A Temporal Versioned Object-Oriented Data Schema Model

L. RODRÍGUEZ*, H. OGATA AND Y. YANO
Department of Information Science and Intelligent Systems, Faculty of Engineering
The University of Tokushima, 2-1 Minami Josanjima Cho, Tokushima 770, Japan
<iliana><ogata><yano>@is.tokushima-u.ac.jp

(Received October 1998; revised and accepted July 1999)

Abstract—This paper describes in a formal way a data schema model which introduces temporal and versioning schema features in an object-oriented environment. In our model, the schema is time dependent and the history of the changes which occur on its elements are kept into version hierarchies. A fundamental assumption behind our approach is that a new schema specification should not define a new database, so that previous schema definitions are considered as alternative design specifications, and consequently, existing data can be accessed in a consistent way using any of the defined schemas.

Copyright © 2001 Elsevier Science Ltd. All rights reserved.

Keywords—Object-oriented databases, Temporal databases, Schema versioning.

1. INTRODUCTION

An important characteristic of databases is that they provide sharing of data between multiple users. Therefore, it is essential for the schema to be carefully designed, satisfying the needs of different users and avoiding unnecessary future modifications. However, changes in the structures used to represent the information can be a consequence of different facts which occur frequently in the real-world. For example, mistakes made by database designers which need to be corrected, or changes on the essentials of the real-world which need to be abstracted on the database model.

On the other hand, there is not one truth, not a single correct understanding of the real world, consequently different users can have different interpretation of the same domain, or different users may have different needs of information about the same domain. So, it is in principle impossible to define a unique and static schema which satisfies all the database users and which lasts forever. Rather, as we propose in this paper, it may be more adequate if different schemas of the same database may coexist in such way that they can be used to manipulate in a consistent way the database.

In this paper, a distinction between evolution and versioning systems is made. The evolution approach regards all changes in the database as corrective [1]. After any schema modification is done, the existing data values are changed according to the new schema specification and so new information constitute the only valid state of the database. This fact does not mean that information about the past has to be removed from the database, for example, temporal models [2] permit to keep and query past information, but it is considered only as a part of the history of...
the system. In contrast, in versioning approach [3] changes are not regarded as corrective, and therefore, former specifications are not made obsolete upon change. The following reasons made our model to adopt schema versioning approach rather than schema evolution.

- Schema versioning promotes change transparency reducing possible expensive upgrades and recompileations of existing application programs. Changes done on the schema elements are local; they do not affect any of the existing schema versions which still might have application programs running on them. This assures that old applications can continue functioning properly and can interoperate with new ones. When schema changes are treated as a schema versioning process, the database can be viewed through new schema definitions and also, later recorded data can be viewed under previous schema.

- Although a database constitutes a means for sharing information, some schema changes may be motivated by needs which are not shared by all the users. Under the schema evolution approach changes must be invariably imposed to all users. In contrast, under schema versioning, only users who need to see the modifications are affected.

- Change is not necessarily corrective all the time, but may also be in order to introduce different interpretations of the same database structure. Different users may have different requirements for the storage and manipulation of information, which means that it may not be reasonable to deal with only a single schema. The schema versioning approach better reflects inherent diversity of human perception. It is well known that, in general, there are different interpretations of the real world in terms of abstractions and organization.

- In schema versioning, information is never removed neither considered obsolete; therefore, it will be always possible to manipulate former specifications of the database. In contrast, in schema evolution approach upgrades are irreversible, there is no way to return to former specifications in case the change was found to be inappropriate. Under the evolution approach, former specifications are always made obsolete while in versioning approach they represent different alternatives over the same database, so schema versioning is a flexible way for dealing with changes.

- Schema versioning allows database designers to derive new schema versions from other existing ones without damaging the database functionalities. Thus, old and new versions can be able to share the same data, independently from the schema through which they are originally created.

- Using time to maintain and manage schema changes enriches the database environment in the sense that it enables the database designers to retrieve different versions of the schema components which existed at any time since the database was created for the first time, reconstruct different alternatives of the database schema and trace the schema changes during time. On the other hand, the use of the database also gains a lot of flexibility, users can access stored data using the desired or required schema version.

Naturally, there are also problems related with schema versioning. The complexity of the systems increase as data has to simultaneously comply with different specifications. Additional dependences have to be managed. The manageability gets more difficult because multiple specifications of the same domain must be regarded, thus the efficiency of the system is affected. Maintaining the database consistently according to multiple specifications may incur performance penalties. The challenge is then, how to organize and maintain consistently a single database which simultaneously reflects multiple specifications of essentially the same domain.

In general, schema modification in an object-oriented database includes adding and dropping classes, adding and dropping inheritance relationships between classes, and adding and dropping attributes of a class, which affect many aspects of the system. There are two fundamental problems to consider. First, the semantics of change, which refers to the effects of the schema change on the overall way in which the information is organized (i.e., effects on the schema itself). Second, the change propagation, which refers to the method of propagating the schema change to the underlying objects. This paper focuses on the former problem, we consider in a formal
way several issues derived from adding temporal capabilities with schema version management to an object oriented model.

In the following sections, a model that accomplishes the information needs of schema versioning in object-oriented database systems is described. After presenting the related work, we introduce in Section 3 our data model called TVOO (Temporal Versioned Object Oriented) and outline the concepts of types, including the type version, values and objects. In Section 4, class, which is the main component of the schema, is described and all the schema elements (types, relationships, and methods) are introduced. The schema, its consistency and modification semantics are described in detail in Section 5. Section 6 gives the conclusions.

2. RELATED WORK

The issues related to schema modification have been a very active area of research in the database community during the past years. Although many different data models which consider schema modification have been proposed [4-13], there is not a formal definition of an object-oriented data model which includes temporal features and versioned schema management as our model does. With this lack of a formal basis, the schema versioning becomes ad-hoc and comparing different approaches is a difficult task.

In many of the previous works, the issues of schema modification and temporal object models are generally considered to be orthogonal and are handled independently. This is unrealistic if we consider that modifications always occur in time, so a model for managing schema changes should include the functionality of temporal models. Another intent for combining schema modification and temporal models can be found in [14], where schema evolution is managed using a temporal object model. However, our approach is more general since TVOO treats schema modification as a versioning process instead of an evolution, as it is in [14].

