| | |
|---|---|
| Title | Towards Temporal Object-Oriented Databases |
| Author(s) | El-Sharkawi, Mohamed; Kambayashi, Yahiko |
| Citation | 数理解析研究所講究録 (1990), 731: 262-273 |
| Issue Date | 1990-10 |
| URL | http://hdl.handle.net/2433/101960 |
| Right | |
| Type | Departmental Bulletin Paper |
| Textversion | publisher |

数理解析研究所講究録
731 巻 1990 年 262-273



# Towards Temporal Object-Oriented Databases

**Mohamed El-Sharkawi Yahiko Kambayashi**
**Dept. of Computer Sci. and Communication Eng.**
**Kyushu University**
**Higashi, Hakozaki, Japan**

**Abstract**

In this paper, we study problems in adding the time dimension into the object-oriented data model. Solutions are given for the following problems: (1) Problems due to object migration. In the object-oriented model an update to some instance variables of an object may enforce the object to "migrate" to a new class. Due to this migration an object's history is distributed among several classes. (2) Problems due to the rich semantics of the model, specifically problems in creating versions of shared and default valued instance variables and problems associated with complex instance variables. (3) Problems due to schema evolution. When the schema changes, we have to keep the old schema in order to answer temporal queries. (4) Problems due to updating the history. In historical and temporal databases, it is possible to correct the history information when some errors have been discovered. Updating the history may affect updates that have been done after the corrected updates have been taking place.

## 1- Introduction

Advanced database applications like computer-aided design (CAD) and office information systems (OIS) have several requirements that are not met by existing record oriented data models. These applications require more semantically rich data models. The object-oriented data model has promising features that make it suitable for these new applications. It has the following important features:
(1) Its 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.

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. Users may access the history data as well as current data. 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 the context of the object-oriented data model.

The focus of our paper is studying and solving problems in adding the time dimension to the object-oriented data model. Due to the rich semantics of the

model, approaches used in extending the relational model with time are not applicable.

Solutions are given for the following problems:
(1) Problems due to object migration. In the object-oriented model an update to some instance variables of an object may enforce the object to migrate to a new class. Due to this migration the object's history is distributed among several classes. Given a query, we need to find an efficient way to determine the class of the queried object. (2) Problems due to the rich semantics of the model, specifically problems in creating versions of shared and default valued instance variables and problems associated with complex instance variables. (3) Problems due to schema evolution. When the schema changes as described in [BKKK], we have to keep the old schema in order to answer temporal queries. (4) Problems due to updating the history. In historical and temporal databases, it is possible to correct the history information when some errors have been discovered. Updating the history may affect updates that have been done after the corrected updates was done.

## 2- Basic Concepts

### 2-1 The Object-Oriented Data Model

We assume that there are the following sets:
(1) A finite set of classes called Basic-Classes. This set includes primitive, system-defined classes INTEGER, REAL, STRING, and BOOLEAN. (2) A countably infinite set of classes $C$. (3) A countably infinite set of names $IV$ called instance variables. (4) A countably infinite set of names $M$ called methods.
(5) A countably infinite set of object identifiers $O$.

A database schema is defined as follows:
DBS = (C, IV, DOM, $PC$, M, IS-A), where, C is a finite subset of $C$. IV is a function that assigns to each class in C a finite subset of instance variables from $IV$. DOM is a function that assigns for each instance variable a domain. This domain is either one of the Basic-Classes or one of the user-defined classes. If each instance variable of a class has its domain as one of the Basic-Classes, the class is called a *simple class*, otherwise is called a *complex class*. An object which is an instance of a simple class is called a *simple object*. On the other hand, if it is an instance of a complex class, it is called a *complex object*. $PC$ is a function that assigns for each class $C_i$ in C a predicate $P_{C_i}$. M is a function that assigns to each class in C a finite subset of $M$. IS-A is the hierarchy in which classes are organized in the schema and is defined as follows:
The IS-A relationship between classes is depicted as a rooted direct acyclic graph (DAG). The root is a system-defined class called OBJECTS. Other nodes in the graph represent user-defined classes in the schema. An edge $e(C_i, C_j)$, from class $C_i$ to class $C_j$ means that $C_i$ IS-A $C_j$. Class $C_i$ ($C_j$) is called to be a subclass (superclass) of $C_j$ ($C_i$), and written $C_i \ll C_j$ ($C_j \gg C_i$). The IS-A relationship is

