データベース・システム　72−12
(1989. 7. 20)

# オブジェクト指向データベースにおけるオブジェクト転送機能

# Object Migration Mechanisms to Support Updates in Object-Oriented Databases

モハメッド エルシャルケウイ　上林弥彦
**Mohamed El-Sharkawi　Yahiko Kambayashi**

九州大学工学部情報工学科
**Dept. of Computer Sci. and Communication Eng.**
**Faculty of Engineering**
**Kyushu University**
**Fukuoka, Japan**

## あらまし

本論文では、オブジェクト指向データベースでの更新について述べる。オブジェクト指向データモデルでは、オブジェクトの更新によるクラスラティス上のオブジェクトの移動(オブジェクト転送)を考えなければならない。オブジェクトに対して3種類の更新がある:(1)インスタンス変数の追加、(2) インスタンス変数の除去、(3) インスタンス変数の値の変更。オブジェクトの更新により引き起こるオブジェクト転送は、インスタンス変数の定義、その定義域の定義および上記の更新の種類により決定する。データベーススキーマにおいて、オブジェクト転送の性質に基づいてクラスを次の3つに分類できる:(1)静的、(2)部分的動的、(3)動的。

本論文では、オブジェクト転送を考慮した更新の機構について述べる。また、オブジェクト指向データベースにおいて時間の変遷に伴う、オブジェクトの変遷をも管理する問題についても検討する。あるオブジェクトバージョンのクラスを決定するための簡単で、かつ効率の良い手続きも示す。

## Abstract

In this paper, we study updates in object-oriented databases. Due to the richness of the object-oriented data model, update to an object may affect the object's position in the class lattice. We consider three types of updates on objects: (1) Adding instance variables, (2) Dropping instance variables, and (3) Modifying instance variables. An update may cause object migration. That is, the updated object may change its current class. Object migration depends on the definition of instance variables and their domains and type of update. Classes in a database schema are classified into three types; static, partially dynamic, and dynamic. This classification is based on the way in which an object in a class may migrate. Actions that should be taken by the system to complete the update are presented. Importance of studying updates is demonstrated by specifying complexity in adding the time dimension to object-oriented databases. We give a simple, yet efficient, procedure to determine the class of an object version.

## 1- Introduction

Record oriented data models such as the relational data model are not adequate to new applications of database systems. CAD, OIS, and AI applications require semantically rich data models. Object-oriented data model is promising to be used in such applications. It has the following important features:
(1) Rich semantics, provide a more natural way to model the real-world. Entities in the real-world are objects in the model. Objects are grouped into classes, classes are organized in a hierarchy (lattice) representing IS-A (specialization) relationship between classes. A class inherits all properties of its superclass(es).
(2) Dynamic aspects of entities can be stored in the database through associating methods to classes.
(3) It supports a uniform language to write applications as well as data definition and manipulation statements. Programmers need not to know two different languages with different characteristics to write applications and to manipulate and define data in the database.
(4) Another advantage of the object-oriented model over the relational model is that, we can implement some functions of the DBMS more easier. For example, modification of a value of some instance variable may cause updates on other values. For example, changing total marks of a GRAD object may cause migration to class GOOD-GRAD, where some of its instance variables need to be modified. In the relational model such modification requires a trigger mechanism which is complicated and increases some overheads. In the object-oriented model such a modification can be done automatically be the update component using the method presented in this paper.

These promising features of the model initiated a lot of research efforts (see [1] for several researches). Several prototype database management systems based on the model have been built (e.g. GemStone [5], ORION[2]). In these systems updates are not discussed in deep. All updates are assumed to be not causing object migration. Updates that may cause object migration are done in two steps: fist, the object is deleted, second a new object is created such that it belongs to the required class. Our objective is to support such object migration automatically.

In this paper, we study updates on objects in object-oriented databases. Specifically, we study the effect of updating an object on its position in the class lattice. An update may affect the object such that it is no longer satisfies conditions of its current class. We consider three types of updates on objects: (1) Adding instance variables, (2) Dropping instance variables, and (3) Modifying instance variables. An update may cause object migration. That is, the updated object may change its current class. As the new class may be different from the class before the update, some actions have to be taken. Our objective is to provide a procedure to automatically perform the migration.

