Managing Schema Evolution using a Temporal Object Model

Iqbal A. Goralwalla, Duane Szafron, M. Tamer Özsu
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
\{iqbal,duane,ozsu\}@cs.ualberta.ca

and

Randal J. Peters
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
randal@cs.umanitoba.ca

Abstract

The issues of schema evolution and temporal object models are generally considered to be orthogonal and are handled independently. This is unrealistic because to properly model applications that need incremental design and experimentation (such as CAD, software design process), the evolutionary histories of the schema objects should be traceable. In this paper we propose a method for managing schema changes by exploiting the functionality of a temporal object model. The result is a uniform treatment of schema evolution and temporal support for many object database management systems applications that require both.

1 Introduction

In this paper we address the issue of schema evolution and temporal object models. These two issues are generally considered to be orthogonal and are handled independently. However, many object database management systems (ODBMS) applications require both. For example:

- The results reported in [Sjo93] illustrate the extent to which schema changes occur in real-world database applications such as health care management systems. Such systems also require a means to represent, store, and retrieve the temporal information in clinical data [KFT91, DM94, CPP95].

- The engineering and design oriented application domains (e.g., CAD, software design process) require incremental design and experimentation [KBCG90, GTC+90]. This usually leads to
frequent changes to the schema over time which need to be retained as historical records of
the design process.

We propose a method for managing schema changes by exploiting the functionality of a temporal
object model. The provision of time in an object model establishes a platform from which tempo-
rality can be used to investigate advanced database features such as schema evolution. Given that
the applications supported by ODBMSs need support for incremental development and experimen-
tation with changing and evolving schema, a temporal domain is a natural means for managing
changes in schema and ensuring consistency of the system. The result is a uniform treatment of
schema evolution and temporal support for many ODBMS applications that require both.

Schema evolution using time is the process of allowing changes to schema without loss of in-
formation. Typical schema changes include adding and dropping behaviors (properties) defined on
a type, adding and dropping subtype relationships between types, to name a few. Using time to
maintain and manage schema changes gives substantial flexibility in the software design process.
It enables the designers to retrieve the interface of a type that existed at any time in the design
phase, reconstruct the super(sub)-lattice of a type as it was at a certain time (and subsequently
the type lattice of the object database at that time), and trace the implementations of a certain
behavior in a particular type over time.

A typical schema change can affect many aspects of a system. There are two fundamental
problems to consider:

1. **Semantics of Change.** The effects of the schema change on the overall way in which the
   system organizes information (i.e., the effects on the schema). The traditional approach to
   solving this problem is to define a set of invariants that must be preserved over schema
   modifications.

2. **Change Propagation.** The effects of the schema change on the consistency of the underlying
   objects (i.e., the propagation of the schema changes to the existing object instances). The
   traditional approach of solving this is to coerce objects to coincide with the new definition of
   the schema.

In this paper we primarily consider the consistent handling of the problem of semantics of change
using a temporal ODBMS. We describe the necessary modifications that could occur on the schema,
and show how the implications of the modifications are managed. Our work is conducted within the
context of the TIGUKAT\textsuperscript{1} temporal ODBMS [ÖPS\textsuperscript{+}95, GLÖS95, GÖS97] that is being developed at the University of Alberta. However, the results reported here extend to any ODBMS that uses time to model evolution histories of objects.

The remainder of the paper is organized as follows. In Section 2, we examine some of the previous work on schema evolution. In Section 3, we give a brief overview of the TIGUKAT temporal object model with an emphasis on how histories of objects are maintained. In Section 4, we describe the schema changes that can occur in TIGUKAT, and how they are managed using a temporal object model. In Section 5, we give examples of queries that allow software designers to retrieve schema objects at any time in their evolution histories. Concluding remarks and results of the paper are summarized in Section 6.

2 Related Work

The issue of schema evolution has been an area of active research in the context of ODBMSs [BKKK87, KC88, PS87, NR89]. In many of the previous work, the usual approach is to define a set of invariants that must be preserved over schema modifications in order to ensure consistency of the system. The Orion [BKKK87, KC88] model is the first system to introduce the invariants and rules approach as a more structured way of describing schema evolution in ODBMSs. Orion defines a complete set of invariants and a set of accompanying rules for maintaining the invariants over schema changes. The work of Smith and Smith [SS77] on aggregation and generalization sets the stage for defining invariants when subtypes and supertypes are involved. Changes to schema in previous works are corrective in that once the schema definitions are changed, the old definitions of the schema are no longer traceable. In TIGUKAT, a set of invariants similar to those given in [BKKK87] is defined. However, changes to the schema are not corrective. The provision of time in TIGUKAT establishes a natural foundation for keeping track of the changes to the schema. This allows applications, such as CAD, to trace their design over time and make revisions, if necessary.

There have been many temporal object model proposals (for example, [RS91, SC91, WD92, KS92, CITB92, BFG96]). In handling temporal information, these models have focussed on managing the evolution of real-world entities. The implicit assumption in these models is that the schema of the object database is static and remains unchanged during the lifespan of the object.

\textsuperscript{1}TIGUKAT (tee-goo-kaht) is a term in the language of Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.
database. More specifically, the evolution of schema objects (i.e., types, behaviors, etc) is considered to be orthogonal to the temporal model. However, given the kinds of applications that an ODBMS is expected to support, we have exploited the underlying temporal domain in the TIGUKAT temporal model as a means to support schema evolution.