transitive. That is, if $C_i \lessdot C_j$ and $C_j \lessdot C_k$, then $C_i \lessdot C_k$. Therefore, we need to distinguish between immediate and non-immediate super (sub) classes of a class. The set of immediate superclasses of a class C is defined as:

$$\{C_j \mid C_j \gtrdot C \wedge \neg \exists\, C_i, C_i \lessdot C_j \wedge C_i \gtrdot C\}.$$

Similarly, the set of immediate subclasses of a class C is defined as:

$$\{C_m \mid C_m \lessdot C \wedge \neg \exists\, C_n, C_n \gtrdot C_m \wedge C_n \lessdot C\}.$$

An instance of a schema is an assignment of objects to classes. An object O belongs to a class $C_i$ iff $P_{C_i}(O)$ is true. We define a subset of instances of a class as its direct instances. This set is defined as: Direct instances of class C =

$$\{O \mid P_C(O) = \text{true} \wedge P_{C_i}(O) \neq \text{true for any other class } C_i\}.$$

We also put the following two restrictions on objects in the database instance:
(1) An object can never be a direct instance of class OBJECT, and (2) Each object is a direct instance of only one class. This restriction is put in many object-oriented systems (e.g. ORION).
An instance of the schema also assigns for each method in a class a code that implements the method.
A database schema is represented by a rooted directed graph. Nodes in the graph represent classes and edges represent IS-A relationships between classes. An edge from class $C_i$ to class $C_j$ means that $C_i$ IS-A $C_j$. Fig.1 shows a schema for a university database.

### 2-2 Temporal Databases

For some applications, it is necessary to store the current status as well as the history of the world. In temporal databases, users may access history data by using temporal queries. The query contains, in addition to conditions should be satisfied by the output, the time at which the output is valid. Several efforts have been done to extend the relational data model with the time dimension.

Temporal databases are classified in [SA] into three types rollback, historical, and temporal databases. This classification is based on two criteria, the type of the time supported by the database and whether it is permitted to update the history. There are two types of time: transaction time and valid time. 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. Rollback databases support transaction time, historical databases support valid time, and temporal databases support both transaction and valid times. In rollback databases, however, it is not permitted to update the history as known of now.

In our discussion it is immaterial which time is supported. On the other hand, however, permitting updating the history has several problems. These problems will be discussed in Section 4-4. In context of the relational data model either tuples or attributes are time stamped, here, we assign time to objects (i.e. tuples).