Like [4,5,15], TVOO assumes that a version element can be seen as a snapshot of an object taken at a certain time and also that several versions can coexist representing different and parallel states of the same object. From [16], we took the concept related with system level versioning, which ensures that every change done on an object results in a new version of that object. Therefore, our model TVOO [17], treats the modifications which occur during time as a version process of each object stored in the database. Since this approach works with the modification history of distinct entities, it offers the possibility of deriving snapshot states at any point of time. A time branching history is kept for each element defined in the database schema as a set of temporal versioned elements. They are represented by tree structures. Each version, which corresponds to an element of a tree, behaves independently of the others and has, among others, a unique identifier vid, a creation time tc, and possibly a deletion time td. During its lifespan ls, versions are called alive. Although reincarnations are allowed, a version can not have overlapping lifespans.

In TVOO there is no concept of current nor default schema as it is in all evolving systems and also in the versioning models of ADVANCE [16] and CLOSQL [10]; instead one can select any snapshot of the schema, i.e., the set of most recent classes at any point of time, for working with. In this way, in TVOO different versions of the same database coexist and are accessible during time. TVOO is the first model which considers the dynamic generation of different alternative schema version at any point in time.

Instead of converting the existing objects conforming to the current versioned schema and creating new objects under the same versioned schema, as it is proposed in ORION [5], Gemstone [6], OTGen [18], O2 [19], LISP02 [9], and COCOON [20], in our model objects are stored under the class version used for their creation, and then are transformed according to the class version used for accessing them. In this way, TVOO promotes change transparency in the sense that database users are not affected by the change. This mechanism can be compared with that of views (see [7,21]), but from the point of view of modeling ours is more flexible because in view

mechanism, view classes are always derived from base classes and so a view can be seen only as a subset of the schema.

In contrast to [5,22], in which schema versioning always handles versions of a complete schema, our model works at the granularity of class like those models proposed by Monk and Somemerville [10] and Talens et al. [23]. Therefore, in TVOO classes are defined as temporal versioned elements, so any class can be independently versioned while it is alive. However, when a class is modified, the change is propagated into the respective class hierarchy in order to preserve the temporal versioned schema in a consistent stage.

The notion of time as used in this paper corresponds to transaction time, that is, the time in which the fact is stored in the database [2]. In this presentation, time is assumed to be discrete and is described by a succession of nonnegative integers in their usual order.

3. TVOO DATA MODEL

Our model, called TVOO, is based on the object model of SIORE1 [24] that encompasses the major characteristics of most object-oriented databases.

3.1. Model Basics

All data in TVOO is encapsulated in objects which are uniquely identified by identifiers (oid), grouped into classes and related to other objects through relationships.

A class which is a template for creating objects also defines a set of attributes which qualify the objects. Attributes can be either variables, relationships or methods. Classes can be defined in terms of other classes creating in this way class hierarchies, so attributes can be either inherited from other classes (called superclasses) or introduced by the class itself.

A type concerns with the abstract description of the variable attributes of a class. In TVOO, types can be either simple or constructed. There is a finite set of simple types $SY = \{\text{atomic types, strings, time}\}$, where atomic types includes integer, floating point, boolean, and char. On the other hand, constructed types $CY$ are built from base types, which can themselves be simple or constructed. A constructor can be either a record type or a sequence type. A record is a fixed heterogeneous sequence of fields, selected by field names, while a sequence is a homogeneous sequence of objects of a given base type.

Structurally, an object is a container for a value corresponding to the type defined in the class to which the object belongs. In TVOO, a value is a unit of stored state. The state of an object is a value, as is the state of each attribute of the object and each component of a constructed value. For each predefined simple type $s \in SY$, there exists a fixed nonempty set of values domain, denoted by $\text{Dom}(s)$. Thus for instance, the domain of the type int is the set of integers $\mathbb{Z}$ and the domain of the type time is $T = \{t_0 = 0, t_1 = 1, t_2 = 2, \ldots, t_k = \text{now}, \ldots\}$. Each of the values $t_i$ is called instant, $t_0$ denotes the database relative beginning instant, and now is a special constant that represents current time, whose value is advancing. Any instant beyond now, that is, $t_{k+1}, t_{k+2}, \ldots$, is future.

Although objects do not nest, it is possible to define a part-subpart relationship between two objects, in this case each object has an independent identity and lifetime.

Moreover, all objects are temporal (or historical), that is, their values can be changed over time, and consequently, the different values are stored in the database. Thus, a state of this temporal object-oriented database at a given time $t$ involves objects and references between objects which are valid at that point of time. To manage the changing values during time, we introduce the concept of Version Type which keeps the history of the changed objects.

In this paper, we clarify and extend the work presented in [17], where we were mainly concerned with the definition of Temporal Versioned Objects. Here TVOO is expanded to include

1Scalable Heterogeneous Object REpository, a persistent object system which uses the OMG data model as starting point for its definition.
Data Schema Model 181

Schema Versioning, so all the facts related to class manipulation and schema version derivation are described. The concepts of Temporal Versioned Schema together with the analysis of the Temporal extent of the classes are formally defined.

3.2. Database Schema

As in most object-oriented models, in TVOO classes can be defined in terms of other classes creating in this way class hierarchies. Therefore, the TVOO schema consists of a time varying set of class hierarchies $C$ and the elements which are part of their definition. Each class hierarchy $C \in C$ contains the definition of the classes $c \in C$ and the relationships between them.

Every class $c \in C$ is temporal versioned, that is, $c$ can be changed and its different definitions are kept and coexist over time. To manage the changing classes during time, the Version Class Hierarchy is defined in the same way that Version Type was done for Temporal Versioned Objects in our former paper [17].

In TVOO, the scope of a schema version is not limited to the objects which have been created under it, instead any object in the database can be accessed and modified through any version of the schema. Thus, each object can be shared by different version of the schema, as it will be described in the following sections.

3.3. Temporal Versioned Basics

To facilitate the definition of Temporal Versioned elements, that is, object and classes, some facts and terminology concerning trees [25] are first introduced.

A tree is a partially ordered set $(P, \leq_p)$ such that $\forall p \in P$, the set $(\cdot, p)_P = \{ q \in P : q <_p p \}$ is well ordered.