In the context of relational temporal models, Ariav [Ari91] examines the implications of allowing data structures to evolve over time in a temporal data model, identifies the problems involved, and establishes a platform for their discussion. McKenzie and Snodgrass [MS90] develop an algebraic language to handle schema evolution. The language includes functions that help track the schema that existed at a particular time. Schema definitions can be added, modified, or deleted. Apart from the addition and removal of attributes, the nature of the modifications to the schema and their implications are not demonstrated. Roddick [Rod91] investigates the incorporation of temporal support within the metadatabase to accommodate schema evolution. In [Rod92], SQL/SE, an SQL extension that is capable of handling schema evolution in relational database systems is proposed using the ideas presented in [Rod91]. The approach used in the TIGUKAT temporal object model is similar in the sense that temporal support of real-world objects is extended in a uniform manner to schema objects, and then used to support schema evolution. Some of the ideas in [Rod91, Rod92, Rod95] have been carried forward in the design of the TSQL2 temporal query language [Sno95].

Skarra and Zdonik [SZ86, SZ87] define a framework within the Encore object model for versioning types as a support mechanism for changing type definitions. A type is organized as a set of individual versions. This is known as the version set of the type. Every change to a type definition results in the generation of a new version of the type. Since a change to a type can also affect its subtypes, new versions of the subtypes may also be generated. This approach provides fine granularity control over schema changes, but may lead to inefficiencies due to the creation of a new version of the versioned part of an object every time a single attribute changes its value. In our approach, any changes in type definitions involve changing the history of certain behaviors to reflect the changes. For example, adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is still accessible at a time before the change was made. This alleviates the need of creating new versions of a type each time any change is made to a type.
3 The TIGUKAT Temporal Object Model

3.1 Fundamentals of TIGUKAT Object Model

The TIGUKAT object model [Pet94, ÖPS95] is purely behavioral with a uniform object semantics. The model is behavioral in the sense that all access and manipulation of objects is based on the application of behaviors to objects. The model is uniform in that every component of information, including its semantics, is modeled as a first-class object with well-defined behavior. Other typical object modeling features supported by TIGUKAT include strong object identity, abstract types, strong typing, complex objects, full encapsulation, multiple inheritance, and parametric types.

The primitive objects of the model include: atomic entities (reals, integers, strings, etc.); types for defining common features of objects; behaviors for specifying the semantics of operations that may be performed on objects; functions for specifying implementations of behaviors over types; classes for automatic classification of objects based on type\(^2\); and collections for supporting general heterogeneous groupings of objects. Figure 1 shows a simple type lattice that will be used to illustrate the concepts introduced in the rest of the paper.

![Figure 1: Simple type lattice.](image)

In this paper, a reference prefixed by "T_" refers to a type, "C_" to a class, "B_" to a behavior, and "T_X< T_Y >" to the type T_X parameterized by the type T_Y. For example, T_person refers to a type, C_person to its class, B_age to one of its behaviors and T_collection< T_person > to the type of collections of persons. A reference such as joe, without a prefix, denotes some other application specific reference. The type T_null in TIGUKAT binds the type lattice from the bottom (i.e., most defined type), while the T_object type binds it from the top (i.e., least defined type). T_null is introduced to provide, among other things, error handling and null semantics for the model.

The access and manipulation of an object's state occurs exclusively through the application\(^2\)Types and their extents are separate constructs in TIGUKAT.

\(\text{To appear in the Sixteenth International Conference on Conceptual Modeling (ER'97), November 1997.}\)
of behaviors. We clearly separate the definition of a behavior from its possible implementations (functions). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types. This provides direct support for behavior overloading and late binding of functions (implementations) to behaviors.

3.2 The Temporal Extensions

The philosophy behind adding temporality to the TIGUKAT object model is to accommodate multiple applications that have different type semantics requiring various notions of time [LGÖS97, GÖS97]. Consequently, the TIGUKAT temporal object model consists of an extensible set of primitive time types with a rich set of behaviors to model time. The only part of the temporal model that is relevant to this paper is the management of event histories. Therefore, we focus on history management and details of other aspects can be found in [GLÖS95, GLÖS96].

Our model represents the temporal histories of real-world objects whose type is T.X as objects of the T.history(T.X) type. For example, suppose a behavior B.salary is defined in the T.employee type. Now, to keep track of the changes in salary of employees, B.salary would return an object of type T.history(T.real) which would consist of the different salary objects of a particular employee and their associated time periods.

A temporal history consists of objects and their associated timestamps (time intervals or time instants). One way of modeling a temporal history would be to define a behavior that returns a collection of <timestamp, object> pairs. However, instead of structurally representing a temporal history in this manner, we use a behavioral approach by defining the notion of a timestamped object. A timestamped object knows its timestamp (time interval or time instant) and its associated value at (during) the timestamp. A temporal history is made up of such objects. The following behaviors are defined on the T.history(T.X) type:

\[
\begin{align*}
B_{\text{history}} & : \ T.collection(T.timeStampedObject(T.X)) \\
B_{\text{timeline}} & : \ T.timeline \\
B_{\text{insert}} & : \ T.X, T.timeStamp \to \\
B_{\text{remove}} & : \ T.X, T.timeStamp \to \\
B_{\text{validObjects}} & : \ T.timeStamp \to T.collection(T.timeStampedObject(T.X)) \\
B_{\text{validObject}} & : \ T.timeStamp \to T.timeStampedObject(T.X)
\end{align*}
\]

Behavior \(B_{\text{history}}\) returns the set (collection) of all timestamped objects that comprise the history. A history object also knows the timeline it is associated with and this timeline is returned.
by the behavior $B_{\text{timeline}}$. The timeline basically orders the timestamps of timestamped objects [GLÖS96]. The $B_{\text{insert}}$ behavior accepts an object and a timestamp as input and creates a timestamped object that is inserted into the history. Behavior $B_{\text{remove}}$ drops a given object from the history at a specified timestamp. The $B_{\text{validObjects}}$ behavior allows the user to get the objects in the history that were valid at (during) a given timestamp. Behavior $B_{\text{validObject}}$ is derived from $B_{\text{validObjects}}$ to return the timestamped object that exists at a given time instant.