## 3- Updates and Object Migration

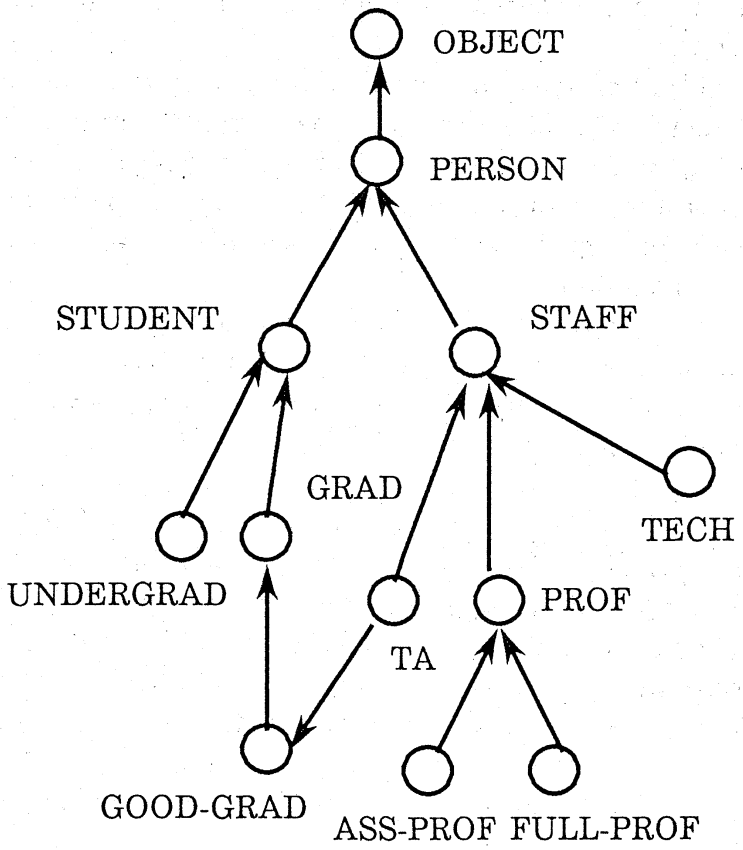


Fig.1 An example schema

In this section, we, informally, discuss updates and the notion of object migration in the object-oriented data model. The formal discussion of this issues is given in [EK]. We consider three types of updates on objects:
(1) Adding instance variables to an object, (2) Dropping instance variables from an object, and (3) Modifying instance variables of an object.
Objects belong to classes. An update to an object may cause object migration. That is, after the update the object is no longer an instance of its class and it has to *migrate* into a new class. For example, instances of class GOOD-GRAD students are defined to be graduate students with total marks exceeding certain value. 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). In [EK], we have studied actions that should be taken by the system when an object migrates from one class to another.

We need the following definition which is related to the coming discussions.

**Definition 1.** 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 $C_j$.

There are three types of object migrations: *Generalization*, *Specialization*, and *Migration*. Generalization is done when an object moves to one of its superclasses. Specialization is done when an object moves to one of its subclasses. Migration is done when an object migrates to class which is neither its super not subclass.

The notion of *migration graph* is introduced in order to trace how an object may migrate. It is defined as follows.

**Definition 2.** 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).

## 4- Temporal Object-Oriented Databases

Applications like medical and office information systems require adding the time dimension to the database in order to access the history of the real-world. Several research efforts have been devoted to study adding time to relational databases (see [M] for a bibliography). It seems that adding the time dimension to object-oriented databases have not been extensively studied [UD]. In temporal databases, objects' histories have to be stored. All versions of an object have the 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 the enhancement of the system performance. In this section, we give a simple, yet efficient, approach to improve the system response.

In this section we use the term temporal database to refer to a database that is augmented with the time dimension and an update on an object creates a new version of the object. The type of time supported by the database is immaterial.

Before, we discuss these problems, a comparison between temporal database issues and version management and control issues is given:

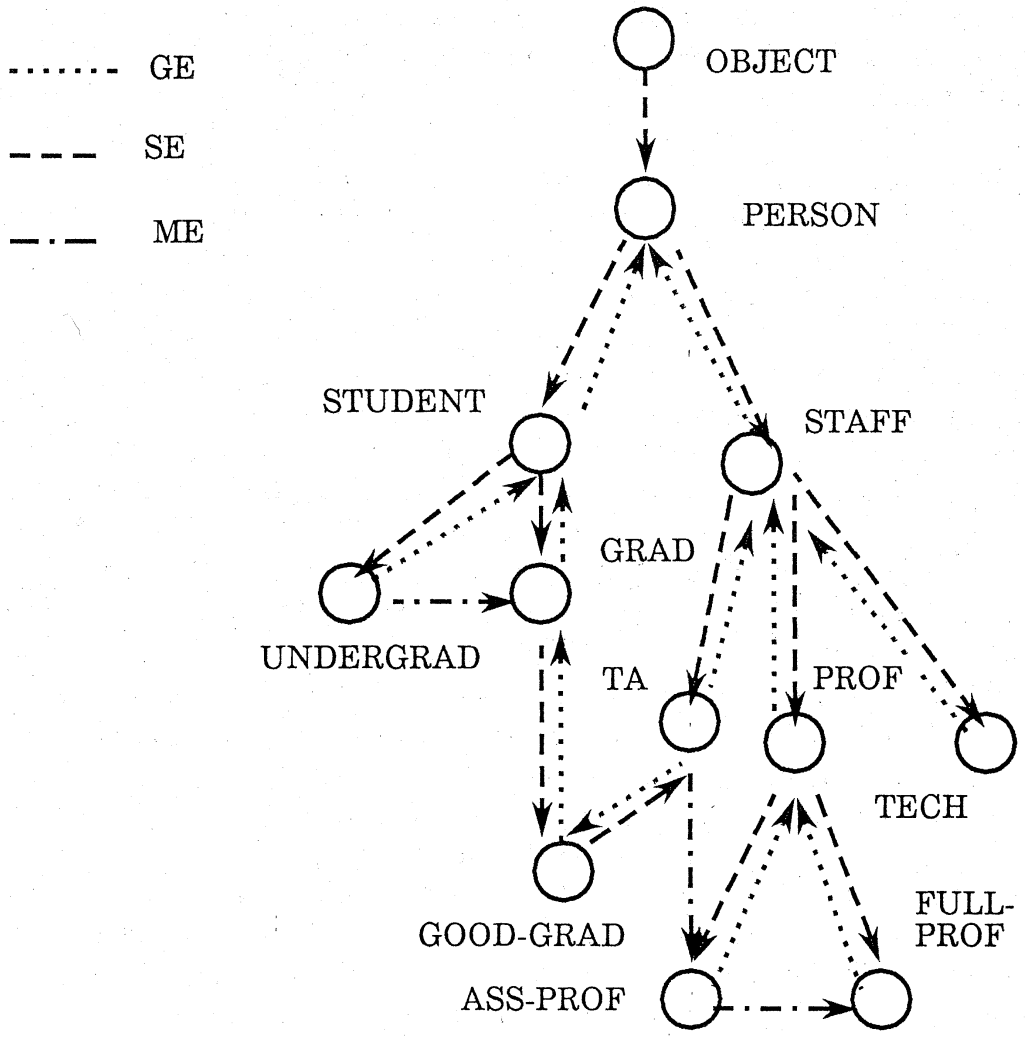


Fig. 2 Migration graph of the example schema

(1) In temporal databases there is a total order between versions of an object. (2) The number of objects in a temporal database is very large. Hence, the problem of efficiently answering queries is critical for the system performance. In CAD databases, there are many of transient and working versions of an object, however, the number of released versions is not so large. Thus, the problem of managing and controlling versions is important in CAD systems. (3) The problem of object migration is not considered in the literature on CAD systems. (4) A CAD object is complex and has several components that are related to the object through the IS-PART-OF relationships. In our model, we don't consider the semantics of IS-PART-OF relationship. The notions of static and dynamic binding of an object with its components is not applicable in databases with the time dimension. The binding in these systems is based on matching the life time of an object version with the versions of the objects it refers. That is, if a version of an object O is

created at time $t_i$ and updated at time $t_j$, its life time spans $t_i$ to $t_j$. Binding this object with the objects it references is determined as all versions of the referenced object with life spans intersect $t_i$ to $t_j$.

### 4-1 A Problem Due to Object Migration

From the previous analysis, we notice that the classes of the versions in an object's history can be specified by the set of classes that consists of the class of the initial version, all its stabilization points, and the classes between the initial class and each 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 class of the queried object, if it is not found search one of the next contiguous classes. This process continues until the required object is found. If the object cannot be found after visiting all the stabilization points of its initial 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 and in some cases the search is unnecessarily performed.

