
○* : derived-object included instance variable, ◎ : class variable

Fig. 2 An ODS obtained from Fig. 3.1.

birthday, the attribute age is regarded as a "derived instance variable." One method to represent a relationship between class A and B is to prepare instance variables "b" in class A and "a" in class B. For example, we prepare an instance variable "dept" in class Emp and an instance variable "emp" in class Dept. By the values of these two instance variables, it is possible to represent the mutual relationships between an employee object and a department object. Since the relationship "work" is many-to-1, we add the class Emp* and introduce the instance-of relationships (Dept, emp, Emp*) and (Emp, dept, Dept) and the association relationship (Emp, Emp*). As for representing the is-a relationship between Emp and Mgr, we can use the subclass-superclass relationship from class Mgr to Emp. Note that the department value for each manager is inherited from Emp's instance variable 'dept' by the subclass-superclass relationship "Mgr is-a Emp". From the semantic constraint (c) stated above, the relationship "manage" can be represented by introducing an instance variable "mgr" in the class Dept and the following instance method "manage" in the class Mgr.: Here, 'dept' is an instance method to return the value of the instance variable 'dept'.

```
manage
↑self dept
```

Since the relationship manage is 1-to-1, the attribute year associated with manage can be transformed into the instance variable with the same name of class Mgr. or class Dept. (In Fig. 2, the instance variable year is attached to the class Mgr.) Since every manager's salary is fixed at \$60000, we add a class variable salary to class Mgr, whose value is \$60000. Also, note that the class Mgr has the class variable salary and the instance variable salary inherited from Emp. In this case, we can give a higher priority to the class variable salary by refering to it in the instance method salary. Fig. 2 shows a diagrammatic representation of the obtained ODS.

The derived instance variable age is implemented as the following instance method age of class Emp:

```
age
|age|
age←(date today elapsedMonthsSince: birthday)//
      12.
Date today monthName=birthday monthName
ifTrue:[Date today dayOfMonth<birthday
      dayOfMonth ifTrue: [age←age-1]].
↑age
```

The derived-object included instance variable is intuitively implemented as a form of block, which is stored as a value of the instance variable. That is, for each Emp instance, if he belongs to 'toy' department, the following block is stored as the value of the instance variable salary:

```
[(self dept mgr salary)*0.6]
```

To do this, we define the following instance method salary:. Here, 'dept', 'mgr', and 'salary' are instance methods to return the value of instance variables 'dept', 'mgr', and 'salary', respectively.

```
salary: x
dept=Toy
ifTure:[salary←[(self dept mgr salary)*0.6]]
ifFalse:[salary←x]
```

In creating a new Emp instance, the following expression can automatically set up this block.

```
(emp new dept:Toy) salary:arbitraryNumber
```

Finally, in order to maintain the instance-of relationships, we add a type checking mechanism for each instance method to store a value of each instance variable. For example, the instance method dept: of class Emp is defined as follows:

```
dept: x
(x class)=Dept
ifTrue:[dept←x]
```

Here, 'class' is a built-in method to return the class name of the receiver.

We summarize a general procedure of the above transformation in the following:

(1) For each entity set, create a class with the same name.

(2) For each ordinary attribute of all entity sets, create an instance variable with the same name for the class corresponding to its original entity set.

(3) If all entities of an entity set E have the same value for attribute A, then create a class variable A of class E and store the value to it.

(4) For each attribute A whose value is always computed from others, create a derived instance variable A in ODS, and realize it as an instance method A to compute and return the value.

(5) For each attribute A whose value is computed from others for some instances, create a derived-object included instance variable A in ODS, and store a block for computing the value to A.

(6) For each is-a relationship from entity set E_1 to E_2, create a subclass-superclass relationship (E_1, E_2) in ODS and realize it as a sub-super class relation in Smalltalk.

(7) For each binary and many-to-many relationship with more than one attribute, replace all the attributes by entity sets with the same names, and apply (8). Otherwise, for each binary relationship between two entity sets E_1 and E_2, create an instance variable V_2 in class E_1 and an instance variable V_1 in class E_2. Also, create an instance-of relationship in the following manner:

(i) 1 to 1: (E_1, V_2, E_2) and (E_2, V_1, E_1)

(ii) many to 1:(E_1, V_2, E_2) and (E_2, V_1, E_1^*)

(iii) many to many:(E_1, V_2, E_2^*) and (E_2, V_1, E_1^*)

Here, E_i^* denotes a class whose instance is the set of instances of class E_i. For each attribute A (if it exists) associated with the binary relationship, according to the rela-

tionship types, apply the following:

(i) 1 to 1: Create an instance variable with name A in class E_1 or E_2.