The idea is explained through an example. Consider the schema shown in Fig. 1. It models people in a university. The schema is represented by a rooted directed graph. Nodes represent classes in the schema. An edge from class $C_i$ to class $C_j$ means that $C_i$ IS-A $C_j$. For instance, TECH (technician) IS-A STAFF, TA IS-A STAFF, and TA IS-A STUDENT. The root of the graph is a system defined class called OBJECT. Suppose that instances of class GOOD-GRAD students are defined to be graduate students with total marks exceeding certain value. From this, an update that modifies this value may cause an object to migrate from class GRAD to class GOOD-GRAD. Such object migration has some side effects that should be handled by the system. If class GOOD-GRAD has some instance variables not defined in class GRAD, values of these instance variables have to be provided by a user, or considered null (which sometimes may not be possible).

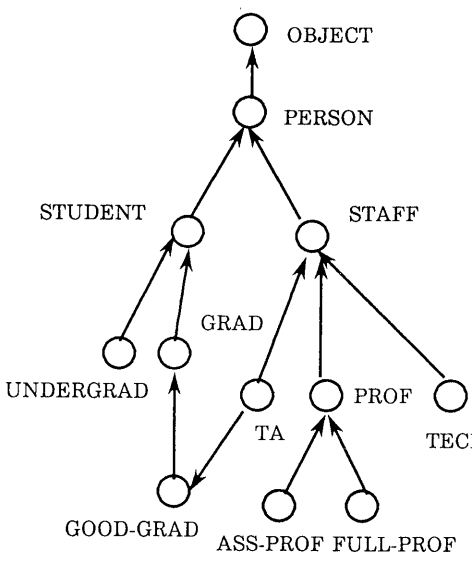

Fig.1 An example schema

We give classifications of classes in a database, with respect to how an object may migrate. This classification is useful for users to specify what kind of migration an object can do. For instance, a static class is a class in which all updates should not cause any object migration.

Now, we can see that, as an update causes object migration the system has to take some actions. With respect to each update causing migration, we give a group of actions should be taken by the system (or in cooperation with a user).

Work in [3] concentrates on schema evolution operations like adding and dropping a class, adding and deleting an edge between two classes, adding and dropping properties of a class, etc. In this paper, the objective is to study modifications on objects.

Another important requirement necessary to support new applications is adding the time dimension into databases. A database should store the real-world history as well as its current status. Extending the relational model with the time dimension has received a lot of research efforts [4]. It seems, however, that the issue is not studied extensively in context of object-oriented data model. Due to object migration, the situation in object-oriented databases is more complicated. Versions of an object's history may be distributed among several classes. That requires new mechanisms to answer queries accessing objects' histories. We give a simple mechanism to answer temporal queries. It avoids searching all objects in classes the queried object may be an instance of.

The paper is organized as follows. In the next section, basic concepts of the object-oriented data model are reviewed. In Section 3, we discuss updates in the model and define classes of database schema. Section 4 demonstrates importance of studying updates by discussing complexity and necessary requirements to add the time dimension into object-oriented databases. Section 5 concludes the paper.

## 2- Basic Concepts

### 2-1 Objects, Instance Variables, and Method

In object-oriented data model, entities in the real-world are considered as objects. Properties of an object are divided into two parts, its status and its behavior. Status of the object is captured through its instance variables. Object behavior is encapsulated in a set of methods associated with the object. A method is a code that manipulates the object's status. To manipulate an object a message should be sent to the object. Response to a message is done by executing a method corresponding to the message. Objects communicate via sending messages.

### 2-2 Classes

Objects having similar properties are grouped together to constitute a class. All objects belong to a class are its instances. All instances in the class have same instance variables and respond to same set of messages. To provide conceptual simplicity and efficiency at implementation level, classes have variables called class variables. Values of class variables are shared by all instances in the class, and called shared-value instance variables. For example, instances of class STUDENTS share value of the class variable UNIVERSITY.

The class may also contain some default values for some instance variables. These variables are called default-valued variables. Instances that have their default-valued instance variables are not defined take the values stated in the class. For example, if the SEX of a STAFF instance is not defined it takes the default value "Male".

### 2-3 Class Hierarchy and Inheritance

Classes in the system are organized in a class hierarchy. An edge between two classes represents IS-A relationship between the two classes. For example, an edge from class GRADUATE-STUDENT to class STUDENT means that GRADUATE-STUDENT IS-A STUDENT. For two classes with IS-A relationship the higher-level class in the hierarchy is a superclass of the lower-level class. The lower-level class is a subclass of the higher-level class. A class inherits all properties of its immediate superclass. It may have also its own properties. A class may have an instance variable with a name similar to one of its inherited instance variables, in this case the instance variable defined within the class overrides the inherited one.

### 2-4 Class Lattice

For data modeling it is necessary to extend the class hierarchy into a class lattice. A class may have several immediate superclasses and it inherits all of their properties. Name conflicts between superclasses of a class are resolved by distinguishing one of the superclasses as a first superclass. Then, a class inherits properties with conflict names from its first superclass. The class lattice, also the class hierarchy, is rooted such that there is no dangling nodes. The root node is a system defined class called OBJECT.