We propose two solutions to this problem. In the first solution, a new system-defined class is introduced. This class is used to keep track of updates in the system. In the other solution, each user defined class is extended by adding some system-defined instance variables to the class.

In the first solution, we introduce a new system-defined class called UPDATES. Instances of this class are updates introduced to the system. There is a tradeoff between storing all updates and only storing updates that enforce objects to migrate. In the first case, methods that answer queries on the real-world evolution can be associated with the class UPDATES. Examples of these methods are, find all updates at a specific time period, find all updates done on a certain object at a certain time, and so on. However, the number of instances in this class will be very large. In the second case, the number of instances is drastically reduced and we may not be able to easily answer queries that manipulate the real-world evolution.

Class UPDATES has the following instance variables: OBJECT, OBJECT-CLASS, TIME, and DEL-INDICATOR. OBJECT is the updated object, 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. When an update occurs, the object migration mechanism will create an instance of UPDATES. All updates over an object may be grouped together and sorted chronologically.

Answering a query will proceed in two steps: first, the class of the queried version is determined by consulting the class UPDATES. Second, when the class has been specified, a query will be invoked to this class on behalf of the user. We assume that a temporal query has the form: GET (O, Time-Specification), where O is the object to be accessed and Time-Specification is either a specific point of time (as zero, now, or $t_i$) or a period of time. When the system receives this message, it

sends to class UPDATES the message FindClass(O, Time-Specification). When receiving this message, UPDATES finds the classes to which object O belongs at this Time-Specification. Then, it sends the message GET (O, Time-Specification$_i$) to each class $C_i$ of these classes. Now, we give an outline of the procedure used to implement the message FindClass(O, Time-Specification).
1- Check that there is an update over O in UPDATES, if there is no such an update, O belongs to its initial class. Send the message GET (O) to class $C_i$.
2- Check that object O has not been deleted before time-specification T.
3- According to the time specification in the query, class UPDATES determines classes that contain the needed versions of the object O. For example, if Time-Specification is the time point "zero", UPDATES sends the message GET(O,zero) to the initial class of O. If Time-Specification is the time point "now", UPDATES finds the most recent update on O, determines the class of O, and sends it the message GET(O,now). If Time-Specification is the point = $t_q$, find an update which is done on object O at time $t_i$ such that $t_i \leqq t_q$, and there is no update on O done at time $t_j$, $t_i < t_j \leqq t_q$. Get the CLASS instance variable of this update. Send the message GET (O, $t_q$) to this class. In these cases, UPDATES sends a message to only one class. When Time-Specification is a period of time, UPDATES may send messages to different classes. For example, if Time-Specification is a period of time starting from $t_{qs}$ to $t_{qe}$, UPDATES will find an update which is done on object O at time $t_i$ such that $t_i \leqq t_{qs}$, and there is no update on O done at time $t_j$, $t_i < t_j \leqq t_{qs}$. Find an update on O that has done at time $t_k$, $t_k \leqq t_{qe}$, and there is no update done at time $t_l$, $t_k < t_l \leqq t_{qe}$. For each class $C_k$ in the update instances from $t_{i+1}$ to $t_{k-1}$, send a message GET(O). For the class corresponding to the update at $t_i$ send the message GET(O, $t_{qs}$ to $t_{i+1}$), and also for the class corresponding to the update at $t_k$ send the message GET(O, $t_k$ to $t_{qe}$). Time-Specification may also be a period from "zero" to $t_q$, $t_q$ to "now", or "zero" to "now".

Now, we give an outline of the second approach to solve the problem of finding the class of a specific version. 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 the name of a class in *n*, Start is the time at which a version of an object has been created in this class, and End is the 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 the 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) Due to cycles in the migration graph, some classes in the CLASS-HISTORY instance variables of a class may have more than one START and END pairs. (2) If we use this approach, the initial class of an object should be visited each time the object migrates. This can be done as an action which is automatically carried out as a result of object migration. Procedure OM can be modified to implement this action.