(ii) many to 1: Create an instance variable with name A in class E_1.

Add a type checking mechanism (described in the above example) to each instance method to store or update the value of V_i.

(8) For each n-ary relationship with more than one attribute, replace all the attributes by entity sets with the same names. Next, for each n-ary relationship between n entity sets $E_i (i=1, \cdots, n)$, the other method to realize it in ODS and in Smalltalk is the use of "relationship object". This is done according to the following steps:

(i) Create a class E_i for each entity set E_i

(ii) Create a class R which has the instance variable $V_{E_i} (1 \leqq i \leqq n)$. (Here each V_{E_i} has an instance object of class E_i).

(iii) Add an instance variable R to each class E_i.

In fact, the n-ary relationship is decomposed into n binary relationships between E_i and R.

(9) According to given semantic constraints, if we can find redundant instance variables, then remove them.

In Fig. 2, there originally existed the instance variable dept in class Mgr, but it was regarded as redundant and removed by applying the above (9), since the superclass of Mgr (i.e. Emp) also has the instance variable dept and the semantic constraint (c) holds.

As a special case, some binary relationships may be defined over the same entity set. As shown in Fig. 3(a), the relationship mother-of is of this type. To represent it in ODS, first, we transform Fig. 3(a) into a loop-free form. That is a many-to-1 relationship "mother-of" from "Children" entity set to "Mother" entity set. Here, both of "Children" and "Mother" are the names of roles of "Person" in this "mother-of" relationship. Then, the above step (7) can be applied, and we obtain an ODS in Fig. 3(b).

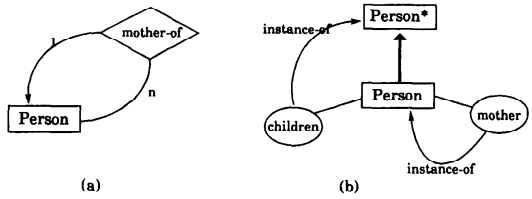


Fig. 3 Recursive relationship.

3.4 Update Propagation

When an update request is issued to an OODB, it is important to check whether the update is valid with respect to given semantic constraints and to reject invalid updates. For example, see a type checking mechanism in the instance method dept: x shown in Section 3.3. Also, it is important to update the contents of the OODB so that the given semantic constraints may be satisfied.

Consider the previous Employee-Department example, where there is a many-to-1 relationship from Emp class to Dept class. [This can be regarded as a "functional dependency" from class Emp to class Dept.] Suppose that currently Bob's department is a Sales object. Assume that we wish to update Bob's department from Sales to *R&D*. In order to make the contents of the OODB after this update satisfy the constraint, the following actions should be executed:

(1) Send a message to ask Sales object to delete the Bob object from the value of the instance variable emp.

(2) Send a message to self (Bob object) to change the value of the instance variable dept from Sales to *R&D*.

(3) Send a message to ask *R&D* object to add self (Bob object) to the value of the instance variable emp.

These actions can be realized by the following instance methods. Each number in the instance method dept: corresponds to the above number.

```
[Instance method of class Emp]
  dept: x
  dept removeEmp: self. ..... (1)
  dept←x. ..... (2)
  x setEmp: self ..... (3)
[Instance methods of class Dept]
  removeEmp: y
  emp←emp remove: y
  setEmp: z
  emp←emp add: z
```

By using the above methods, for example,

Bob dept: *R&D*

will perform the update correctly.

The above is just an example, and we will need a general methodology for designing 'methods' to achieve updates so that the given semantic constraints may be satisfied after updates.

Here, we show a set of general methods which is used to achieve the update propagation for maintaining a class of relationship types, such as 1-to-1, 1-to-many, and many-to-many relationships. Just by adding some "data definition" facility, it is easy to let the system automatically generate the methods for each class which achieve the update propagation correctly for 1-to-1, 1-to-many and many-to-many relationships.

Suppose that there is a binary relationship R between the instances of class A and the instances of class B. Following the previous transformation from *ER* model to ODS, we assume that both class A and B have an instance variable named 'r' which contains each counterpart of the relationship.

(a) If the relationship R is 1-to-1, then the four methods in Fig. 4(a) are defined in both class A and class B.

(b) If R is 1-to-many, then the four methods in Fig. 4(a) are defined in only class A. Also, the four methods in Fig. 4(b) are defined in class B.