The height $ht_P(p)$ of $p$ in $(P, \leq_p)$ is the order type of $(\cdot, p)_P$. The $\alpha$ level of $(P, \leq_p)$ is the set $R_\alpha P = \{ p \in P : ht_P(p) = \alpha \}$.

We shall identify $(P, \leq_p)$ with its domain $P$. The height $ht_P$ of $P$ is the ordinal $\min \{ \alpha : R_\alpha P = \emptyset \}$. We will be concerned with trees having the property:

$$ht_p(p) < \omega_0, \quad \forall p \in P,$$

i.e., $ht_p(p)$ is a finite number which implies that $ht_P \leq \omega_0$, where $\omega_0$ is the first infinite ordinal, and so, $ht_P$ is either finite or countable. Also, because of property (f), we get that the levels are all levels of finite height. For example, the $n$th level is $R_n P = \{ p \in P : ht_P(p) = n \}$ and $R_0 P = \{ p \in P : ht_P(p) = 0 \}$.

Let $p, q \in P$, then $\rho(p, q) = (\cdot, p)_P \cap (\cdot, q)_P \equiv$ the set of common roots of $p$ and $q$. Because of property (f), we can define $\max \rho(p, q) \equiv R(p, q) \equiv$ most recent common root of $p$ and $q$.

A node of $P$ is any equivalence class of the relation $\sim$ defined on $P$ by $p \sim q$ if and only if $(\cdot, p)_P = (\cdot, q)_P$. So every node $N$ of $P$ is a subset of some level $R_n P$ of $P$ in which case we call $n$ the height of $N$ in $P$. Note that for every $p \in P$, the set of all immediate successors of $p$ is a node of $P$ which we denote as $N_p = R_0 \{ q \in P : p <_p q \}$. In our case, i.e., if property (f) holds, every node has this form (but in general, without (f), this is not true). Therefore for every node $N$, there exist a $p \in P$ such that $N = N_p$.

A branch of a tree $P$ is a maximal chain of $P$. A path of $P$ is any chain of $P$ which is also an initial part of $P$, where a subset $U$ of $P$ is said to be initial if $(\cdot, p)_P \subseteq U, \forall p \in U$.

3.3.1. Version hierarchies

Every element $p \in P$ defines a subtree of $P$ denoted by $P^p = \{ q \in P : p \leq_p q \}$ whose initial root is the element $p$ and the elements of $P^p \setminus \{ p \}$ are all the successors of $p$ in $P$.

Let $OI$ be the set of object identifiers (oid) and $CI$ the set of class identifiers (clid). By $OI|t$ and $CI|t$, we denote the set of object identifiers and the set of class identifiers, respectively,
at time \( t \in T \). In our model for every \( t' \leq t \), \( OI_{t'} \subset OI_t \) and \( CI_{t'} \subset CI_t \). Let \( Y \) be the set of types (see Section 4.1.4). For every type \( y \in Y \) by \( y_{e|t} \), we denote the extension of type \( y \) at instant \( t \), that is, the set of valid type values of type \( y \) at instant \( t \). If \( c \) is the class identifier of the class \( c \), we also denote the time varying extension of the class hierarchy \( C \in C \) by \( C_{e|t} = \bigcup \{ c_{e|t} : c \in C \} \), where \( c_{e|t} \) is the extension of the object type of the class \( c \). Similarly we have \( Y_{e|t} = \bigcup \{ y_{e|t} : y \in Y \} \), and so, \( C_{e|t} \subset Y_{e|t} \). Also, for \( C \in C \), let \( \Omega C_{e|t} = \bigcup \{ c \times c_{e|t} : c \in C \} \).

**Definition 1. Identifiers Version Hierarchy Ver**

\[
CVer = \left\{ (P, \leq P) : (P, \leq P) \text{ has property } (f), \right. \\
P \subset \bigcup_{t \in T} (CI_t \times \{a, d\}), \quad \leq P \text{ is such that if } \\
p = (i_p, \ast), \quad q = (i_q, \ast), \quad p <_T q, \quad \text{then,} \\
i_p \in CI_{i_p}, i_q \in CI_{i_q}, \quad \text{and } t_p <_T t_q \right\},
\]

where
- \( \leq_T \) is the order in \( T \),
- \( p = (i_p, \ast) \) is because now \( p \in P = \bigcup_{t \in T} (CI_t \times \{a, d\}) \), and so, \( i_p \in CI_{i_p} \) and \( \ast \) is either "a = alive" or "d = dead"
- \( t_p <_T t_q \) is understood as \( t_p \) being chronologically before \( t_q \), as it is written above.

Similarly \( OVer(C) \) is defined by substituting \( CI_t \) by \( \Omega C_{e|t} \) in the above, and \( OVer = OVer(C) = \bigcup \{ OVer(C) : C \in C \} \). Let \( Ver = CVer \cup OVer \).

Below by \( I_{|t} \), we mean either \( CI_t \) or \( \Omega C_{e|t} \) for some \( C \in C \).

**Definition 2. Version Hierarchy State S.** For every \( t \in T \) we define the state of a version hierarchy \( P \in Ver \) at \( t \) by

\[
S_t(P) = \left\{ p \in P : p = (i_p, \ast), \quad i_p \in I_{|t_p}, \quad t_p \leq_T t \right\}.
\]

If \( p \in S_t(P) \), then \( (\cdot, \cdot)_P \subseteq S_t(P) \), i.e., \( S_t(P) \) is an initial part of \( P \). Also, we can define the subset \( SR_t(P) \subset S_t(P) \) as

\[
SR_t(P) = \left\{ p \in S_t(P) : P^d \cap SR_t(P) = \{ p \} \right\},
\]
i.e., only recent elements of \( S_t(P) \).

As for "alive" and "dead" elements of a hierarchy \( P \in Ver \), we define

\[
P^a \equiv S^a(P) = \{ p \in P : p = (\cdot, a) \},
\]

\[
S_t^a(P) = P^a \cap S_t(P),
\]

\[
SR_t^a(P) = P^a \cap SR_t(P).
\]

Similarly we define \( P^d \equiv S^d(P) \), \( S_t^d(P) \), and \( SR_t^d(P) \) which represent the dead elements of the hierarchies, the dead elements of \( S_t(P) \) and the recent dead elements of \( SR_t(P) \), respectively.