Each timestamped object is an instance of the $T_{\text{timeStampedObject(TX)}}$ type. This type represents objects and their corresponding timestamps. Behaviors $B_{\text{value}}$ and $B_{\text{timeStamp}}$ defined on $T_{\text{timeStampedObject}}$ return the value and the timestamp (time interval or time instant) of a timestamped object, respectively.

**Example 3.1** Suppose the type $T_{\text{patient}}$ shown in Figure 1 represents different patients in a hospital. To represent a patient's blood test history over the course of a particular illness, the behavior $B_{\text{bloodTests}}$ is defined on $T_{\text{patient}}$ to return an object of type $T_{\text{history(T.bloodTest)}}$. Each blood test is represented by an object of the type $T_{\text{bloodTest}}$. Therefore, the history of the different blood tests undertaken by $\text{joe}$ (an instance of $T_{\text{patient}}$) would then be retrieved using the behavior application $\text{joe}.B_{\text{bloodTests}}$. Let us call this history object $\text{bloodTestHistory}$. Now, suppose $\text{joe}$ was suspected of having septicemia\(^3\) and had diagnostic hematology and microbiology blood tests on 15 January 1995. As a result of a raised white cell count, $\text{joe}$ was given a course of antibiotics while the results of the tests were pending. A repeat hematology test was ordered on 20 February 1995. To record these tests, three objects with type $T_{\text{bloodTest}}$ were created and then entered into the object database using the following TIGUKAT behavior applications:

\[
\begin{align*}
\text{bloodTestHistory}.B_{\text{insert}}(\text{microbiology}, 15 \text{ January 1995}) \\
\text{bloodTestHistory}.B_{\text{insert}}(\text{hematology1}, 15 \text{ January 1995}) \\
\text{bloodTestHistory}.B_{\text{insert}}(\text{hematology2}, 20 \text{ February 1995})
\end{align*}
\]

If subsequently there is a need to determine which blood tests $\text{joe}$ took in January 1995, this would be accomplished by the following behavior application:

\[
\text{bloodTestHistory}.B_{\text{validObjects}}([1 \text{ January 1995}, 31 \text{ January 1995}])
\]

This would return a collection of the two timestamped objects, \{$T_{\text{timeStampedMicrobiology}}, T_{\text{timeStampedHematology1}}$\}, representing the blood tests $\text{joe}$ took in January 1995. The first timestamped

\(^3\)An infection of the blood.
object would have `microbiology` as its value and the second would have `hematology1` as its value. To assist in clarifying the contents and structure of a history object, we give a pictorial representation of `bloodTestHistory` in Figure 2.

![Diagram of blood test history](image)

**Figure 2:** A pictorial representation of a patient’s blood test history.

In the figure, the boxes shaded in grey are objects. Objects have an outgoing edge labeled by each applicable behavior that leads to the object resulting from the application of the behavior. For example, applying the behavior `B.timeline` to the object `bloodTestHistory` results in the object `bloodTestTimeline`. A circle labeled with the symbols `{}` represents a collection object and has outgoing edges labeled with “∈” to each member of the collection. For example, applying the `B.history` behavior to the object `bloodTestHistory` results in a collection object whose members are the timestamped objects `timeStampedMicrobiology`, `timeStampedHematology1`, and `timeStampedHematology2`. Finally, the `B.insert` behavior updates the blood test history (`bloodTestHistory`) when given an object of type `T.bloodTest` and a timestamp. Similarly, the `B.validObjects` behavior returns a collection of timestamped blood test objects when given a timestamp.

---

4It should be noted that although we have two different timestamped objects containing the values `microbiology` and `hematology1`, they both contain the same timestamp. That is, although `timeStampedMicrobiology.B.value = microbiology` and `timeStampedHematology1.B.value = hematology1`, `timeStampedMicrobiology.B.timestamp = timeStampedHematology1.B.timestamp = 15 January 1995.`

---

4 Management of Schema Evolution by the Temporal Object Model

4.1 Schema Related Changes

There are different kinds of objects modeled by TIGUKAT, some of which are classified as schema objects. These objects fall into one of the following categories: type, class, behavior, function, and collection. There are three kinds of operations that can be performed on schema objects: add, drop and modify. Table 1 shows the combinations between the various schema object categories and the different kinds of operations that can be performed in TIGUKAT [Pet94, PÖ97]. The bold entries represent combinations that implicate schema changes while the emphasized entries denote non-schema changes.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Add (A)</th>
<th>Drop (D)</th>
<th>Modify (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type (T)</td>
<td>subtyping</td>
<td>type deletion</td>
<td>add behavior (AB) drop behavior (DB) add supertype link (ASL) drop supertype link (DSL)</td>
</tr>
<tr>
<td>Class (C)</td>
<td>class creation</td>
<td>class deletion</td>
<td>extent change</td>
</tr>
<tr>
<td>Behavior (B)</td>
<td>behavior definition</td>
<td>behavior deletion</td>
<td>change association (CA)</td>
</tr>
<tr>
<td>Function (F)</td>
<td>function definition</td>
<td>function deletion</td>
<td>implementation change</td>
</tr>
<tr>
<td>Collection (L)</td>
<td>collection creation</td>
<td>collection deletion</td>
<td>extent change</td>
</tr>
</tbody>
</table>

Table 1: Classification of schema changes.