### 2-5 Domains of Instance Variables

An instance variable gets its possible values from instances of a class in the system. The class domain is either one of system defined basic classes or any other user defined class. Basic classes include INTEGER, REAL, CHAR, and BOOLEAN. An instance variable that gets its value from one of the basic classes is called a basic instance variable, otherwise it is called a complex instance variable.

Some instance variables may take any value in its domain, some others, however, have restricted range in which the value should exist. For example, instance variable SALARY of class MANAGER should satisfy the following condition SALARY $\geqq$ ¥1000000. The range $R_i$ of instance variable $IV_i$ is a predicate which may be undefined, a simple predicate, or simple predicates connected by logical AND/OR. A simple predicate has one of the following forms:
(1) $R_i \equiv IV_i$ (Operator) K,
(2) $R_i \equiv K_j \leqq IV_i \leqq K_u$, where K, $K_j$, and $K_u$ are elements of $D_i$ (the domain of $IV_i$), and Operator is one of the set $\{=, \neq, <, \leqq, >, \geqq\}$.

An object-oriented schema is represented by a directed rooted graph. Nodes in the graph correspond to classes in the schema. There is an edge from (node corresponds to) class $C_i$ to class $C_j$ means that $C_i$ IS-A $C_j$. The root of the graph is a system defined class called OBJECT.

## 3- Updates in Object-Oriented Databases

In this section, we study updates in object-oriented databases. Semantic richness of the object-oriented data model necessitates such study. Objects are grouped into classes, updating instance variables of an object may affect the position of the object in the class lattice. We study effects of three types of updates on objects' positions. These updates are done over instance variables: Adding some instance variables, Dropping some instance variables, and Modifying some instance variables. Updates may affect the object's position in the class lattice as follows:
(1) Adding instance variables: the object will migrate to one of the subclasses of its current class.
(2) Dropping instance variables: the object will migrate to an appropriate one of the superclasses of its current class.
(3) Modifying instance variables: the object may migrate to another class.

The next subsection gives classification of classes according to effects of updates on objects. Then, in the last subsection, we give actions that should be carried out automatically when an update causes object migration. These actions depend on the update type, we give also a procedure to do object migration.

### 3-1Classification of Classes in Object-Oriented Schema

Classes in an object-oriented schema are classified into three types. The classification is based on how objects in the class may migrate with respect to an update. These three types are static, partially-dynamic, and dynamic. Definitions of these classes are given in this subsection. Before that, we need the following definitions.
Definition. A class $C_i$ is "Contiguous to" class $C_j$, if there is an update that makes some objects in $C_i$ migrate to class $C_j$.
According to types of updates, "Contiguous to" relationship is classified into generalization/specialization contiguous to (written "GS-Contiguous to") and modification contiguous to (written "M-Contiguous to").
Definition. A class $C_i$ is "G-Contiguous to" class $C_j$, if an object in class $C_i$, due to updating some of its instance variables, migrates into class $C_j$, class $C_j$ is a superclass of $C_i$.

Definition. A class $C_i$ is "S-Contiguous to" class $C_j$, if an object in class $C_i$, due to updating some instance variables, migrates into class $C_j$, class $C_j$ is a subclass of $C_i$.
Definition. A class $C_i$ is "M-Contiguous to" class $C_j$, if an object in class $C_i$, due to modifying some of its instance variables, migrates into class $C_j$, class $C_j$ is neither a super nor subclass of $C_i$.
Note that:
1- When class $C_i$ is "M-Contiguous to" class $C_j$ via a set of instance variables, for each $IV_{ih}$, $h=a,...,m$, there should be an instance variable $IV_{jk}$, $k=b,...,n$, in $C_j$ such that:
1- they have same semantics, 2- defined over same domains, and 3- migrate compatible.
Migrate compatibility, is a condition to prevent meaningless object migrations. For instance, modifying a person's name does not cause migration from class PERSON to class COMPANY.
2- If class $C_i$ is "G-Contiguous to" class $C_j$, class $C_j$ is "S-Contiguous to" class $C_i$.
3- "M-Contiguous to" relationship is not necessarily symmetric. That is, if class $C_i$ is "M-Contiguous to" class $C_j$, it may not be true that $C_j$ is "M-Contiguous to" $C_i$. For example, class ASS-PROF is "M-Contiguous to" class PROF, however, in general, PROF is not "M-Contiguous to" ASS-PROF.