### 3.3.2. Temporal versioned hierarchies tem

We also assume that \( Ver \) is closed under time subhierarchies, that is, we have that if \( P \in Ver \), then \( S_t(P) \in Ver \), for every \( t \leq_T Cht(P) \), where \( Cht(P) = \inf \{ t \in T : S_t(P) = P \} \) is the Chronological height of \( P \).
We further define the following sets for any time $t \in T$:

$$CVer|_t = \{ P \in CVer : Cht(P) \leq t \},$$

$$CTem|_t = \{ P \in CVer|_t : \text{if } P \subset P', P' \in CVer|_t, \text{ then } P = P' \}.$$

Similarly we define $OVer(C)|_t, OVer|_t = OVer(C)(t = U\{OVer(C)|_t : C \in C\}$, and $OTem|_t = OTem(C)|_t = U\{OTem(C)|_t : C \in C\}$. Let $Ver|_t = CVer|_t \cup OVer|_t$ and $Tem|_t = CTem|_t \cup OTem|_t$.

Thus $Ver|_t$ is the set of hierarchies whose elements were created before or at instant $t$ and $Tem|_t$. Temporal Versioned Hierarchy, is the set of maximal hierarchies in $Ver|_t$. As was mentioned in [17], we have the following consistency axiom in our model: for any identifier $i \in CI|_t$, $t \in T$, there exists one and only one $P = P(p, t) \in CTem|_t$ and one and only one $p \in P$ such that $p = (i, \ast)$.

3.3.3. Class inheritance hierarchies

Subclassing is an object-oriented feature that allows classes to be built incrementally from other classes and can be naturally combined with the versioning one as we propose. In our model classes are grouped into class hierarchies where a class hierarchy is a combination of class inheritance hierarchy and class version hierarchy. Thus a class hierarchy is a set with two structures in it, a tree structure and a lattice structure. In other words, by a class hierarchy (or class identifier hierarchy) we understand a triple $(P, \leq_P, \preceq_P)$, where $(P, \leq_P)$ is a tree (class identifier versioned hierarchy) which depicts the versioning structure as explained above and $(P, \preceq_P)$ is a lattice (class identifier inheritance hierarchy) which depicts the subclassing-superclassing structure. Thus every element $P \in CTem|_t$ has an additional structure to the one given above. By a lattice, we do not mean a lattice in the strict mathematical sense of the word. In fact, our term lattice is the same as that of trees without the property of well ordering in the chains of $P$. Thus, in this paper by a lattice we understand a partially ordered set $(P, \leq_P)$, such that for every $p \in P$, the set $(\langle \cdot, p \rangle)_P = \{ q \in P : q \prec_P p \}$ is finite. If $c_2 \prec c_1$, then we say that $c_2$ is a superclass of $c_1$, and so, $c_1$ is a subclass of $c_2$. Also, for $q, p \in P$, we denote by $M(q, p) = \{ m \in P : q \prec_P m \prec_P p \}$.

**Definition 3. Immediate Superclasses ISup.** Given an element $p \in P$, the immediate superclasses set $ISup(p) = \{ q \in P : q \prec_P p, M(q, p) = \emptyset \}$.

**Definition 4. Immediate Subclasses ISub.** Given an element $p \in P$, the immediate subclasses set $ISub(p) = \{ q \in P : q \succ_P p, M(p, q) = \emptyset \}$.

**Note 1.** For every $p \in P \in OTem|_t$, we have $ISup(p) = ISub(p) = \emptyset$, since every $P \in OTem|_t$ has only one structure in it, the tree structure.

Similarly to the above sets, one can define the following two sets.

**Definition 5. Immediate Ancestor IAnc.** Given an element $p \in P$, the immediate ancestor $IAnc(p) = R(p, p)$.

**Definition 6. Immediate Descendants IDesc.** Given an element $p \in P$, the immediate descendants set $Ides(p) = N_p$.

4. CLASSES

A TVOO class $c \in C \in C$ has two components: the definition and the extent. While the definition part contains all the information related with the class structure and behaviour, the extent maintains the history about the objects, called instances, whose versions contain the said class. The class definition includes two parts: the interface definition and the implementation. The interface definition contains the information about the type of the objects which are the extent of the said class, while the implementation part contains the code which executes the methods. The description of the later part is omitted in this paper, since it is language depending.
4.1. Class Definition

A class definition in TVOO consists of a set of distinctly named attributes which can be either variables, relationships, or methods. Variables define the repositories for information hidden inside the objects, relationships make possible the establishment of links between objects and methods form the executable part of an object so they may change its state and/or return information about its current state.

To each class identifier \( cid \in CI \), there corresponds a class whose class definition \( c \) is as follows.

**DEFINITION 7. CLASS DEFINITION.** A class definition (or for brevity, Class) \( c \) is an 8-tuple \((p, name, ls, typ, rel, meth, pred, succ)\), where

1. \( p \) is the version element \((cid, *)\) of the class \( c \);
2. \( name \in AN \) is the name given to the class \( c \);
3. \( ls \) is the lifespan of the class \( c \);
4. \( typ \) is a type which contains the description of the object type associated with \( c \);
5. \( rel \) is a set which contains the relationships between objects of the class \( c \) and others;
6. \( meth \) is a set which contains the methods defined for \( c \);
7. \( pred \) are the sets of class identifiers from which \( c \) is derived either as a version or as a subclass;
8. \( succ \) are the sets of class identifiers which are derived directly from \( c \) either as a subversions or as a subclasses.

4.1.1. Class version \((p)\)

There is a 1-1 correspondence between classes and class identifiers (and so between classes and class versions \( p \in P, P \in CTem|_t \) at any time \( t \in T \).