(c) If R is many-to-many, then the four methods in

Fig. 4(b) are defined in both class A and class B.

The method "set$R$" in Fig. 4 is used to update the relationship R. Suppose that the relationship R is 1-to-1 and that an instance $a1$ of class A is related to the instance $b1$ of class B (here, we denote it by ($a1$, R, $b1$)). When we wish to update the relationship ($a1$, R, $b1$) into ($a1$, R, $b2$), we do the following:

$a1$ set$R$: $b2$

Since the relationship R is 1-to-1, the execution of the method set$R$ (in Fig. 4(a)) invokes the following three message expressions:

$b1$ removeR': $a1$
$r \leftarrow b2$
r setR': $a1$

The first line is to assign NIL to the instance variable r of the instance $b1$. The second line is to assign $b2$ to the instance variable r of the instance $a1$. The third line is to assign $a1$ to the instance variable r of the instance $b2$. During the execution of the third line, if the relationship ($a2$, R, $b2$) already exists for a certain instance $a2$, first, the following message expression is executed (see the definition of the method setR'):

$a2$ removeR': $b2$

This message expression is to drop the relationship ($a2$, R, $b2$) by assigning NIL to the instance variable r of the instance $a2$.

In the similar manners, the methods in Fig. 4(a), (b) can correctly maintain several (1-to-1, 1-to-many, many-to-many) relationship types when update requests occur. Since the methods in Fig. 4(a), (b) are general ones, it is an easy task to let the system automatically generate the corresponding methods when users specify only types of his relationships.

Fig.3.4(a)

```
setR: x
    (r isNil) ifFalse: [r remover': self] .
    r ← x.
    r setR': self

setR': x
    (r isNil) ifFalse: [r removeR': self].
    r ← x.

removeR: x
    (r = =x) ifTrue: [r removeR': self. r ← nil.]

removeR': x
    (r = =x) ifTrue: [r ← nil].
```

Fig.3.4(b)

```
setR: x
    r add: x.
    x setR': self.

setR': x
    r add: x.

removeR: x
    r remove: x.
    x removeR': self.

removeR': x
    r remove: x.
```

Fig. 4 Methods for maintaining 1-to-1 1-to-many, many-to-many relationship types.

4. Views

In database systems, views play an important role [9]. They provide the immunity of users' application programs to the change of a conceptual schema. Users can reorganize a conceptual schema to obtain their own schema by defining views. Realizing the view mechanism is very important in the OODB as well. Very little attention, however, has been paid to this topic.

In this section, we will discuss the realization mechanism of views in OODB, especially in the environment of Smalltalk. Since Smalltalk was designed as a programming language, it does not support the view mechanism of databases directly.

Therefore we take an approach to define a new class to manage all the views defined in the OODB.

In relational databases, the requirements for views are as follows:

(a) Users can handle views as if they were base relations.

(b) Only the definitions of views are stored in the system, so that the effects of update to the base relations are directly reflected on the views.

Later, we will see how our mechanism satisfies these requirements.

Basically, in relational databases, a view can be regarded as the answer of a query. So, a view definition can be represented by a query, and the relational algebra consists of basic operations to represent a query. In the OODB, subclass-superclass hierarchies are important construct of database schemata. So, we believe that the views in the OODB should basically have the following two functions:

(i) Creation of new (virtual) classes.

(ii) Reorganization of subclass-superclass hierarchies of conceptual schemata.

The function (i) is almost the same as in relational databases. The query in this case, however, is a Smalltalk program in general. So, we can define a wide variety of views in the OODB. This kind of view is called a virtual class in this paper. The function (ii) is unique to OODBs. This function, in fact, means to change all the classes, in which a user is not interested, into abstract classes (classes which cannot have their own instance objects) in the subclass-superclass hierar-

chy. We call the subclass-superclass hierarchies obtained in this way structural views. For the definition of structural views, the operations to change normal classes into abstract ones and vice versa are considered to be primitive operations. For this purpose, we will define two methods "union:" and "difference:" in the following.

In this section, basic methods to obtain all the instances or classes in a subclass-superclass hierarchy are given in Section 4.1. In Section 4.2 and 4.3, we will discuss virtual classes and structural views, respectively.

4.1 Basic Methods Used in View Definitions

First, we will define the following instance methods (of the class "Behavior") to obtain all the instances of a class (and of all its subclasses).

•allInstances	[return all instances of a class.]
•allSubInstances	[return all instances of a class and of its direct and indirect subclasses.]
•allSubInstancesList	[return all instances of a class and of its direct and indirect subclasses as a form of list structure.]