It is the case that some instance variables don't change their values even when the object migrates. For these instance variables, the system, automatically, has to use their values in the new class. For example, when an object of class ASS-PROF migrates to class PROF, value of its ADDRESS instance variable does not change (unless it is modified in the update). These instance variables are called fixed instance variables and defined as follows:
Definition. An instance variable is called fixed if its value does not change due to object migration.
Instance variables that are not fixed are called unfixed instance variables.
Note that:
(1) Such values could be updated by users, but not changed automatically due to migration.
(2) When an object migrates from class $C_i$ to $C_j$, values of its fixed instance variables don't change. However, the name of a fixed instance variable in class $C_i$ may be different from its name in class $C_j$. Corresponds between these names have to be defined at schema design.

Another issue that characterizes classes is a stabilization point of a class.
Definition. Class $C_j$ is the stabilization point of class $C_i$, if an object in $C_i$ migrates (via one or more updates) into class $C_j$ and then any update does not affect its position in $C_j$.
From this definition, the trivial G-stabilization point of class $C_i$ is the system defined class OBJECT, the nontrivial G-stabilization point is a class $C_j$ such that $C_j$ is a superclass of $C_i$ and OBJECT is the only superclass of $C_j$.
The S-stabilization point of class $C_i$ is a class $C_j$ such that $C_j$ is a subclass of $C_i$ and there is no class $C_k$ such that $C_k$ is a subclass of $C_j$.
Note that a class may have more than one stabilization point. If classes in a schema constitute a class hierarchy (not a lattice), any class has at most one nontrivial G-stabilization point. It may, however, have several S-stabilization points.

Now, classes are classified into three types, static, partially-dynamic, and dynamic. Definitions of these types are as follows.
Definition. A class $C_i$ is static if there is no class $C_j$ such that $C_i$ is "Contiguous to" $C_j$.
It means that, any update on a class instance does not cause object migration.
Definition. A class $C_i$ is partially-dynamic if there is no class $C_j$ such that $C_i$ is "M-Contiguous to" $C_j$.
That is, there is no update that causes an object migration to a class which is neither a super nor subclass of $C_i$.
Definition. A class $C_i$ is dynamic if all three "Contiguous to" relationships are defined.
That is, dynamic classes are the most general classes. A class may be "GS-Contiguous to" or "M-Contiguous to" some other classes.

These notions can be used in schema definition to instruct the system about what types of updates are applicable on classes. For example, an update on a partially-dynamic class that causes migration to another class and the last one is neither a super nor subclass of the first class, should be rejected automatically by the system. Object-oriented schema may be classified with respect to its classes types. That is, a schema is said to be static if all its classes are static, partially-dynamic if it contains only static and partially-dynamic classes. A dynamic schema contains at least one dynamic class.

## 3-2 Object Migration Procedures

In this subsection, we state actions that should be taken by the system as an update causes object migration. These actions depend on the update type. For each type of update, actions should be taken are given, and then we give procedure object migrator (OM).

**1- Dropping instance variables:**
(1-a) The system has to check that the dropped instance variables of object O in class $C_i$ are:
all (not a subset of) unfixed non-inherited instance variables.

For example, it is possible to drop instance variables of an object in class FULL-PROF to migrate into PROF, or all instance variables defined in class FULL-PROF and those inherited from class PROF and not from classes STAFF, PERSON, and OBJECT. The object will migrate into class STAFF.
We have the following three cases:
case 1-1: The dropped instance variables are non-inherited and there is a unique immediate superclass of $C_i$.
(1-b) the object migrates into its immediate superclass and if there is an instance variable $IV_i$ defined for class $C_i$ of which O is an instance and $IV_i$ overrides $IV_j$ of the superclass of $C_i$; after the migration the value of $IV_i$ should be replaced by value of $IV_j$ (the original instance variable). Same action is also necessary in case of overridden methods.
(1-c) Non-inherited methods should be dropped.

case 1-2: The dropped instance variables are non-inherited and the class has several immediate superclasses
(1-d) In this case, system has to consult the user in order to determine the new class. For example, after dropping non-inherited instance variables of an object in class TA, the system has to consult the user in order to determine if the object, after the update, belongs to class STAFF or GOOD-GRAD. The new class can be determined automatically if there is some integrity constraint that prevents the updated object to migrate into a specific superclass. For example, the TA object after the update is no longer a STAFF.

case 1-3: The dropped instance variables are non-inherited and those inherited from $C_n$, $C_n$ is an immediate superclass of $C_i$. The object will migrate to class $C_k$, where $C_k$ is the immediate superclass of $C_n$.