For any \( t \in T \) and any \( P \in CTem|_t \), let \( C(P) = \{ p : (p, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \in P \} \), i.e., the class hierarchy corresponding to \( P \). Thus, for every class hierarchy \( C \in C|_t \), there corresponds a unique element \( P = P(C) \in CTem|_t \), and for every \( P \in CTem|_t \), there corresponds a unique class hierarchy \( C = C(P) \in C|_t \), and also we have that \( C(P) = C \) and \( P(C(P)) = P \). Therefore we get that

\[
 C|_t = \bigcup_{P \in CTem|_t} C(P),
\]

and vice versa,

\[
 CTem|_t = \bigcup_{C \in C|_t} P(C).
\]

**NOTE 2.** For every \( P \in CTem|_t \), the corresponding \( C(P) \) is also endowed with two structures, the same as \( P \). That is, \( C(P) \) can be looked at as a triple \((C(P), \leq P, \preceq P)\). Thus, for every \( c \in C, C \in C \), there exist one and only one temporal versioned element \( p = (cid, \cdot) \), where \( cid \) is the unique class identifier of \( c \), \( cid \in CI \).

We now define the following sets which we would need below. For every \( p \in P \in CTem|_t \), let \( C(p) \) be the corresponding class in \( C(P) \) and similarly for every \( c \in C \in C|_t \), let \( p(c) \) be the corresponding element in \( P(C) \). We can now write the mentioned set as follows:

\[
\begin{align*}
\text{sup}(c) &= \{ c' : p(c') \in Isup(p(c)) \}, \\
\text{sub}(c) &= \{ c' : p(c') \in Isub(p(c)) \}, \\
\text{anc}(c) &= \{ c' : p(c') \in Ianc(p(c)) \}, \\
\text{des}(c) &= \{ c' : p(c') \in Ides(p(c)) \}.
\end{align*}
\]
4.1.2. Name (name)

A name is a “character string” which can be considered as the semantic identifier given to a class. Let $\mathcal{CN}_t$ be the set of class names existing at time $t \in T$. For every $c = (., n, ., ., ., ., .) \in C[1]_t$, we have $\text{name}(c) = n \in \mathcal{CN}_t$.

**Axiom 1.** Let $P \in C \text{Tem}_t$ and $C = C(P)$. Let $c_1$ be any class in $C(P)$. Then, for any other class $c_2$, $c_2$ can have the same name as that of $c_1$ if and only if $c_2 \in C(P)$ and there exists $p \in P$ such that $p_1, p_2 \in (., p)P$, where $p_i$ corresponds to the class $c_i$, $i = 1, 2$, that is, $c_i = (p_i, ., ., ., ., .)$. Thus, unless there is the above connection between two classes, then they cannot have the same name.

4.1.3. Lifespan (Is)

Each element $p \in P \in T \text{em}_t$ has a lifespan which we define in the following way.

**Definition 8. Lifespan.** Let $D$ be the “death time” operator

$$D : S^d(P) \equiv P^d \to T, \ P \in Tem_t,$$

$$p = (i_p, d) \to D(i_p, d) = t_d,$$

where $i_p \in I_t$ and $t_p < t < t_d$ (with $t_p$ being the creation time of $p$).

The lifespan $ls$ of an element $p = (i_p, *) \in P \in Tem_t$ is the time interval

$$ls(i_p, *) = \begin{cases} [t_p, t], & \text{if } * = a, \\ [t_p, t_d], & \text{if } * = d, \end{cases}$$

where $[t_p, t] = \{t' \in T : t_p \leq t' \leq t\}$ and $[t_p, t_d] = \{t' \in T : t_p \leq t' \leq t_d < t\}$.

By the lifespan of a class $c$, we understand the lifespan of $p = (cid, *)$, where $cid$ is the unique class identifier of $c$. This is well defined because of 1-1 correspondence between classes and class identifiers.

4.1.4. Type (typ)

A typ which corresponds to the set of “instance variables” in SmallTalk or “data members” in C++ defines the information repositories hidden inside the objects. In TVOO, all data members are typed, in the sense that they can only hold values of the associate type domain. Types can be either simple or constructed.

**Definition 9. Types Y.** The set of types $Y$ is defined recursively as follows.

1. The finite set of simple types $SY \subset Y$, where $SY = \{\text{atomic.types}, \text{strings}, \text{time}\}$ and
   - atomic.types includes integer, floating_point, Boolean, and char;
2. If $a_1, \ldots, a_n$ are different members of $\mathcal{A}$ and $y_1, \ldots, y_n \in Y$, then $\text{record.of}(a_1 : y_1, \ldots, a_n : y_n)$ is a constructed type $CY$ denoted as $\text{record.of type}$. A record is a fixed heterogeneous sequence of fields, where a field is composed of a name and a type. The names in a record are all distinct;
3. If $y \in Y$, $\text{list.of}(y)$, $\text{set.of}(y)$ and $\text{bag.of}(y)$ are $CY$ denoted as $\text{sequence.of type}$. A set contains zero or more distinct objects, and a bag and list, zero or more objects that are not necessarily distinct. List members are ordered while bag members are not;
4. $Y = SY \cup CY$.

As can be noticed from the above definition, $SY$ is the only fixed set since $CY$, and hence, $Y$, can be modified during time. By $Y_t$, we denote the set of types at time $t \in T$ (which is distinct from $Y_c_t$ which denotes the set of valid type values at time $t$).
DEFINITION 10. OBJECT TYPES $OY$. The set of Object Types $OY$ is defined recursively in terms of Class Hierarchies $C$, in the following manner.

1. For each $C \in C|_t$, $t \in T$, there corresponds a unique $P = P(C) \in CTEM|_t$, and so each $C \in C|_t$ has two structures in it as defined in Section 3.3.3. Here by $C|_t$, we understand the set of class hierarchies at time $t \in T$;
2. $OY|_t = \bigcup \{oby(c) : c \in C, C \in C|_t\}$ is the set of object types at time $t \in T$, where $oby(c)$ is the type component (typ) of $c$. Thus, $OY|_t \subset Y|_t$. In our model for every $t' \leq t$, we have that $OY|_{t'} \subset OY|_t$ and $Y|_{t'} \subset Y|_t$.

4.1.5 Relationships (rel)

A rel defines a set of special kind of attributes called relationship, which make possible the establishment of links between objects. Like other attributes, relationships attributes take values which are part of the state of the objects. Therefore, a relationship must designate a class $c \in C$, called the “target class” of the relationship which defines the set $E_t(c)$ from where the possible linked object can be taken. The set $E_t(c)$ is the set of all objects containing the class $c$ in their version at time $t \in T$ (see Section 4.2).