Similarly, the following methods obtain all the direct (and indirect) subclasses of a class.

•subclasses	[return all direct subclasses of a class.]
•allSubclasses	[return all (direct and indirect) subclasses of a class.]
•allSubclassesList	[return all (direct and indirect) subclasses of a class as a form of list structure.]

The methods "subclasses", "allSubclasses" and "allInstances" are Smalltalk built-in methods. Other methods can be easily defined. For example the method "allSubInstances" is implemented as follows:

```
allSubInstances
|temp1 temp2|
temp1←self allSubclasses.
temp2←self allInstances asOrderedCollection.
temp1 do:[:a|temp2←temp2 addAll: (a allInstances)].
↑temp2
```

Let us consider an example of University Faculty whose subclass-superclass hierarchy is shown in Fig. 5. Given each of the above defined messages, the class "Faculty" returns the following answer:

subclasses	(EngineeringFaculty LinguisticFaculty)
allSubclasses	(EngineeringFaculty EEFaculty CSFaculty LinguisticFaculty)
allSubclassList	(EngineeringFaculty (EEFaculty CSFaculty) LinguisticFaculty)

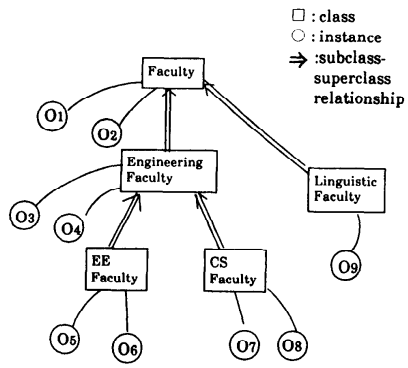


Fig. 5 A subclass-superclass hierarchy of university faculties.

allInstances	$(O_1\ O_2)$
allSubInstances	$(O_1\ O_2\ O_3\ O_4\ O_5\ O_6\ O_7\ O_8\ O_9)$
allSubInstancesList	$(O_1\ O_2(O_3\ O_4(O_5\ O_6)(O_7\ O_8))(O_9))$

4.2 Virtual Classes

We define a new class called "View" to manage all the views in OODB. The class "View" defined here is different from the built-in class "View" of MVC mechanism in Smalltalk-80. The important methods and variables defined in the class "View" are as follows:

```
instance variables: viewdef
class methods
  define: aSymbol as: aBlock
  |temp|
  temp←Smalltalk at: aSymbol put: View new.
  temp setview: aBlock

instance methods
  setview: aBlock
  viewdef←aBlock
  allInstances
  ↑viewdef value
```

For each view defined by users, an instance of the class "view" is created, and the instance variable "viewdef" stores the definition of that view as a Block.

For example, if we want to define a view "YoungFaculty" which consists of all the faculty members under 30 years old,

```
View define:#YoungFaculty
  as:[Faculty allSubInstances select:[:x|x age<30]]
```

is executed. This message has the following two effects.

(i) create a new instance of the class "View" and substitute it into the global variable YoungFaculty.

(ii) set the block defining this view in the instance variable "viewdef".

Once the view YoungFaculty is created, users can use it as if it were a class, except for defining methods and variables of its own. Therefore, if we want to know all

instances of YoungFaculty,

```
YoungFaculty allInstances
```

responds them.
As shown above, the function of the instance method "allInstances" of the class "View" is to evaluate the block in the instance variable "viewdef". Furthermore, since a view behaves like a class, we can define a view using other views. For example, if we want to define a view "YoungWomanFaculty" using the view "YoungFaculty", the following message expression is executed.

```
View define:#YoungWomanFaculty
  as:[YoungFaculty allInstances select:[:x|x sex=
  'female']]
```

As stated above, our implementation mechanism satisfies the requirement (a) for views. Since our mechanism keeps not the instantiated objects, but the definition of a view, it also satisfies the requirement (b).

4.3 Structural Views

Now, we will define instance methods (of the class "Class" or "Behavior") "union:" and "difference:" to reorganize a subclass-superclass hierarchy virtually. First we will give the definition of structural view and clarify the objectives and implication of these operations.

[Def 4.1] (Structural Views)
(1) A class is a structural view.
(2) If both sv_1 and sv_2 are structual views, (sv_1 union: sv_2) is a structural view.
(3) If both sv_1 and sv_2 are structural views, (sv_1 difference: sv_2) is a structural view.
(4) Only the objects obtained by (1), (2) and (3) are structual views.