(1-e) Instance variables inherited from any class $C_h$, $C_h$ is a subclass of $C_n$ and a superclass of $C_i$, have to be dropped.

(1-f) If $C_i$ has other immediate superclass which is neither a super nor a subclass of $C_n$, instance variables inherited from this class has to be dropped when migrating to $C_k$. Values of instance variables inherited in $C_k$ should be provided or considered null.

The problem here, is that if $C_i$ has some fixed instance variables inherited from a superclass which is not in the path of $C_j$. The user has to be informed with such case.

## 2- Adding instance variables:

(2-a) The system has to ensure that the added instance variables are one of those defined in one of the subclasses of the current class. Otherwise, the update is not correct.

(2-b) Value of any instance variable or method which will be overridden in the new class has to be modified.

(2-c) When instance variables are added to object in class $C_i$ and the new class is $C_j$ such that $C_j$ has several superclasses, values of all instance variables defined in this set (except those of $C_i$) have to be provided or assumed to be null valued.

## 3- Modification of instance variables

There are the following four cases:

case 3-1: The object's class before modification ($C_i$) and the new one ($C_j$) have same immediate superclass. In this case the following actions has to be taken:

(3-1-a) Unfixed non-inherited properties in class $C_i$ have to be dropped from the object.

(3-1-b) Non-inherited instance variables defined in class $C_j$ have to be added to the object after the update and their values should be provided or considered to be null valued.

(3-1-c) All inherited properties that are overridden in class $C_i$ ($C_j$) and not so in class $C_j$ ($C_i$) have to be modified in the object after the update.

(3-1-d) If $C_j$ has superclasses different from those of $C_i$, values of instance variables that are inherited from these classes should be provided or considered to be null valued.

case 3-2: The object's class before modification ($C_i$) and the new one ($C_j$) have different immediate superclass.

(3-2-a) All unfixed properties defined in class $C_i$ will be dropped from the new object.

(3-2-b) All properties defined in class $C_j$ become properties of the new object.

case 3-3: New class $C_j$ is a superclass of the original class $C_i$. In this case, there are two possibilities, $C_i$ has only one immediate superclass $C_j$ and $C_i$ has several immediate superclasses.

In the first case, there are two situations; 1-when $C_j$ is the unique immediate superclass of $C_i$, the following actions have to be done:

(3-3-a) Unfixed non-inherited properties have to be dropped from the object after update.

(3-3-b) Overridden inherited properties defined in class $C_i$ have to be modified in the new object.

the second situation when: 2- $C_j$ is not the immediate superclass of $C_i$, actions (3-3-a) and (3-3-b) have to be done in addition to the following actions:

(3-3-c) Properties inherited from any class $C_k$ such that $C_k$ is a subclass (and superclass) of $C_j$ ($C_i$) has to be dropped.

(3-3-d) If $C_i$ has immediate superclass $C_k$ which is not a subclass of $C_j$, properties inherited from this class has to be dropped. Same problem of fixed instance variables inherited from a superclass which is not in the path of $C_j$ will arise.

In the second case, when $C_i$ has several immediate superclasses,:

(3-3-e) All unfixed non-inherited properties defined in $C_i$ and overridden properties originally defined by a superclass $C_k$ ($C_k \neq C_j$) have to be dropped from the object after update.

(3-3-f) Values of instance variables defined in class $C_j$ and overridden in class $C_i$ have to be modified in the new object.

case 3-4: New class $C_j$ is a subclass of the original class $C_i$. There are two situations:

case 3-4-1: there is no class $C_k$ such that $C_k$ is a superclass of $C_j$, the following action will be taken:

(3-4-a) Properties defined in class $C_j$ are added to the object and values of overridden inherited properties should be modified.

case 3-4-2: $C_j$ has superclasses other than $C_i$.

(3-4-b) is similar to (3-4-a).

(3-4-c) Properties inherited will be defined and values of inherited instance variables from these classes should be provided or considered to be null valued.

(3-4-d)All overridden inherited properties from superclasses other than $C_i$ have to be defined in the new object.

Now, we give a procedure, called Object Migrator, OM, to perform updates and their side effects on the object's position. We need to define a directed graph called migration graph.

Definition: A migration graph (V, E, L) is a directed labeled graph. V is the set of nodes. Each node corresponds to a user defined class in the schema and rooted with the system defined class OBJECT. E is the set of edges in the graph. Each edge has a label, and L is the set of labels. There is an edge from class $C_i$ to class $C_j$, if some update on object in $C_i$ will cause object migration to class $C_j$. The set E is union of three sets, GE, SE, and ME. Generalization edge (GE) corresponds to deletion of instance variables. Specialization edge (SE) corresponds to addition of instance variables. Modification edge (ME) corresponds to modification of instance variables. A GE (SE) edge from $C_i$ to $C_j$ is labeled with instance variables to be deleted (added) from an object in $C_i$ to move into $C_j$. An ME edge from $C_i$ to $C_j$ is labeled with $\alpha$. $\alpha$ consists of instance variables that cause the migration. Each instance variable is accompanied with the range predicate to be satisfied.