In the context of a temporal model, adding refers to creating the object and beginning its history, dropping refers to terminating the history of an object, and modifying refers to updating the history of the schema object. Since type-related changes form the basis of most other schema changes, we describe the modifications that affect the type schema objects. Type modification (depicted at the intersection of the M column and T row in Table 1) includes several kinds of type changes. They are separated into changes in the behaviors of a type (depicted as MT-AB and MT-DB in Table 1) and changes in the relationships between types (depicted as MT-ASL and MT-DSL in Table 1). Invariants for maintaining the semantics of schema modifications in TIGUKAT are described in [Pet94, PÖ97]. The invariants are used to gauge the consistency of a schema change in that the invariants must be satisfied both before and after a schema change is performed.

The meta-model of TIGUKAT is uniformly represented within the object model itself, providing reflective capabilities [PÖ93]. One result of this uniform approach is that types are objects and
they have a type (called T.type) that defines their behaviors. T.type defines behaviors to access a type’s interface (B.interface), its subtypes (B.subtypes), its supertypes (B.supertypes), plus many others that are not relevant for the scope of this paper. Since types are objects with well-defined behaviors, the approach of keeping track of the changes to a type is the same as that for keeping track of the changes to objects discussed in Section 3.2. This is one of the major advantages of the uniformity of the object model. The semantics of the changes to a type are discussed in the following sections.

4.2 Changing Behaviors of a Type

Every type has an interface which is a collection of behaviors that are applicable to the objects of that type. A type’s interface can be dichotomized into two disjoint subsets:

1. the collection of native behaviors which are those behaviors defined by the type and are not defined on any of its supertypes;

2. the collection of inherited behaviors which are those behaviors defined natively by some supertype and inherited by the type.

There are three behaviors defined on T.type to return the various components of a type’s interface: B.native returns the collection of native behaviors, B.inherited returns the inherited behaviors and B.interface returns the entire interface of the type.

Types can evolve in different ways. One aspect of a type that can change over time is the behaviors in its interface (i.e., adding or deleting behaviors). To keep track of this aspect of a type’s evolution, we define histories of interface changes by extending the interface behaviors with time-varying properties. The definition of the extended behaviors are as follows:

\[
\begin{align*}
    B_{\text{native}} & : T.\text{history}(T.\text{collection}(T.\text{behavior})) \\
    B_{\text{inherited}} & : T.\text{history}(T.\text{collection}(T.\text{behavior})) \\
    B_{\text{interface}} & : T.\text{history}(T.\text{collection}(T.\text{behavior}))
\end{align*}
\]

Each behavior now returns a collection of a collection of timestamped behaviors. Adding a new behavior to a type changes the history of the type’s interface to include the new behavior. The old interface of the type is still accessible at a time before the change was made.

Note that we do not need to explicitly maintain separate histories for each of these behaviors. For example, in an implementation we can choose to only maintain the native behaviors of a type.
The entire interface of a type can be derived by unioning the native behaviors of all the supertypes of the type. The inherited behaviors can be derived by taking the difference of the interface and the native behaviors of the type. As another alternative, we may choose to maintain the interface of a type and derive the native and inherited behaviors. In this approach, the native behaviors of a type can be derived by unioning the interfaces of the direct supertypes and subtracting the result from the interface of the type. The inherited behaviors can be derived in the same way as above.

With the time-varying interface extensions, we can determine the various aspects of a type's interface at any time of interest. For example, Figure 3 shows the history of the entire interface for the type $T_{person}$.

![Diagram](image)

Figure 3: Interface history of type $T_{person}$.

At time $t_0$, behaviors $B_{name}$, $B_{birthDate}$, and $B_{age}$ are defined on $T_{person}$ and the initial history of $T_{person}$'s interface is $\{<t_0, \{B_{name}, B_{birthDate}, B_{age}\}>\}$. At time $t_5$, behavior $B_{spouse}$ is added to $T_{person}$. To reflect this change, the interface history is updated to $\{<t_0, \{B_{name}, B_{birthDate}, B_{age}\}>, <t_5, \{B_{name}, B_{birthDate}, B_{age}, B_{spouse}\}>\}$. This shows that between $t_0$ and $t_5$ only behaviors $B_{name}$, $B_{birthDate}$, and $B_{age}$ are defined and at $t_5$ behaviors $B_{name}$, $B_{birthDate}$, $B_{age}$, $B_{spouse}$ exist. Next, at time $t_{10}$, behavior $B_{age}$ is dropped from type $T_{person}$ and at the same time behavior $B_{children}$ is added. The final history of the interface of $T_{person}$ after this change is $\{<t_0, \{B_{name}, B_{birthDate}, B_{age}\}>,$
$t_5, \{B\text{.name}, B\text{.birthDate}, B\text{.age}, B\text{.spouse}\}, t_{10}, \{B\text{.name}, B\text{.birthDate}, B\text{.spouse}, B\text{.children}\}$\)\(^5\). The native and inherited behaviors would contain similar histories. Using this information, we can reconstruct the interface of a type at any time of interest. For example, at time $t_3$ the interface of type $T\text{.person}$ was $\{B\text{.name}, B\text{.birthDate}, B\text{.age}\}$, at time $t_5$ it was $\{B\text{.name}, B\text{.birthDate}, B\text{.age}, B\text{.spouse}\}$, and at time $t_{10}$ (now) it is $\{B\text{.name}, B\text{.birthDate}, B\text{.spouse}, B\text{.children}\}$.