As stated before, a structural view represents a virtual hierarchy in which some classes are abstract ones. For a given structural view sv, let GH(sv) denote the corresponding virtual hierarchy, and let normal (GH(sv)) denote the set of normal classes in GH(sv). Also, for a class C, let subtree(C) represent the C's subtree (i.e. the subtree of which root is C) in a virtual hierarchy.

[Def 4.2] (Virtual Hierarchies represented by Structural Views)
For a structural view sv, GH(sv) is defined as follows:
Case 1: (sv is a class C)
GH(C) is a virtual hierarchy in which all the classes in subtree(C) are normal classes and all the other classes are abstract ones.
Case 2: (sv is(sv_1 union:sv_2))
GH(sv) is a virtual hierarchy whose normal classes are the union of the normal classes in GH(sv_1) and GH(sv_2). (i.e. normal(GH(sv_1))$\cup$normal(GH(sv_2))).

Case 3: (sv is(sv_1 difference: sv_2))
GH(sv) is a virtual hierarchy whose normal classes are normal(GH(sv_1))-normal(GH(sv_2)).

In the University faculty example shown in Fig. 5, a structural view

(Faculty difference: EngineeringFaculty) (4-1)

represents the virtual hierarchy shown in Fig. 6(a). In this and the following figures, dotted rectangles represent abstract classes, and all the Smalltalk classes which do not appear in the figures are assumed to be abstract classes. Users of this structural view are only interested in the instance objects of all the faculty members other than engineering faculties. We cannot remove the abstract classes which are ancestors of the class "Faculty" in Fig. 6(a), because although these abstract classes cannot have their own instances, some instances in which users are interested (i.e. O_1, O_2 and O_9) may inherit variables or methods of them. Any other abstract classes could be removed, but we do not do this because they might "survive" later on. Let us assume that we want to start a new research project of automatic language translation, and that we want to define a view consisting of the objects of faculties related to this project. Also assume that all faculties in this university other than engineering faculties are related to the project, and that although CS faculties are engineering faculties, they are also related to the project. In other words, we want to define a structural view consisting of objects (O_1 O_2 O_7 O_8 O_9) in Fig. 5. This view can be defined as follows:

```
Faculty difference: EngineeringFaculty union:
CSFaculty                                    (4-2)
```

The corresponding virtual hierarchy is shown in Fig. 6(b).

Next we will show how structural views are implemented in Smalltalk environment. From the requirement (a) stated at the beginning of this section, views have to behave like a normal class. So, if we send a message "allInstances" to the view in (4-1) and (4-2), the response has to be a collection (O_1 O_2 O_9) and (O_1 O_2 O_7 O_8 O_9), respectively.

Sending the message "union:" or "difference:" to a class, an object of the class "View" is obtained as a response. This object behaves like a normal class as explained in Section 4.2. These two methods are implemented as follows.

```
union: aClass
|temp|
temp←View new.
temp setview: [self allSubInstances addAll:
                (aClass allSubInstances)].
↑temp

difference: aClass
```

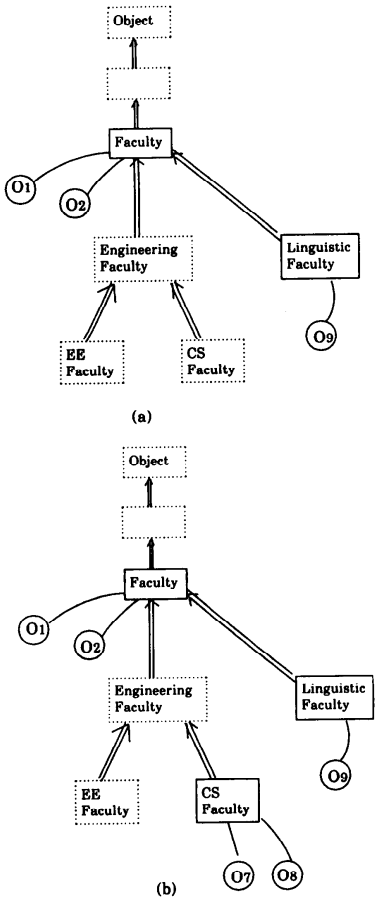


Fig. 6 Virtual hierarchies represented by structural views.

```
|temp|
temp←View new.
temp setview: [self allSubInstances
                removeAll:(aClass allSubInstances)].
↑temp
```

To allow the definition of a structural view using other views, we introduce an instance method "allSubInstances" whose definition is exactly same as "allInstances", and define the above two methods as instance methods of the class "View".