The migration graph of the example schema is shown in Fig. 2 (for clarity labels are omitted).

Procedure OM is as follows:

**PROCEDURE OM**

**INPUT:** An update has one of the following three forms:

ADD(O,$IV_q$), DROP(O,$IV_q$), MODIFY(O,$IV_q$), where, O is the object to be updated and $IV_q$ are updated instance variables.

**METHOD:**

```
Case update type:
 if DROP(O,$IV_q$)$\Rightarrow$
```

```
        Compare labels of GE edges originating
        from the class of object O with $IV_q$;
            if $IV_q$ don't match any label then send an
            error message;
            else determine the new class and do
            actions (1-a) to (1-f) ;
    if ADD(O,$IV_q$)⇒
        Compare labels of SE edges originating
        from the class of object O with $IV_q$;
            if $IV_q$ don't match any label then send an
            error message;
            else determine the new class and do
            actions (2-a) to (2-c) ;
    if MODIFY(O,$IV_q$)⇒
        Compare labels of ME edges originating
        from the class of object O with $IV_q$;
            if $IV_q$ don't match any label then send an
            error message;
            else determine the new class and do
            actions :
                (3-1-a) to (3-1-d)      or
                (3-2-a) and (3-2-b)     or
                (3-3-a) and (3-3-b)     or
                (3-3-a) to (3-3-d)      or
                (3-3-e) and (3-3-f)     or
                (3-4-a)                 or
                (3-4-b) to (3-4-d);
END OM;
```

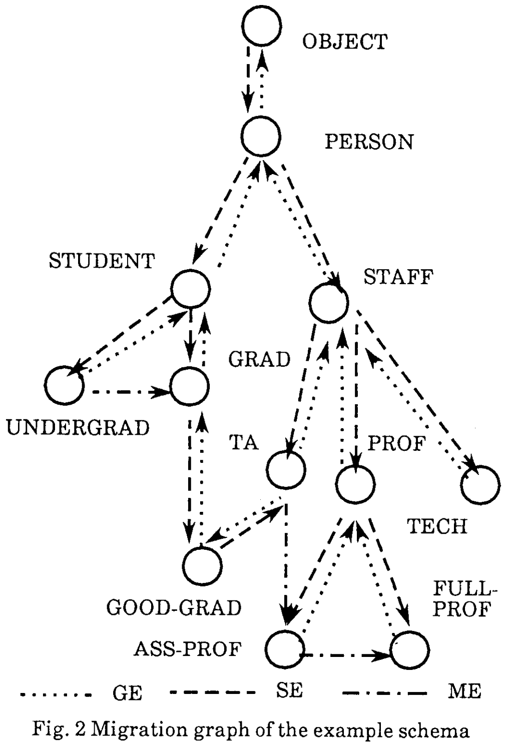

Fig. 2 Migration graph of the example schema

Notes:

In the previous discussions, we concentrate on updates on the object, itself, that causes migration. It is also the case that an update to an object may cause another object to migrate. This is caused by updating an object that corresponds to a complex instance variable of another object.

Effects of schema evolution operations, discussed in [3], on instances in the database may be considered as objects migration and handled in this framework.

Other operations like SPLIT of an object into several objects or MERGE of several objects to constitute one big object can also be handled in this framework.

We can use this framework to provide users with helping instructions to supply values of some instance variables in order to complete the update. Usually, the user may not be aware of the side effects of the update. It can, also, be used to check if an update matches the class type.

Procedure OM is a generalization of some similar problems. For instance, in a CAD system, the database is divided into private, group, and public databases. A design object may migrate from private database into group database, and then to public database. For example, if logic circuit simulation has been completed the design object moves into group database.

## 4- Temporal Object-Oriented Databases

Some applications require adding the time dimension to databases in order to access the history, not only the current status, of the real-world. Such applications include medical information systems and office information systems. Several research efforts have been devoted to study addition of the time dimension into relational databases ([4]). It seems that adding the time dimension to object-oriented databases have not been extensively studied. The semantic richness of the object-oriented data model makes it inappropriate to use techniques and methods applied to create temporal relational databases in creating temporal object-oriented databases. In temporal databases, object's history has to be stored, users may access that history. Each version in the object's history has same object identity. Enhancing the performance of query processing is an important issue to implement temporal databases. As seen from the previous section, updating an object may cause object migration, thus, versions of the object's history may belong to different classes. This complicates enhancement of system performance. In this section, we give a simple, yet efficient, approach to improve system response. The problem arises from the fact that an object's history is distributed among several classes in the schema.