The behavioral changes to types include the MT-AB and MT-DB entries of Table 1. These changes affect various aspects of the schema and have to be properly managed to ensure consistency of the schema.

**Modify Type - Add Behavior (MT-AB).** This change adds a native behavior $b$ to a type $T$ at time $t$. The MT-AB change has the following effects:

- The histories of the native and interface behaviors of type $T$ need to be updated. The behavior applications $T\text{.}B\text{.native}.B\text{.insert}(b, t)$ and $T\text{.}B\text{.interface}.B\text{.insert}(b, t)$ perform this update. For example, the behavior application $T\text{.}B\text{.person}.B\text{.interface}.B\text{.insert}(B\text{.spouse}, t_5)$ updates the interface history of $T\text{.person}$ when behavior $B\text{.spouse}$ is added to $T\text{.person}$ at time $t_5$.

- The implementation history of behavior $b$ needs to be updated to associate it with some function $f$. This is achieved by the behavior application $b.B\text{.implementation}.B\text{.insert}(f, t)$ (details on implementation histories of behaviors are given in Section 4.3). For example, if the function associated with behavior $B\text{.spouse}$ is the stored function $s\text{.spouse}$, then the implementation history of $B\text{.spouse}$ is updated using the behavior application $B\text{.spouse}.B\text{.implementation}.B\text{.insert}(s\text{.spouse}, t_5)$.

- The history of inherited and interface behaviors of all subtypes of type $T$ needs to be adjusted. That is,

$$\forall T' \mid T' \text{ subtype-of } T, T'.B\text{.inherited}.B\text{.insert}(b, t) \text{ and } T'.B\text{.interface}.B\text{.insert}(b, t)$$

For example, the histories of inherited and interface behaviors of types $T\text{.employee}$ and $T\text{.patient}$ (see Figure 1) need to be adjusted to reflect the addition of behavior $B\text{.spouse}$ in type $T\text{.person}$ at time $t_5$. For the $T\text{.employee}$ type, this is accomplished using the

---

\(^5\)Note that in Figure 3 objects that are repeated in the timestamped collections are actually the same object. For example, the $B\text{.name}$ object in all three timestamped collections is the same object. It is shown three times in the figure for clarity.

---

behavior applications $T_{employee}.B_{interface}.B_{insert}(B_{spouse},t_5)$ and $T_{employee}.B_{inherited}.B_{insert}(B_{spouse},t_5)$. Similar behavior applications are carried out for $T_{patient}$.

Modify Type - Drop Behavior (MT-DB). This change drops a native behavior $b$ from a type $T$ at time $t$. When a behavior is dropped, its native definition is propagated to the subtypes unless the behavior is inherited by the subtype through some other chain. In this way, as with the supertypes, the subtypes of a type also retain their original behaviors. Thus, only the single type involved in the operation actually drops the behavior and the overall interface of the subtypes and supertypes are not affected by the change. Many behavior inheritance semantics are possible. One such semantics is that when a native behavior is dropped from a type, all subtypes retain that behavior. This means that if another supertype of the subtype defines this behavior, there is no change. Otherwise, the behavior in the subtype moves from the inherited set to the native set. This is the semantics we are modeling in this paper. If any other behavior inheritance semantics are used, appropriate changes can easily be made to the temporal histories. The MT-DB change has the following effects:

- The native behaviors history of type $T$ changes. The behavior application $T.B_{native}.B_{remove}(b,t)$ performs this update. For example, the behavior application $T_{person}.B_{native}.B_{remove}(B_{age},t_{10})$ updates the history of native behaviors of $T_{person}$ when the behavior $B_{age}$ is dropped from type $T_{person}$.

- The native and inherited behavior histories of the subtypes of $T$ (possibly) change. For example, the behavior applications $T_{employee}.B_{native}.B_{insert}(B_{age},t_{10})$ and $T_{employee}.B_{inherited}.B_{remove}(B_{age},t_{10})$ add behavior $B_{age}$ to the native behaviors of $T_{employee}$, and drop behavior $B_{age}$ from the inherited behaviors of $T_{employee}$ respectively, when $B_{age}$ is dropped from $T_{person}$ at $t_{10}$. This is because $B_{age}$ is not inherited by $T_{employee}$ through any other chain. If $B_{age}$ was inherited by $T_{employee}$ from some other supertype, nothing would change. Similar behavior applications are carried out for type $T_{patient}$.

4.3 Changing Implementations of Behaviors

Each behavior defined on a type has a particular implementation for that type. The $B_{implementation}$ behavior defined on $T_{behavior}$ is applied to a behavior, accepts a type as an argument and returns
the implementation (function) of the receiver behavior for the given type. In order to model the aspect of schema evolution that deals with changing the implementations of behaviors on types, we maintain a history of implementation changes by extending the \( B_{\text{implementation}} \) behavior with time-varying properties. The definition of the extended behavior is as follows:

\[
B_{\text{implementation}} : T_{\text{type}} \rightarrow T_{\text{history}}(T_{\text{function}})
\]

With this behavior we can determine the implementation of a behavior defined on a type at any time of interest. For example, Figure 4 shows the history of the implementations for behavior \( B_{\text{age}} \) on type \( T_{\text{person}} \). There are two kinds of implementations for behaviors [Pet94]. A computed function consists of runtime calls to executable code and a stored function is a reference to an existing object in the object database.