5. Incomplete Information

5.1 Incomplete Objects and Their Classification

In OODBs, several types of incomplete information are also represented as objects. Here, by "incomplete objects", we mean objects which convey incomplete information. We classify "incomplete objects" into the following three types:

(0) Null object

This corresponds to null values in conventional data model. There are two kinds of null objects: nonexisting objects and unknown objects.

(1) Unidentifiable object

For an object O, if neither the value of O's key nor the name of global variable referring to O is known, then the object O is said to be an *unidentifiable object*.

(2) Disjunctive object

For an object O, if there exist $n(\geqq 2)$ distinct objects $O_1, \cdots, O_n$ such that $O=O_1$ or $O=O_2$ or $\cdots$ or $O=O_n$, and $O=O_i$ is unknown for each i $(1 \leqq i \leqq n)$, then the object O is said to be a *disjunctive object*, and denoted by $O=O_1 \vee \cdots \vee O_n$.

(3) Unclassified object

Suppose that there exist a class C, its subclasses $C_1, \cdots, C_n$ and an instance object O of the class C (see Fig. 7). From the viewpoint of incomplete information, this situation can be semantically interpreted as follows: Although it is known that the "category" of the object O is C, it is unknown whether the "subcategory" of O is C_1 or C_2 or $\cdots$ or C_n or others. Under this interpretation of the sub-superclass hierarchy, the object O can be regarded as an incomplete object, and it is further classified into the following three types:

(3-a) It is known that the instance object O does not belong to any subclass $C_i(1 \leqq i \leqq n)$, and there is no adequate subclass, to which O should be classified, in the schema.

(3-b) It is known that the instance object O belongs to either C_1 or $\cdots$ or $C_n(n \geqq 2)$, but it is unknown to which C_i it belongs.

(3-c) It is known that the instance object O is of type (3-a) or type (3-b), but it is not known of which type it is.

Since each subclass itself is also an object, in the case of the above (3-a), the subclass, which O belongs to, is regarded as a unidentifiable object. In the case of the above (3-b), the subclass, to which O belongs to, is regarded as a disjunctive object $C_1 \vee \cdots \vee C_n$.

5.2 Realization of Incomplete Objects

(0) Null Objects

In Smalltalk, a special object 'nil' is provided. To represent a null object, we just use this 'nil' object.

(1) Unidentifiable Objects

It is straightforward to realize unidentifiable objects, that is, by the use of instance variable. Suppose that a person Smith has a child whose age is 3, and the child's

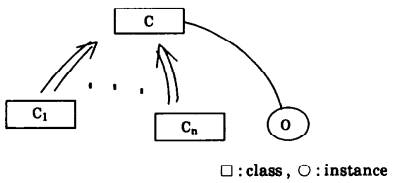


Fig. 7 Unclassified Object.

name is not known. Then in order to store this incomplete information, it suffices only to execute the following:

Smith child: (Person new age: 3)

Here, child: and age: are instance methods of class Person, where child:x stores the value x to the instance variable child, and age:y stores the value y to the instance variable age. As a result of this expression, a new person object (denoted "a Person" in Smalltalk), whose instance variable age has 3 as its value, is created and stored as a value of the instance variable child of Smith object. Also, suppose that child and age are instance methods of class Person, which return the value of instance variables child and age, respectively. Then,

Smith child

will return a Person (a Person object), and

Smith child age

will return 3 as its answer.

(2) Disjunctive objects

Basic requirements for manipulating disjunctive objects are as follows:

(i) Disjunctive objects should be able to be stored as the values of instance variables.

(ii) Any message to a disjunctive object should be propagated to each of its components.

(iii) When a disjunctive object is returned as an answer for a request, the information that the object is of disjunctive type should be explicitly returned.

For example, suppose that an employee object Bill is working at the sales department or research & development, where the sales department's manager is Mary and the research & development department's manager is John. Then, Bill's department is represented as a disjunctive object Sales $\vee R\&D$. Then, according to (i), the disjunctive object Sales $\vee R\&D$ should be stored as the value of Bill's instance variable "department". As for (ii), when a request to ask Bill's manager, then this message should be propagated to both of the Sales object and the *R&D* object. Further, according (iii), the answer for this request should be "Mary ∨ John". Here, we show two methods to realize the requirements.