From the previous analysis, we notice that classes of versions in an object's history can be specified by the set of classes between the class of the initial version and all of its stabilization points. The very intuitive approach to answer a user's query is to search the set of all objects in the initial version of the queried object, if it is not found search one of the next contiguous classes. This process continuous until the required object is found, or cannot find it after visiting all stabilization points of its class a message is sent to the user that the object cannot be found. This approach is very much time consuming. If the object has been updated several times, it is necessary to search instances of several classes, in some cases the search is unnecessarily performed. In this section, we present a simple procedure and a data structure used to improve query response.

The data structure we use is a table called UPDATES. It contains information about all updates performed on the objects in the database. It has the following attributes: Update#, OBJECT, OBJECT-CLASS, TIME, and DEL-INDICATOR. Update# is a unique identifier for each update. OBJECT is the object that is updated, OBJECT-CLASS is its class after the update. TIME is the time at which the update has occurred. DEL-INDICATOR is used to mark updates that are deletion of objects. All updates over an object may be grouped together and sorted chronologically. This table can be implemented as a meta-class and some methods can be defined to get some information about the real-world evolution.

This table is used to answer queries. Given a user's query that accesses an object O at specific point of time T. The table is consulted to determine the class to which the version of the object, created at time less than or equal to T, belongs. When the class is recognized, only instances of this class should be searched to find object O. The following procedure process such a query.

```
PROCEDURE ANSWER-TEMPORAL-QUERIES
INPUT: A query has the form GET(O,T), where O is
       an object and T is a specific point of time.
METHOD: 1- Check that there is an update over O in
          UPDATES, if there is no such an update, O
          belongs to its initial class $C_i$. Search instances
          of $C_i$ for O.
        2- Check that object O has not been deleted
          before time T. This is done by looking for a
          tuple in the O's set of tuples with DEL-
          INDICATOR = 1 and TIME $\leqq$ T.
        3- Find the update $u_f$ performed on O such that
          $u_f$(TIME) = T or $u_f$(TIME) < T and it is the
          maximum among time of all updates performed
          on O before time T.
        4- The class of object O at time T is the value in
          OBJECT-CLASS attribute of update $u_f$.
        5- Search instances of class $u_f$(CLASS) for
          object O updated at time $u_f$(TIME).
        6- Return result.
        7- END ANSWER-TEMPORAL-QUERIES
```

This procedure can, very easily, be extended to answer queries accessing the object's history in some period of time or accessing the whole object's history.

In this method, we introduced the data structure UPDATES into the system. We can handle the problem without adding such a data structure, keeping the system homogeneous. Since we know for each class the set of its stabilization points, a set of instance variables correspond to stabilization points can be defined. These instance variables are not inherited by any subclass of the class. They are instance variables defined, only, for instances in this class. The approach is as follows.

(1) Associate with each class a set n (n is the number of classes in the path from this class to each stabilization point) of system defined instance variables called CLASS-HISTORY.
(2) Each CLASS-HISTORY is an ordered pair (Class-Name, Start, End), where Class-Name is name of a class in n, Start is the time at which a version of the object created in this class, End is time at which the version in Class-Name is updated to create a version in another class in n.
(3) At the beginning an object has no values for CLASS-HISTORY instance variables except for the initial class. When an update that causes migration occurs on an object, it modifies CLASS-HISTORY in its initial class. This is done by adding the new class name, and initiating its Start to be the update time. End of the previous class is set to be the update time.
(4) To answer a query Q(O,$T_q$), the initial class of O is visited and values of CLASS-HISTORY are checked to find the period in which $T_q$ lies. When this period is found, the version's class of O is identified.

Note that:
(1) For some classes in the stabilization points of a class, their may be more than one period. This due to cycles in the migration graph.
(2) If we use this approach, initial class of an object should be visited each time the object migrates. This can be done as an action automatically carried out as a result of object migration. Procedure OM can be modified to implement this action.

Another problem which is specific to object-oriented databases is that an update, which is not applied on an object, may create a version of this object. There are two cases of such type of updates.