### 4-2 Problems Due to the Semantics of the Model

There are two problems due to the rich semantics of the model:
(1) Updating a shared-value instance variable in a class requires creating new versions of all instances of the class. Also, when a default-valued instance variable is updated, versions of all instances that consider this default value have to be created.
It is not efficient to create versions of all objects affected by updating the shared-value and default-valued instance variables. Instead, we associate with the class an array that stores updates on each of these instance variables and the times of the updates. Versions of objects are only created when a query is submitted. A synchronization process is necessary to determine the temporally matching default-valued or shared-value versions. For example, suppose that in class STAFF there is a shared-value instance variable named UNIVERSITY and its value is "Kyushu Institute". This value is propagated to the instances of all subclasses of STAFF. The UNIVERSITY value is updated twice, at time $t_1$ the value was changed to be "Kyushu Imperial University" and at time $t_2$ it was 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) An update on an object O' may create two versions, one of O', and the other of an object O that has O' as one of its (complex) instance variables.
Suppose there is a query that accesses a version of object O at time $T_q$. O has object O' as one of its instance variables. This query is split 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 $T_{q1}$ to $T_{q2}$ 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 split into three queries. In general, if this nesting level is *n* the original query should be split into *n* queries each accessing versions of one of the nested objects. Synchronization between the *n* histories has to be done.

### 4-3 Problems Due to Schema Evolution

Advanced applications, like CAD systems, require modifying the schema without interrupting system operations fully. In [BKKK], the approach taken in ORION to support schema modifications is presented. In this section, we study how to handle schema changes in temporal object-oriented databases. We follow the approach taken in [BKKK], we list schema changes and see how to handle them in temporal object-oriented databases. When the schema is updated, we create a new schema version and also a new schema instance. That is, we create a new version for each object affected by the schema change. For example, when a new instance variable is added to a class definition at time $t_i$, the system has to create new versions of all objects in class $C_i$. All queries that access objects in $C_i$ before time $t_i$ will not see the value of the new added instance variable. In this section, we list possible schema changes and give how to handle each change. These changes can be classified into two groups. In the first group of changes, we have to create a version of the schema along with versions of all objects affected by the schema change. In the other group, the system has only to store the kind and time of the schema change in order to decide which versions will be accessed as a query answer. That is, we do not immediately create versions of objects affected by the change, instead at the time of answering a query.

The first group of changes consists of:
(1) Add a new instance variable V to class C, (2) Drop an instance variable V from class C, (3) Make a class S a superclass of a class C, (4) Remove a class S from the superclass list of a class C, and (5) Make a shared variable V of class C unshared.
These changes are handled by creating a new version of the affected class and versions of all its objects. Instances of the new version of the class are created by creating versions from the most recent versions of the objects in the old class. The time of such a schema change should be stored in the database catalog.

The second group of changes consists of:
(6) Change the name of an instance variable V of a class C, (7) Change the name of a method of a class C, (8) Add a new method to a class C, (9) Change the name of a class C, and (10) Change the code of a method in class C.
All these changes can be handled by extending the database catalog. The catalog consists of three tables:
Classes (ID, Name, InstanceVariables, SuperClasses, SubClasses, Methods),
InstanceVariables (ID, Name, Class, Domain), and
Methods (ID, Name, Class, Code).
For each attribute in these table, except the ID, we add two additional history related attributes. The first is StartTime, which gives the time at which this value is effective. The second is EndTime, which gives the time after which this value is not valid. For example, suppose we have a method CalculateSalary in class Employee, and at Aug. 1989, the code implementing this method has changed. Instead of creating a new version of class Employee along with versions of all its

instances, the tuple that corresponds to this method in table Methods is modified. A new code value of method CalculateSalary will be added with StartTime Aug. 1989 and the EndTime time value of the old code will be set to July 1989.