Figure 4: Implementation history of behavior \( B_{\text{age}} \) on type \( T_{\text{person}} \).

In Figure 4, we use \( c_i \) to denote a computed function, \( s_i \) to denote a stored function. At time \( t_2 \), the implementation of \( B_{\text{age}} \) changed from the computed function \( c_1 \) to the computed function \( c_3 \). At time \( t_4 \), the implementation of \( B_{\text{age}} \) changed from the computed function \( c_3 \) to the stored function \( s_1 \). All these changes are reflected in the implementation history of behavior \( B_{\text{age}} \), which is \( \{<t_0, c_1>, <t_2, c_3>, <t_4, s_1>\} \).

Using the results of this section and Section 4.2, we can reconstruct the behaviors, their implementations and the object representations\(^6\) for any type at any time \( t \). For example, the interface

\(^6\)Stored functions associated with behaviors allow us to reconstruct object representations (i.e., states of objects)
of type $T\text{person}$ at time $t_3$ is given by the behavior application $T\text{person}[t_3]B\text{interface}$ which results in $\{B\text{name}, B\text{birthDate}, B\text{age}\}$, as shown in Figure 3. We use the syntax $o[t]b$ to denote the application of behavior $b$ to object $o$ at time $t$. The implementation of $B\text{age}$ at time $t_3$ is given by $B\text{age}[t_3]B\text{implementation}(T\text{person})$ which is $c_3$, as shown in Figure 4.

In this paper we are assuming that there is no implementation inheritance. That is, if the binding of a behavior to a function changes in a type, the bindings of that behavior in the subtypes are unaffected. If implementation inheritance is desired, it can easily be modeled by temporal histories similarly to behavioral inheritance.

4.4 Changing Subtype/Supertypes of a Type

In Section 4.2 we described how the changes in a type's interface was one aspect in which a type evolves. Another aspect of a type that can change over time is the relationships between types. These include adding a direct supertype link and dropping a direct supertype link. The $B\text{supertypes}$ and $B\text{subtypes}$ behaviors defined on $T\text{type}$ return the direct supertypes and subtypes of the receiver type, respectively. In order to model the structure of the type lattice through time, we define histories of supertype and subtype changes of a type by extending the $B\text{supertypes}$ and $B\text{subtypes}$ behaviors with time-varying properties:

$$B\text{supertypes} : T\text{history}(T\text{collection}(T\text{type}))$$
$$B\text{subtypes} : T\text{history}(T\text{collection}(T\text{type}))$$

Using the $B\text{supertypes}$ and $B\text{subtypes}$ behaviors, we can reconstruct the structure of a type's supertype and subtype lattice at any time of interest. To facilitate this, the derived behaviors $B\text{superlattice}$ and $B\text{sublattice}$ are defined on $T\text{type}$:

$$B\text{superlattice} : T\text{history}(T\text{poset}(T\text{type}))$$
$$B\text{sublattice} : T\text{history}(T\text{poset}(T\text{type}))$$

The behavior $B\text{superlattice}$ is derived by recursively applying $B\text{supertypes}$ until $T\text{object}$ is reached, while the behavior $B\text{sublattice}$ is derived by recursively applying $B\text{subtypes}$ until $T\text{null}$ is reached. In both cases, the intermediate results are partially ordered. Figure 5 shows the supertype lattice history for type $T\text{employee}$.

for any type at any time $t$. This is useful in propagating changes to the underlying object instances. In this paper however, we are concerned primarily with the effects of schema changes on the schema itself.

At time $t_0$, the superlattice history of type $T_{\text{employee}}$ included the types $T_{\text{person}}, T_{\text{taxSource}},$ and $T_{\text{object}}$. At time $t_5$, the supertype link between $T_{\text{employee}}$ and $T_{\text{taxSource}}$ is dropped. To reflect this change, the superlattice history of $T_{\text{employee}}$ is updated to $\{t_0, \{T_{\text{person}}, T_{\text{taxSource}}, T_{\text{object}}\}\}, \{t_5, \{T_{\text{person}}, T_{\text{object}}\}\}$.

The relationships between types include the MT-ASL and MT-DSL entries of Table 1. Similar to the behavioral changes to types discussed in Section 4.2, the relationships between types affect various aspects of the schema and have to be properly managed to ensure consistency of the schema.

**Modify Type - Add Supertype Link (MT-ASL).** Add a type, say $S$, as a direct supertype of another type, say $T$ at time $t$. The MT-ASL change has the following effects:

- The history of the collection of supertypes of type $T$ is updated. The behavior application $T.B_{\text{supertypes}}.B_{\text{insert}}(S, t)$ performs this update. The history of the superlattice of $T$ is adjusted accordingly. For example, adding the supertype link between $T_{\text{employee}}$ and $T_{\text{taxSource}}$ at $t_0$ necessitates an update to the history of supertypes for $T_{\text{employee}}$. This is done by the behavior application $T_{\text{employee}}.B_{\text{supertypes}}.B_{\text{insert}}(T_{\text{taxSource}}, t_0)$. The history of the direct supertypes of $T_{\text{employee}}$ would then be $\{t_0, \{T_{\text{taxSource}}\}\}$. 