[Method 1]

As shown in Fig. 8, we define the class Or, which is a subclass of OrderedCollection class. The OrderedCollection class is a built-in class of Smalltalk, each of whose instance is an ordered set of objects. Also, the OrderedCollection class is a descendent of the built-in class Object in Smalltalk. Each instance of this Or class is used to represent one disjunctive object. In order to make each instance of Or class behave as a disjunctive object, first, we define an instance method or: for the class Object as follows:

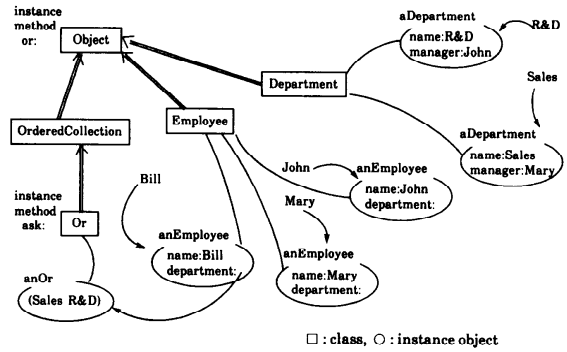


Fig. 8 Realization of Disjunctive Objects (Method 1).

or: *x*
"Create an Or instance whose elements are the receiver and the argument x."
|temp|
temp←Or new.[Creation of a new Or instance.]
temp add: self. [Add the receiver object to the Or instance as its element.]
temp add: x. [Add the argument object x to the Or instance as its element.]
↑temp

To make our idea work well also for disjunctive objects of more than three objects, we need to overwrite this instance method or: in the class Or in the following manner:

or: *x*
"Add the argument object x to the receiver Or instance as its element."
self add: x.
↑self

Using the or: method, it is possible to create a disjunctive object. For example, to store the "Sales ∨ *R&D*" as the value of Bill's department (instance variable), we execute the following:

Bill department: (Sales or: *R&D*)

When an arbitrary message is given to a disjunctive object O, it is necessary to broadcast it to each element of O. To do this, we define an instance method ask: in the Or class as follows:

ask: aSymbol
|temp1 temp2|
temp1←self size. [Store the number of elements of the receiver to temp1.]
temp2←Or new. [Create a new Or instance which is to convey the answer.]
(temp1=0)
ifFalse:[1 to: temp1
do:[:i|temp2 add: ((self at:i) perform: aSymbol)]].
[Store the answer to temp2. Here,

"self at:*i*" returns one element of the disjunctive object. Then, "perform: aSymbol" is sent to each of these elements, where perform: *x* is a built-in method to send the message *x*.]

↑temp2 [Return the answer.]

Using the ask: method, for example,

Bill department ask:#name

will return

Or(Sales *R&D*)

as its answer. Here we assume that "name" is an instance method of Department class and that it returns a department name. Also,

Bill department ask:#manager

will return

Or(an Employee an Employee),

and

(Bill department ask:#manager) ask:#name

will return

Or(Mary John).

Intuitively, an Or instance is a list of pointers, each of which points to an object, (i.e. a component of a disjunctive object). A disadvantage of this method is that we must use the above special message ask: for disjunctive objects.

[Method 2]

This method does not use any new class for disjunctive objects. Instead, we assume that each class has an instance variable 'or', and we define the following instance method or:
for the class Object (see Fig. 9).

or: *x*
|temp|
temp←self copy. [Make a copy of the receiver object and store it into temp.]
temp setOr: *x*. [Store *x* as the value of the instance variable 'or' of the object temp.]
↑temp [Return temp as its answer.]

If *A* is an instance of class *C*, then (*A* or: *B*) denotes an instance object of class *C* and its instance variable 'or' has *B* as its value. Each message sent to the object (*A* or: *B*) is automatically propagated to the object *B*. For example,

(Sales or:*R&D*) manager name

will return

Or(Mary John).

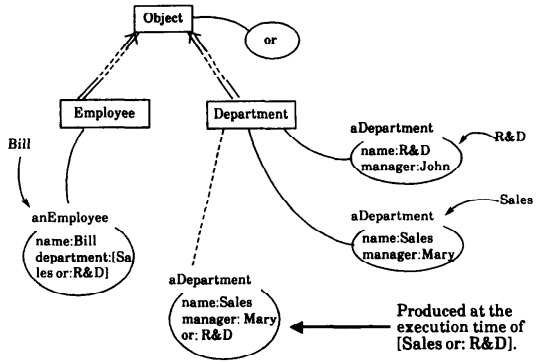


Fig. 9 Realization of Disjunctive Objects (Method 2).