It is possible to declare a pair of relationships to be inverse to each other, meaning that an inverse link is also maintained. If $r^-$ is declared to be inverse to $r$, $r$ must be declared to be inverse to $r^-$ in the respective target class of $r$.

DEFINITION 11. RELATIONSHIPS. A relationship $r$ is a four-tuple $(name, target, inv, c)$, where

1. name $\in AN$ is the name given to $r$;
2. target is the target class $c' \in C$ as explained above;
3. inv $\in AN \cup \{\Lambda\}$ is the name given to the inverse of the relationship or $\Lambda$ if rel is not paired with an inverse;
4. c is the class identifier of the class in which $r$ is defined.

Let $R(t)$ be the set of all the relationships defined at time $t \in T$. Thus, $rel_t(c) = \{r \in R(t) : r = (\cdot, a, \cdot, c)\}$ are the relationships defined at time $t \in T$ for $c \in C$.

For $n < \omega$, let $E^n_t(c)$ be the product of $n$ copies of $E_t(c)$, that is, $E^n_t(c) = \prod_{i=1}^{n} X_i$, where $X_i = E_t(c)$, for $i = 1, \ldots, n$. Also, let

$$F^n_t = \bigcup_{c \in C, C \in C|_t} E^n_t(c).$$

DEFINITION 12. VALUE OF RELATIONSHIP $r$. For $r \in Rel_t(c)$, the value of $r$ at time $t$ is a function $r|_t : E_t(c) \rightarrow (\bigcup_{n < \omega} F^n_t) \cup \{\emptyset\}$.

Thus, $R|_t = \bigcup_{r \in R(t)} r|_t$ depicts all the values of the relationships in the database at time $t \in T$. If there are not relationships attributed to some object identifier, this some identifier is mapped by $r|_t$ to the empty set $\emptyset$.

EXAMPLE 1. Given two classes $c_1$ and $c_2$ between which the inverse pair relationships $r$ and $r^-$ are defined in such a way that $r \in rel_t(c_1)$ and $r^- \in rel_t(c_2)$. If at any time $t' \geq t \in T$ there are two object identifiers $o_1 \in E|_t(c_1)$ and $o_2 \in E|_t(c_2)$, such that $r|_t(o_1) = (o'_1, \ldots, o'_m = o_{n+1}, \ldots, o''_{n+1})$, then $o_1$ is a factor of $r^-|_{t'}(o'_1, \ldots, o''_{n+1})$, then there exists a $k(i), 1 \leq k(i) \leq n(i)$, such that $o''_{k(i)} = o_1$.

4.1.6. Methods (meth)

A meth is a set of methods that introduces the operations of the class. Each method, which defines the signature of a method of a class, is identified by its name; it has a list of arguments that must be supplied to the method when it is invoked. It optionally returns a value result which as well have an associated type.
Definition 13. Method. A method \( m \) is a five-tuple \( (\text{res}, \text{name}, \text{par}, \text{imp}, c) \), where we have the following.

1. \( \text{res} \) is the type \( y \in Y(t) \cup \{\Lambda\} \) which defines the range \( y|_t \) of \( m \), that is, the results of \( m \) are valid values of \( y|_t \). If the method does not return a result, then the range is \( \Lambda \).
2. \( \text{name} \in AN \) is the name given to \( m \).
3. \( \text{par} \) is a list which contains the description of the parameters of \( m \). Each parameter is a 3-tuple \( (\text{mode}, \text{type}, \text{name}) \), where \( \text{mode} = \{\text{in}, \text{out}, \text{inout}\} \) indicates if the corresponding argument is modified or not by the method. A parameter of \( \text{in} \) mode is not modified and a parameter of \( \text{out} \) mode is independent of the initial contents supplied as an argument, \( \text{type} = y \in Y(t) \) defines the domain \( y|_t \) for the parameter, and \( \text{name} \in AN \) is the name given to it.
4. \( \text{imp} \) is the implementation of \( m \), that is, the code which \( m \) executes upon invocation. Since it is specified in a language-dependent manner, the syntax details are omitted in this paper.
5. \( c \) is the identifier of the class in which \( m \) is defined.

Let \( M(t) \) be the set of methods at time \( t \in T \). Thus, \( \text{meth}_t(c) = \{m \in M(t) : m = (-, \cdot, \cdot, c)\} \) is the set of methods defined at time \( t \in T \) for a class \( c \in C \).

4.1.7. Predecessors (pred) and successors (succ)

As we mentioned in Section 3.3.3, in TVOO class hierarchies have a combined structure \((C(P), \leq_P, \prec_P)\) which includes versioning \((C(P), \leq_P)\) and inheritance \((C(P), \prec_P)\). So, the predecessor (pred) of a class \( c \) can be either \( \text{anc}(c) \) or \( \text{sup}(c) \). In the former case, it is either a one element set (or \( \emptyset \)), while in the latter case it is a finite set (or \( \emptyset \)). One must add that for any class \( c \), \( \text{anc}(c) \cap \text{sup}(c) = \emptyset \). In other words, one cannot have both \( p \prec_P q \) and \( p \prec_P q \) for any \( p, q \in P, P \in CT_{\text{em}}|_t, t \in T \). Thus, if \( p \) and \( q, p \neq q \), are \( \leq_P \) related, then they cannot be \( \preceq_P \) related and vice versa. On the other hand, the successor (succ) of a class \( c \) are two finite sets \( \text{des}(c) \) and the set of \( \text{sub}(c) \). Again, these two sets cannot intersect.

4.2. Class Extent

The extent of the class is the set of its instances, that is, the objects which had been created using the said class. In our model objects are temporal versioned, so they are treated in a similar way as classes. We will now briefly give some notions related with objects, one can consult [17] for more details.

In TVOO the history of object modifications is represented by \( OT_{\text{em}}|_t \). Each object besides its version \( v = (c, c_t, \ast) \), has a lifespan which keeps the time interval during which it is alive in the database, its immediate descendants \( N_v \) and its ancestor \( R(v, v) \) to keep the changes occurring during time.

