Managing Schema Evolution Using a Temporal Object Model

Iqbal A. Goralwalla[1], Duane Szafron[1], M. Tamer Özsu[1], and Randal J. Peters[2]

[1] Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2H1
[2] Department of Computer Science, University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

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 [16] 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 [3].
- The engineering and design oriented application domains (e.g., CAD, software design process) require incremental design and experimentation [7, 5]. 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 temporality 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 experimentation 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 information. 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) definition of change semantics, and (2) change propagation. Change semantics is usually defined by specifying of invariants over schema changes. Change propagation deals with reflecting changes to the individual objects by *coercing* them to coincide with the new schema definition. 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 *temporal* ODBMS [10,6] 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 [1,11,9]. 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 model [1] 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 [18] 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 [1] 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, [15, 19, 20, 2]). 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 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.

Skarra and Zdonik [17] 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 [12, 10] 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; 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.

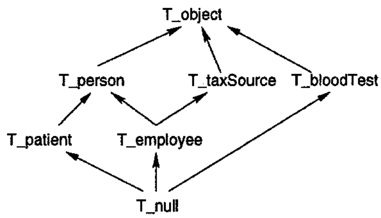

Fig. 1. Simple type lattice.

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. In Figures 2-4, 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. A circle labeled with the symbols { } represents a collection object and has outgoing edges labeled with "∈" to each member of the collection.

The access and manipulation of an object's state occurs exclusively through the application 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 [8, 6].

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 *timestamped objects*. A timestamped object knows its timestamp (time interval or time instant) and its associated value at (during) the timestamp. The following behaviors are defined on the $\mathtt{T\_history}\langle\mathtt{T\_X}\rangle$ type:

$$
\begin{aligned}
B\_history &: \mathtt{T\_collection}\langle\mathtt{T\_timeStampedObject}\langle\mathtt{T\_X}\rangle\rangle \\
B\_timeline &: \mathtt{T\_timeline} \\
B\_insert &: \mathtt{T\_X}, \mathtt{T\_timeStamp} \rightarrow \\
B\_remove &: \mathtt{T\_X}, \mathtt{T\_timeStamp} \rightarrow \\
B\_validObjects &: \mathtt{T\_timeStamp} \rightarrow \mathtt{T\_collection}\langle\mathtt{T\_timeStampedObject}\langle\mathtt{T\_X}\rangle\rangle \\
B\_validObject &: \mathtt{T\_timeStamp} \rightarrow \mathtt{T\_timeStampedObject}\langle\mathtt{T\_X}\rangle
\end{aligned}
$$

Behavior *B_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_timeline*. The timeline basically orders the timestamps of timestamped objects [6]. The *B_insert* behavior accepts an object and a timestamp as input and creates a timestamped object that is inserted into the history. Behavior *B_remove* drops a given object from the history at a specified timestamp. The *B_validObjects* behavior allows the user to get the objects in the history that were valid at (during) a given timestamp. Behavior *B_validObject* is derived from *B_validObjects* to return the timestamped object that exists at a given time instant.

Each timestamped object is an instance of the $\mathtt{T\_timeStampedObject}\langle\mathtt{T\_X}\rangle$ type. This type represents objects and their corresponding timestamps. Behaviors *B_value* and *B_timeStamp* defined on `T_timeStampedObject` return the value and the timestamp (time interval or time instant) of a timestamped object, respectively.

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 [12, 14]. The **bold** entries represent combinations that implicate schema changes while the *emphasized* entries denote non-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 [12, 14]. 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.

	Operation		
Objects	Add (A)	Drop (D)	Modify (M)
Type (T)	**subtyping**	**type deletion**	**add behavior**(AB) **drop behavior**(DB) **add supertype link**(ASL) **drop supertype link**(DSL)
Class (C)	**class creation**	**class deletion**	*extent change*
Behavior (B)	*behavior definition*	**behavior deletion**	**change association**(CA)
Function (F)	*function definition*	**function deletion**	*implementation change*
Collection (L)	**collection creation**	**collection deletion**	*extent change*

Table 1. Classification of schema changes.

The meta-model of TIGUKAT is uniformly represented within the object model itself, providing reflective capabilities [13]. 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: the collection of *native* behaviors which are those behaviors defined by the type and are not defined on any of its supertypes; and 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 *B_native* behavior is

B_native : `T_history⟨T_collection⟨T_behavior⟩⟩`. The definitions for behaviors *B_inherited* and *B_interface* are extended similarly. Each behavior now returns a history 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. 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 2 shows the history of the entire interface for the type `T_person`.

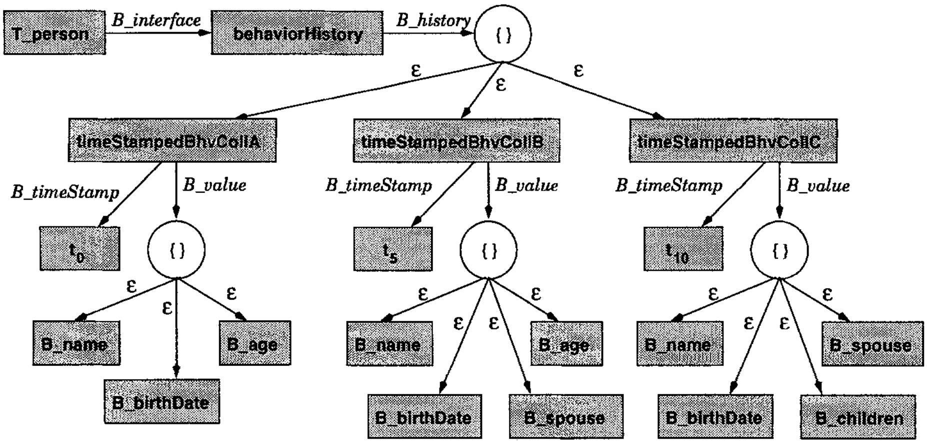

Fig. 2. Interface history of type `T_person`.