(1) Updating a shared-value and/or a default-valued instance variable in a class requires creating new versions of all instances of the class, when updating a shared-value instance variable and/or all instances that consider the default value of a default-valued instance variable.
(2) Update to an object O' may create two versions, one of O', and the other of object O that has O' as one of its (complex) instance variables.
These cases are handled as follows:

(1) It is not efficient to create versions of all objects affected by updating shared-value and default-valued instance variables. Instead, we associate with the class an array that stores updates of these instance variables and time of update. Versions of objects are only created when a query is submitted that requires this creation. Synchronization process is necessary to determine temporally matching default or shared-valued versions.
Example: Suppose that there is a shared-value instance variable UNIVERSITY = "Kyushu Institute" is defined in class STAFF. This value is propagated to instances of all subclasses of STAFF. Then the UNIVERSITY value is updated twice. At time $t_1$ the value changed to be "Kyushu Imperial University" and at time $t_2$ changed to be "Kyushu University". A query to get the UNIVERSITY value of Prof. Yamada at time $t_i$ is submitted. The answer is given by synchronizing $t_i$ with one of the periods 0-$t_1$, $t_1$-$t_2$, and $t_2$-NOW to determine the appropriate value of UNIVERSITY.

(2) Suppose that a query is submitted that accesses version of object O at time $T_q$. O has object O' as one of its instance variables. This query is splitted into two queries. One is accessing the version of object O at time $T_q$ and the other is accessing the version of object O' at the same time. If the query is accessing versions of O at some period of time $T_{q_1}$ to $T_{q_2}$ or the whole history of O, time synchronization of versions of O with versions of O' is necessary. Note that, if O' has some complex instance variable the query has to be splitted into three queries. In general, if this nesting level is n the original query should be splitted into n queries each accessing versions of one of the nested objects. Synchronization between the n histories has to be done.

Another issue we need to discuss is application of methods on objects' histories. As discussed in previous section, a method which is applicable on an object, may become no longer applicable after updating the object. In temporal databases, if a method is applied to an object's version at a specific point of time, the method may not be applicable and an error message is sent to the user. If, however, the method is applied to an object's versions at some period of time, some versions may execute the method and some others may not accept it since it is not defined. Response to the method may be different in two versions of an object due to name overriding. The user has to be acknowledged of such situations.

There are two more issues we need to point out. First, the type of time (either transaction time or valid time [7]) is immaterial in the above discussion. Transaction time is the time at which information was stored in the database. Valid time is the time at which the real-world was changed. Another notion pointed out in [7] is that some category of temporal databases permits updating the past history. In context of object-oriented data model, updating the past of an object may have several side effects. For example, it may be necessary to reexecute all changes done after the modified update, which may cause changes in the object's current position in the class lattice. Second issue we need to point out is that schema modifications as discussed in [3] affect objects' histories. For example, as a result of creating a new class some objects in an existing class may migrate to the new one, creating versions in their histories. Effects of schema modifications on object migration are left for future detailed work on temporal object-oriented databases.

From the above discussions, it is observed that the simple the schema type, the few and simple requirements are necessary. For example, in static schema all versions of an object's history belong to the initial object's class.

## 5- Conclusion

The object-oriented data model has several promising features that make it suitable for new and nonstandard applications. Due to its semantic richness, it need to be extensively studied. In this paper, we studied updates in such data model. An update may affect the position of an object in the class lattice. These effects have been studied and classes in an object-oriented schema are classified according to object migration. Functions required from the DBMS when an update occurs are studied in each class of schema. Next, importance of our study is demonstrated by studying addition of the time dimension to object-oriented databases. We, also, give a simple, yet efficient, procedure to answer queries in TOODBs. For future research, we think that this work is a first step to study several issues of object-oriented databases. Schema design, TOODBs, and concurrency control are among these issues. For TOODBs, we need to consider schema evolution operations as well as updates.

## Acknowledgement

We would like to thank Mr. M. Arikawa and Mrs. H. Xu for helpful disscussions.

## References

[1] ACM Transactions on Office Information Systems, Vol.5, No.1, Jan. 1987.

[2] Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications", in [1], pp. 3-26.

[3] Banerjee, J., et al. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proc. ACMSIGMOD, 1987, pp 311-322.

[4] Mckenzi, E. "Bibliography: Temporal Databases," ACM SIGMOD Record, Vol. 15, No. 4, Dec. 1986, pp. 40-52.

[5] Maier, D., et al. "Development of an Object-Oriented DBMS", Proc. OOPSLA, 1986, pp. 472-482.

[6] Stefik, M.; Bobrow, D.G. "Object-Oriented Programming: Themes and Variations," AI Magazine, Jan. 1986, pp. 40-62.

[7] Snodgrass, R; Ahn, I. "A Taxonomy of Time in Databases," Proc. of ACMSIGMOD, 1985, pp.236-246.