Let \( DB(C)|_t \) be the set of objects at time \( t \in T \) whose version \( v \) contains a class identifier \( c \) of a class \( c \in C \). Also let \( DB|_t = \cup\{DB(C)|_t : C \in C\} \), that is, all the objects in the database at time \( t \in T \). Similarly one can define the “alive” objects of the database which we denote by \( DB^a(C)|_t \) and \( DB^a|_t \).

For any class identifier \( c \in CI \),

\[
E_t(c) = \{o : o \in DB|_t, \text{class}(o) = c\},
\]

and

\[
E^a_t(c) = \{o : o \in DB^a|_t, \text{class}(o) = c\}.
\]

Since one of the main objectives of our model is that objects can be query under version of a class, we would also need the following sets. Consider a class \( c \in C \) and its corresponding
Let \( p_0 \) be the minimal element of the set \( \{ \cdot, p \} \) and let \( P_c = P^{p_0} \).

Then we define

\[
VE_t(c) = \{ o : o \in DB|_t, \, \text{class}(o) = c, \, c = \text{id}(p), \, p \in P_c \},
\]

and

\[
VE^a_t(c) = \{ o : o \in DB^a|_t, \, \text{class}(o) = c, \, c = \text{id}(p), \, p \in P_c \}.
\]

Table 1 shows the class extent functions defined in TVOO. It can be noticed that by \( \text{crt} \) we mean "created", by \( \text{dld} \)—"deleted", by \( \text{alv} \)—"alive", and by \( \text{ext} \)—"existing" objects containing a given class.

### Table 1. Class extent functions at any time \( t \in T \).

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{crt} )</td>
<td>( C \times { t' : t' \leq t } \rightarrow - \bigcup_{C \in C} \bigcup_{t' \leq t} 2^{DB(C)} )</td>
<td>( (c, t') \rightarrow { o : o \in VE_t(c), o = (...) } )</td>
</tr>
<tr>
<td>( \text{dld} )</td>
<td>( C \times { t' : t' \leq t } \rightarrow - \bigcup_{C \in C} \bigcup_{t' \leq t} 2^{DB(C)} )</td>
<td>( (c, t') \rightarrow { o : o \in VE_t(c), o = (...) } )</td>
</tr>
<tr>
<td>( \text{alv} )</td>
<td>( C \times { t' : t' \leq t } \rightarrow - \bigcup_{C \in C} \bigcup_{t' \leq t} 2^{DB^a(C)} )</td>
<td>( (c, t') \rightarrow { o : o \in VE^a_t(c) } )</td>
</tr>
<tr>
<td>( \text{ext} )</td>
<td>( C \times { t' : t' \leq t } \rightarrow - \bigcup_{C \in C} \bigcup_{t' \leq t} 2^{DB^a(C)} )</td>
<td>( (c, t') \rightarrow { o : o \in VE^a_t(c) } )</td>
</tr>
</tbody>
</table>

**NOTE 3.** Although in our model each object is of one specific class, objects can be modified into objects whose versions are of different classes from the same class hierarchy.

### 5. SCHEMA

Since the main objectives of our model is to keep the history of the data base in order to be able to query the objects under any \( c \in C \subseteq C|_t \), the schema at time \( t \in T \) is defined as the union of the temporal sets introduced in Section 4.1. Thus,

\[
\text{Schema} = S|_t \equiv C|_t \cup OY(t) \cup M(t) \cup R(t),
\]

where \( C|_t \) is the union of class hierarchies \( C \) at time \( t \in T \). Below by **attributes**, we mean the union of variables, relationships, and methods of a class.

#### 5.1. Schema Consistency

Before focusing on schema modification, we present the definitions and axioms defined in our model for keeping the schema in a consistent stage at any time \( t \in T \).

**DEFINITION 14.** The *full inherited attributes* of a class \( c \in C \), \( FA(c) \) is the union of the attributes defined by all the superclasses of \( c \) in the case of \( \text{sup}(c) \neq \emptyset \). In the case that \( \text{anc}(c) \neq \emptyset \), then \( FA(c) \) is a subset of the attributes of the class whose class identifier is \( \text{anc}(c) \), regardless of whether \( \text{sup}(c) = \emptyset \) or \( \neq \emptyset \).

When two common attributes are inherited from multiple superclasses, a conflict arises and some form of conflict resolution must be performed. Since our interest at the moment is on the semantic of the model, rather than on its implementation, we assume that conflict attributes are solved.

The following axioms are added to keep the consistency of the Schema.
Acyclicity

There are no cycles in \((C(P), \leq_P, \preceq_P)\). This follows directly from the axioms of partial order.

Ownership

The own attributes of a class \(c \in C\), \(OA(c)\), are those attributes that are directly defined in \(c\).

Override

The overriding attributes of a class \(c \in C\), \(RA(c) \equiv FA(c) \cap OA(c)\). In other words, \(RA(c)\) are those attributes that are at the same time defined in \(c\) and inherited from any of its superclasses.

Inheritance

The inherited attributes of a class \(c \in C\), \(IA(c) \equiv FA(c) \setminus RA(c) = FA(c) \setminus OA(c)\). Thus, in the case of equally named attributes which are defined inside the class \(c\) and simultaneously inherited from another class \(c' \in C\), those defined directly inside a class hide the inherited ones. One can note that \(OA(c)\) and \(IA(c)\) are disjoint sets.

Interface

The interface \(Int(c)\) of a class \(c \in C \equiv IA(c) \cup OA(c)\). In this way, the interface groups the specifications of all the attributes of \(c\) to which the objects of class \(c\), that is, \(E_t(c)\), can respond. All attributes within a class have different names and different origins.

5.2. Semantics of Modification

Typical schema modification includes adding and deleting classes, adding and deleting inheritance relationships between classes, and adding and deleting attributes of a class. Table 2 describes the set of operators that can be performed on the schema elements in TVOO to support its modifications. Although our set of operators correspond with the widely accepted classification of schema changes given in [5], the semantic we proposed is different because of the temporal versioned dimension of our model. One can note that the first row, i.e., the operations related with a class \(c \in C\) effect directly the structure of \((C(P), \leq_P)\), while the rest effect the class definition itself and so \((C(P), \preceq_P)\). The operations related with object types, methods, and relationships are done by modifying the class containing them.