---

To appear in the *Sixteenth International Conference on Conceptual Modeling (ER'97), November 1997.*
• The history of the collection of subtypes of type \( S \) is updated. The behavior application \( S.B_{\text{subtypes}}.B_{\text{insert}}(T, t) \) performs this update. The history of the sublattice of \( S \) is adjusted accordingly. In this case, the history of the collection of subtypes of \( T.\text{taxSource} \) has to be updated. This is done by the behavior application \( T.\text{taxSource}.B_{\text{subtypes}}.B_{\text{insert}}(T.\text{employee}, t_0) \). The history of the direct subtypes of \( T.\text{taxSource} \) would then be \( \{<t_0, \{T.\text{employee}\}>\} \).

• The behaviors of \( S \) are inherited by \( T \) and all the subtypes of \( T \). Therefore, the inherited behavior history of \( T \) and all subtypes of \( T \) is adjusted. The current behaviors of \( S \) are inherited by \( T \) and all subtypes of \( T \), and timestamped with \( t \) - the creation time of the supertype link.

\[
\forall b \in S.B_{\text{interface}}.B_{\text{history}}.B_{\text{last}}, \forall T' \mid T' \text{ subtype-of } T, T'.B_{\text{inherited}}.B_{\text{insert}}(b, t)
\]

Behavior \( B_{\text{last}} \) returns the collection of behaviors that are currently valid from the interface history of \( S \). Let us assume \( T.\text{taxSource} \) has the behavior \( B.\text{taxBracket} \) defined at \( t_0 \). \( B.\text{taxBracket} \) then has to be added to the history of inherited behaviors of \( T.\text{employee} \). This is done by the behavior application \( T.\text{employee}.B_{\text{inherited}}.B_{\text{insert}}(B.\text{taxBracket}, t_0) \). The history of the inherited behaviors would then be \( \{<t_0, \{B.\text{name}, B.\text{birthDate}, B.\text{age}, B.\text{taxBracket}\}>\} \). Behaviors \( B.\text{name}, B.\text{birthDate}, B.\text{age} \) are inherited from type \( T.\text{person} \) (see Figure 3), while behavior \( B.\text{taxBracket} \) is inherited from type \( T.\text{taxSource} \).

**Modify Type - Drop Supertype Link (MT-DSL).** Drop a direct supertype link between two types (a direct supertype link to \( T.\text{object} \) cannot be dropped) at time \( t \). Consider types \( T \) and \( S \) where \( S \) is the direct supertype of \( T \). Then, removing the direct supertype link between \( T \) and \( S \) at time \( t \) has the following effects:

• Adjust the history of supertypes of \( T \) and the history of subtypes of \( S \). For example, dropping the supertype link between \( T.\text{employee} \) and \( T.\text{taxSource} \) at \( t_5 \) requires updating the history of supertypes of \( T.\text{employee} \) and history of subtypes of \( T.\text{taxSource} \). This is carried out using the behavior applications \( T.\text{employee}.B_{\text{supertypes}}.B_{\text{remove}}(T.\text{taxSource}, t_5) \) and \( T.\text{taxSource}.B_{\text{subtypes}}.B_{\text{remove}}(T.\text{employee}, t_5) \).

• The MT-ASL operation is carried out from \( T \) to every supertype of \( S \), unless \( T \) is linked to the supertype through another chain. This operation is not required when the
supertype link between T.employee and T.taxSource is dropped because T.employee is linked to the supertype of T.taxSource (T.object) through T.person.

- The MT-ASL operation is carried out from each subtype of T to S, unless the subtype is linked to S through another chain. This operation requires adding a supertype link between T.null and T.taxSource.

- The native behaviors of S are dropped from the interface of T. That is, the history of inherited behaviors of T is adjusted. This means the behavior B.taxBracket, defined natively on T.taxSource, has to be dropped from the history of inherited behaviors of T.employee. The behavior application T.employee.B.inherited.B.remove(B.taxBracket,t5).

5 Queries

In this section we show how queries can be constructed using the TIGUKAT query language (TQL) [PLÖS93] to retrieve schema objects at any time in their evolution histories. This gives software designers a temporal user interface which provides a practical way of accessing temporal information in their experimental and incremental design phases. TQL incorporates reflective temporal access in that it can be used to retrieve both objects, and schema objects in a uniform manner. Hence, TQL does not differentiate between queries (which query objects) and meta-queries (which query schema objects).

5.1 The TIGUKAT Query Language

In this section, we briefly discuss the TIGUKAT Query Language (TQL). TQL\textsuperscript{7} is based on the SQL paradigm [Dat87] and its semantics is defined in terms of the object calculus. Hence, every statement of the language corresponds to an equivalent object calculus expression. The basic query statement of TQL is the select statement which operates on a set of input collections, and returns a new collection as the result:

\[
\text{select } < \text{object variable list}> \\
\text{[ into } < \text{collection name} > \text{]} \\
\text{from } < \text{range variable list} > \\
\text{[ where } < \text{boolean formula} > \text{]} 
\]

The select clause in this statement identifies the objects to be returned in a new collection. There can be one or more object variables with different formats (constant, variables, path expressions

---

\textsuperscript{7}TQL was developed before the release of OQL [Cat94]. It is quite similar to OQL in structure.
or index variables) in this clause. They correspond to free variables in object calculus formulas. The *into clause* declares a reference to a new collection. If the *into clause* is not specified, a new collection is created; however, there is no reference to it. The *from clause* declares the ranges of object variables in the *select* and *where* clauses. Every object variable can range over either an existing collection, or a collection returned as a result of a subquery, where a subquery can be either given explicitly, or as a reference to a query object. The *where clause* defines a boolean formula that must be satisfied by objects returned by a query.