### 4-3 Problems Due to History Update

Databases that support the time dimension are classified into three types [SA] rollback, historical, and temporal. One of the features of historical and temporal databases is the possibility of updating the history as known of now. For instance, a GRADE of a student was updated at time $t_1$ from C to B. Later, at time $t_2$, it is discovered that it should be A instead of B. In this case, the history can be updated at time $t_2$ to reflect the correct situation. In object-oriented databases, modifying the history may have side effects. This is because of the possibility of objects migration . The current status of the object may be affected by correcting the history.

There are two possibilities when the history is updated:

1- The correction of an update done at $t_i$ does not affect any update done at $t_j$, $t_j > t_i$. In this case, we apply the correction. It happens when: (a) The corrected instance variable is not one of those causing objects in the class to migrate, or (b) The corrected instance variable is one of those causing objects in the class to migrate, however the correction will not trigger the migration.
For example, a STUDENT will move from class GRAD to GOOD-GRAD when her TOTALGRADES > 100. A history correction to this instance variable that makes its value 70 instead of 60 will not cause migration.

2- The correction of an update done at $t_i$ does affect some update done at $t_j$, $t_j > t_i$. The corrected update may affect other updates in the following cases: (a) The correction will make one of the migration conditions satisfiable. For example, an object in class $C_i$ may migrate into class $C_j$ iff: $A \geqq 50$, $B < 80$, and $C = 20$, where A, B, and C are instance variables. Suppose at time $t_1$ A is updated to be 45, at time $t_2$ B is updated to 70 and C to 20. The update at time $t_2$ does not cause any migration. Later, at time $t_3$ it is discovered that the first update was not correct, A should be 54 instead of 45. If this mistake is corrected, the update at time $t_2$ should be completed, since, now, all conditions of migration are satisfiable.
(b) The corrected instance variable does not cause any migration to objects in the class, however, it may cause migration of some versions in some other class.

The system has to detect whether correcting an update has side effects. When the correction does not have any effect, it can be executed. Otherwise, the user has to be informed and the decision of performing the correction should be taken by the user. The system may support the user with information that help him to take the decision. These information may include the instance variables that causes the side effects, the new position of the object when the history is corrected, and etc.

Before we give an outline of the procedure that checks history corrections, we need the following definitions.

**Definition 3.** Associated with each class $C_i$ a set of *migration conditions* denoted by $MC(C_i)$. A migration condition has the form:
Instance variable OP Value, where OP belongs to $\{=, \neq, >, \geqq, <, \leqq\}$ and Value is obtained from the domain of the instance variable.

**Definition 4.** The class of the most recent version of an object O is called the *current class* of O and denoted CC(O). The class of the version that will be updated is called the *history update class* and denoted HUC(O). The *history path* of object O, denoted HP(O), is the union of HUC(O), CC(O), and each class $C_j$ such that O has a version belongs to $C_j$ and $C_j$ is in the path between HUC(O) and CC(O).

PROCEDURE HISTORY UPDATE
INPUT: Given a correction $U(O,IV_i, NEW\text{-}VALUE)$
OUTPUT: The correction is done, or, the user is informed of the side effects through a fiction modification.
METHOD: **for** every class $C_k$ in HP(O) **do**
**if** correcting the history makes makes one of $MC(C_k)$ true **than** inform the user; execute a fiction history update; **exit**;
**end do**;
perform the correction;
**exit**;
END HISTORY UPDATE

## 5- Conclusion

This paper showed problems and requirements necessary to build temporal object-oriented databases. The problems are due to the rich semantics of the model, object migration possibilities, and to requirements of new applications that schema evolution should be done without system shutdown.

## References


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

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

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

[EK] El-Sharkawi, M., Kambayashi, Y "Object Migration Mechanisms to Support Updates in Object-Oriented Databases", PARBASE-90, Int. Conference on Databases, Parallel Architectures, and their Applications, Miami, Florida, March 1990.

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

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

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