Table 2. Schema changes.

<table>
<thead>
<tr>
<th>Schema Element</th>
<th>Operation</th>
<th>Name</th>
<th>Notation</th>
<th>Add</th>
<th>Delete</th>
<th>Modify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td></td>
<td></td>
<td>(c \in C)</td>
<td>addClas</td>
<td>delClas</td>
<td>namClas</td>
</tr>
<tr>
<td>Object type</td>
<td></td>
<td></td>
<td>(y \in OY)</td>
<td>addVar</td>
<td>delVar</td>
<td>namVar</td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
<td>(m \in M)</td>
<td>addMet</td>
<td>delMet</td>
<td>namMet</td>
</tr>
<tr>
<td>Relationship</td>
<td></td>
<td></td>
<td>(r \in R)</td>
<td>addRel</td>
<td>delRel</td>
<td>namRel</td>
</tr>
</tbody>
</table>

5.2.1. Class version modification

As it was mentioned before, in TVOO an object \(o = (\cdot, v = (c, c_t, *), \cdot, \cdot, \cdot)\) whose version belongs to a class \(c \in C\) can be accessed from any class \(c'\) such that \(p' \in P(C)^{p_0}\), where \(p_0\) is the minimal element of \((\cdot, p)^{P(C)}\) and the elements \(p, p'\) correspond to the classes \(c, c'\),
respectively. Therefore, whenever a class modification is done, a pair of functions which perform the transformations between its instances must be defined as well. Let \( m_r \) be a function which perform a retrospective transformation of the objects, that is, it describes how to convert an object \( o = (\cdot, v = (c, c_t, *), \cdot, \cdot, \cdot) \) into an object \( o' = (\cdot, v = (c', c_t', *), \cdot, \cdot, \cdot) \), where \( c' \equiv \text{anc}(c) \). Also, let \( m_p \) be the inverse function of \( m_r \), that is, the function which describes how to convert \( o' = (\cdot, v = (c', c_t', *), \cdot, \cdot, \cdot) \) into \( o = (\cdot, v = (c, c_t, *), \cdot, \cdot, \cdot) \). As methods (see Section 4.1.6), \( m_r \) and \( m_p \) must be specified by the designer of the data base in a language-dependent manner.

Any version modification occurring on a class \( c \in C \) (and this can be done only if the class \( c \) is "alive") results in the generation of a new version \( c_1 \) of that class. Since a change on a class can affect its subclasses, new versions for the subclasses have to be generated. To be more precise (see Figure 1), let \( p \) be the element of \( P \equiv P(C) \in \text{CTem}_t \) corresponding to the class \( c \) (Figure 1a). We denote by \( *P \) (to be distinguished from \( PP \)) the set \( \{ q : q \geq p \} \) which is a subset of \( P \) and let \( *PP_+ \) be the "alive" elements of \( *P \) at time \( t \in T \). When one modifies the class \( c \) to a class \( c' \) and thus its corresponding element \( p' \in P \) (Figure 1(b1)), one has to generate modifications of the elements in \( *PP_+ \setminus \{ p \} \) and also generate the necessary structure between these elements as can be seen from Figure 1(b2). In the diagrams, \( \rightarrow \) means \( < \) and \( \Rightarrow \) means \( \leq \).

5.2.2. Class inheritance modification

When a new class \( c' \) is defined to be a subclass of any other \( c \in C \in \mathcal{C}_t \) (and this can be done only if the class \( c \) is "alive"), then similar to Section 5.2.1, subclasses to the subversions of the class \( c \) has to be generated. To be more precise, let \( p \) be the element of \( P \equiv P(C) \in \text{CTem}_t \) corresponding to the class \( c \). Let \( *PP_+ \) be the "alive" elements of \( *P \) at time \( t \in T \). When one defines a subclass \( c' \) and thus its corresponding element \( p' \in P \) of the class \( c \), one has to generate

![Diagram](image-url)
subclasses for the elements in $P^{n, o} \setminus \{p\}$ and also the necessary structure between these elements as can be seen after interchanging $\rightarrow$ and $\rightarrow$ in Figure 1.

Let $c \in C$ and let $p \in P = P(C) \subset \mathcal{T}_{\text{element}}$ be its corresponding elements. If the class $c$ is deleted, then together with $p$ one deletes also the elements of $P^{n, o}$ (and so also the corresponding classes). If $\text{des}(c) \neq \emptyset$, then the classes of $\text{des}(c)$ are promoted. By this we understand that any $c' \in \text{anc}(c)$ (respectively, $\in \text{sup}(c)$) becomes a member of $\text{anc}(c')$ (respectively, $\text{sup}(c')$) for any $c' \in \text{des}(c)$. An example is given in Figure 1. Later if the designer of the database do not need these subversions as part of the schema, he can delete them.

6. CONCLUSIONS

Schema versioning is one of a number of related areas dealing with the same general problem of using multiple heterogeneous data schemas in a database. Although there is a lot of research in this field, not a lot of work has been done on a formal model to deal with schema versioning. This paper presents a formal model that combines in a natural way object-oriented technology with temporal and versioning concepts. The notion of temporal and versioned types were introduced and the notion of object and schema versioning were defined. In our model objects can be queried under any schema version in the sense that not only objects created under old versions of schema elements can be queried from new versions, but also objects created by new schema element versions can be queried from old versions.

This model of versions has been developed using the C++ binding of Shore [24] Beta-release version 1.0 on a Solaris 2.5 platform. At the moment, we are testing the effectiveness of the proposed model and in the near future the results will be shown. We need to study the object transformation mechanism in detail and define the integrity constraint method which deals with past histories of objects as well as its query implications. Finally, the implementation performance of the system under development, which in most scenarios is very important, have to be investigated.

REFERENCES


18. B.S. Lerner and N. Habermann, Beyond schema evolution to database reorganization, in Proceedings of the Joint Conference on Object-Oriented Systems, Languages and Applications (OOPSLA) and ECOOP’91, pp. 67–76. (1990).


21. Young-Gook Ra and E.A. Rundensteiner, A transparent schema-evolution system based on object-oriented view technology, IEEE Transactions on Knowledge and Data Engineering 9 (4), 600–623 (July/August 1997).