Having described TQL, we show in the next section how temporal objects can uniformly be queried using behavior applications without changing any of the basic constructs of TQL.

5.2 Query Examples

*Example 5.1* Return the time when the behavior *B*.*children* was added to the type *T*.*person*.

```
select b.B.timestamp
from b in T.person.B.interface.B.history
where B.children in b.B.value
```

The result of this query would be the time $t_{10}$ as seen in Figure 3. □

*Example 5.2* Return the types that define behaviors *B*.*age* and *B*.*taxBracket* as part of their interface.

```
select T
from T in C.type
where (b1 in T.B.interface.B.history and B.age in b1.B.value) or
      (b2 in T.B.interface.B.history and B.taxBracket in b2.B.value)
```

This query would return the types *T*.*person*, *T*.*taxSource*, *T*.*employee*, and *T*.*null*. The type *T*.*person* defines behavior *B*.*age* natively (see Figure 3), while the type *T*.*taxSource* defines behavior *B*.*taxBracket* natively. The behaviors *B*.*age* and *B*.*taxBracket* are inherited by types *T*.*employee* and *T*.*null* since they are subtypes of *T*.*person* and *T*.*taxSource* as shown in Figure 1. □

*Example 5.3* Return the implementation of behavior *B*.*age* in type *T*.*person* at time $t_1$.

```
select i.B.value
from i in B.age.B.implementation(T.person).B.history
where i.B.timestamp.B.leastShaneqto(t_1)
```
The behavior $B_{\text{lessthaneqto}}$ is defined on type $T_{\text{timeStamp}}$ and checks if the receiver timestamp is less than or equal to the argument timestamp. The result of the query is the computed function $c_1$ as shown in Figure 4. □

**Example 5.4** Return the super-lattice of type $T_{\text{employee}}$ at time $t_3$.

```sql
select r.B_value
from r in T_{\text{employee}}.B_super-lattice.B_history
where r.B_timestamp.B_{\text{lessthaneqto}}(t_3)
```

The super-lattice of $T_{\text{employee}}$ at $t_3$ consists of the types $T_{\text{person}}$, $T_{\text{taxSource}}$, and $T_{\text{object}}$. This is shown in Figure 5. □

**Example 5.5** Return the types that define behavior $B_{\text{age}}$ with the same implementation as one of their supertypes.

```sql
select T
from T in C_{\text{type}}, S in T.B_supertypes.B_history,
    i in B_{\text{age}}.B_implementation(T).B_history,
    j in B_{\text{age}}.B_implementation(S.B_value).B_history
where b in S.B_value.B_interface.B_history and B_{\text{age}} in b.B_value and
    i.B_value = j.B_value and i.B_timestamp = j.B_timestamp
```

This query would return the types $T_{\text{employee}}$, $T_{\text{patient}}$, and $T_{\text{null}}$, assuming the implementation of behavior $B_{\text{age}}$ is not changed when it is inherited by these types. □

### 6 Conclusion

In this paper a uniform treatment of schema evolution and temporal support for object database management systems (ODBMS) is presented. Schema evolution is managed by exploiting the functionality of a temporal object model. The evolution history of the interface of types, which includes the inherited and native behaviors of each type, describes the semantics of types through time. Using the interface histories the interface of a type can be reconstructed at any time of interest. The evolution histories of the supertype and subtype links of types describe the structure of the lattice through time. Using these histories, the structure of the lattice can be reconstructed at any time of interest. The implementation histories of behaviors give us the implementations of behaviors on types at any time of interest. From these, we can reconstruct the representation...
of objects by examining the stored functions associated with behaviors at a given time. The TIGUKAT query language gives designers a practical way of accessing temporal information in their experimental and incremental design phases.

Our next step is to give a comprehensive treatment to the change propagation problem during schema evolution. That is, devising methods to propagate schema changes to the existing object instances in the TIGUKAT temporal ODBMS. In order for the instances to remain meaningful after the schema has changed, either the relevant instances must be coerced into the new definition of the schema or a new version of the schema must be created leaving the old version intact. Conversion of objects can be optional in our model. Since the evolution history of schema objects is maintained, all the information for older objects is available and we can use this information to continue processing these objects in the old way. Since our model is time based, the old information of the object is available. Thus, even if objects are coerced to a newer schema definition, historical queries can be run by giving an appropriate time point in the history of the object.

To overcome the corrective nature of schema evolution, the concept of schema versioning in ODBMSs has been proposed [SZ86, SZ87, KC88, ALP91, MS92, MS93]. In most of these systems, a change to a schema object may result in a new version of the schema object, or the schema in general. However, schema changes are usually of a finer granularity than definable versions. This implies that not every schema change should necessarily result in a new version. Rather, one should be able to define a version during any stable period in the evolutionary history of the schema. Within a particular version, the evolution of the schema should be traceable. For example, in an engineering design application many components of an overall design may go through several modifications in order to produce a final product. Furthermore, each intermediate version of the component may have certain properties that need to be retained as a historical record of that particular component (the different versions may have been used in other products). The interconnection of the various versions of components also gives rise to versions of the overall design. The resulting designs may be part of others and so on. Our contention is that schema evolution using temporal modeling sets the stage for full-fledged version control. We intend to use the schema evolution policies reported in this paper as a basis for version control in ODBMSs.

References