Next, in order to store the information "Sales ∨ *R&D*" into the Bill object, we execute

Bill department:[Sales or: *R&D*].

where department: is an instance method to store a value to the instance variable department. Here the square brackets denote a "block" in Smalltalk, whose execution is postponed until the "value" message is sent to it. To retrieve the names of Bill's departments, we need to execute the following:

Bill department value name.

Using the block [*A* or: *B*] to store disjunctive objects, we need not propagate any updates performed for *A* and/or *B* to the disjunctive object because of the delayed execution mechanism of blocks. Of course, the major disadvantage of this method is that users must explicitly send "value" message to disjunctive objects.

(3) Unclassified Objects

In Fig. 5.1, when a classified object *O* is of type (3-a), it is necessary to define a subclass whose name is not known, and to make *O* be an instance of this subclass. In Smalltalk, it is not easy to define a subclass whose name is unknown. Also, whenever an instance of class *C* is to be added, we need some mechanism to check whether or not the instance belongs to conventional subcategories $C_1, \cdots, C_n$.

In the case of type (3-b), it is necessary to define a new class "C_1 or C_2 or$\cdots$or C_n" to which unclassified objects of type (3-b) belong. The class "C_1 or C_2 or$\cdots$or C_n" is a subclass of *C*. Also, we need some mechanism to propagate class variables of each C_i to the subclass "C_1 or C_2 or$\cdots$or C_n". That is, if each subclass contains a common class variable *V* and its value $v_i (i=1, \cdots, n)$ for each class C_i, then each instance of "C_1 or C_2 or$\cdots$or C_n" class must be able to refer to the class variable *V* and to obtain the value "v_1 or v_2 or$\cdots$or v_n".

6. Concluding Remarks

In this paper, we discussed schema design, update

propagation, view implementation and the treatment of incomplete information in the OODB. Also, we showed some methods to realize these issues by Smalltalk. Especially for the schema design, we introduced the notion of ODS (Object Database Schema) and the transformation of an ER schema into an ODS. As for views, we introduced the notions of 'virtual class' and 'structural view' for the OODBs, and identified basic operations to define these views. Also, we discussed a method to realize these views in Smalltalk. Finally, we introduced the notion of incomplete objects, which are objects conveying some incomplete information. The classification of the incomplete objects and their realization methods are discussed.

Since this paper is a starting paper to discuss several problems of schema design, views and incomplete information in the OODB, the following problems need further research.

(a) Identification of required basic operations and the criteria about the representation power of query languages for the OODB.

(b) A more general methodology for designing "methods" needed for the OODB.

(c) Design criteria for ODSs, which will be useful to discuss what design of OODB is good.

(d) Identification of what is lacking in Smalltalk to realize the OODB and what is needed for the extension of Smalltalk language.

References

1. ZANIOLO, C., AIT-KACI, H., BEECH, D., CAMMARATA, S., KERSCHBERG, L. and MAIER, D. Object Oriented Database Systems and Knowledge Systems, *In Expert Database Systems, Kerschberg*, L. Ed. *Benjamin/Cummings*, Menlo Park (1986), 49–65.

2. COPELAND, G. and MAIER, D. Making Smalltalk a Database System, *Proc. ACM-SIGMOD International Conference on Management of Data* (June, 1984), 316–325.

3. DITTRICH, K. and DAYAL, U. (ed.), Proceedings of the 1986 International Workshop on Object-Oriented Databse Systems (Sept, 1986).

4. GOLDBERG, A. and ROBSON, D. Smalltalk-80: The Language and its Implementation, Addison Wesley, Reading, Mass. (1983).

5. CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E. and SHEKITA, E. J. Object and File Management in the Extensible Database System, *Proc. 12th International Conference on VLDB* (August, 1986), 91–100.

6. MANOLA, F. and ORENSTEIN, J. A. Toward a General Purpose Spatial Data Model for an Object-Oriented DBMS, *Proc. 12th International Conference on VLDB*, (August, 1986), 328–335.

7. CHEN, P. P. S. The Entity-Relationship Model-Toward a Unified View of Data, *ACMTODS*, **1** (1) (March, 1976), 9–36.

8. HULL, R. and KING, R. Semantic Detabase Modeling: Survey, Applications, and Research Inssues, *Technical Report, Computer Science Department*, University of Southern California, TR-86-201 (Apr. 1986).

9. CHAMBERLIN, D. D., GRAY, J. N. and TRAIGER, I. L. Views, Authorization, and Locking in a Relational Database System, *Proc. AFIPS National Computer Conference* (May, 1975), 425–430.

(Received January 11, 1988; revised January 25, 1989)