At time t_0, behaviors *B_name*, *B_birthDate*, *B_age* are defined on `T_person`. 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 $\{<t_0, \{B\_name, B\_birthDate, B\_age\}>, <t_5, \{B\_name, B\_birthDate, B\_age, B\_spouse\}>, <t_{10}, \{B\_name, B\_birthDate, B\_spouse, B\_children\}>\}$[1]. 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_person` was $\{B\_name, B\_birthDate, B\_age\}$,

[1] Note that in Figure 2 objects that are repeated in the timestamped collections are actually the same object. For example, the *B_name* object in all three timestamped collections is the same object. It is shown three times in the figure for clarity.

at time t_5 it was $\{B\_name, B\_birthDate, B\_age, B\_spouse\}$, and at time t_{10} (now) it is $\{B\_name, B\_birthDate, B\_spouse, B\_children\}$.

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 $T.B\_native.B\_insert(b,t)$ and $T.B\_interface.B\_insert(b,t)$ perform this update. For example, the behavior application `T_person`.*B_interface.B_insert*(B_spouse,t_5) updates the interface history of `T_person` when behavior *B_spouse* is added to `T_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 $b.B\_implementation.B\_insert(f,t)$ (details on implementation histories of behaviors are given in Section 4.3).
- The history of inherited and interface behaviors of all subtypes of type T needs to be adjusted. That is, $\forall T' \mid T'$ subtype-of $T, T'.B\_inherited.B\_insert(b,t)$ and $T'.B\_interface.B\_insert(b,t)$. For example, the histories of inherited and interface behaviors of the `T_employee` and `T_patient` types (see Figure 1) need to be adjusted to reflect the addition of behavior *B_spouse* in type `T_person` at time t_5. For the `T_employee` type, this is accomplished using the 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. 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.
- The native and inherited behavior histories of the subtypes of T (possibly) change.

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_implementation* behavior with time-varying properties – *B_implementation* : **T_type** → **T_history⟨T_function⟩**. With this behavior we can determine the implementation of a behavior defined on a type at any time of interest. For example, Figure 3 shows the history of the implementations for behavior *B_age* on type **T_person**. There are two kinds of implementations for behaviors [12]. 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.

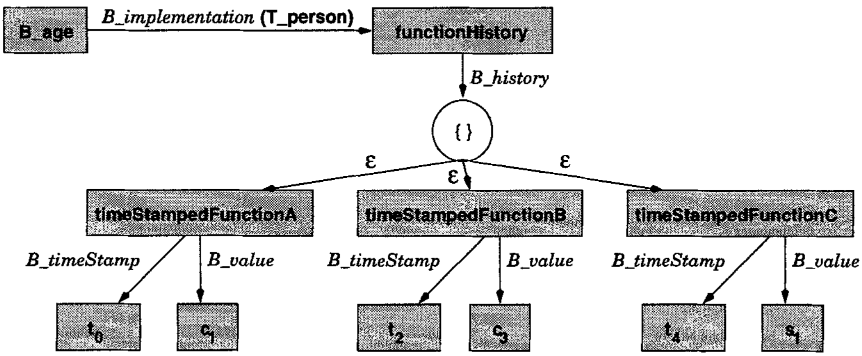

Fig. 3. Implementation history of behavior *B_age* on type **T_person**.

In Figure 3, we use c_i to denote a computed function, s_i to denote a stored function. At time t_2, the implementation of *B_age* changed from the computed function c_1 to the computed function c_3. At time t_4, the implementation of *B_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_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[2] for any type at any time t. For example, the interface of type **T_person** at time t_3 is given by the behavior application **T_person**.$[t_3]$*B_interface* which results in $\{B\_name, B\_age, B\_birthDate\}$, as shown in Figure 2. We use the syntax $o.[t]b$ to denote the application of behavior b to object o at time t. The implementation of *B_age* at time t_3 is given by *B_age*.$[t_3]$*B_implementation*(**T_person**) which is c_3, as shown in Figure 3.

[2] Stored functions associated with behaviors allow us to reconstruct object representations (i.e., states of objects) 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.

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_supertypes* and *B_subtypes* behaviors defined on `T_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_supertypes* and *B_subtypes* behaviors to return a history of a collection of supertypes and subtypes, respectively. Using the *B_supertypes* and *B_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_superlattice* and *B_sublattice* are defined on `T_type`. The behavior *B_superlattice* is derived by recursively applying *B_supertypes* until `T_object` is reached, while the behavior *B_sublattice* is derived by recursively applying *B_subtypes* until `T_null` is reached. In both cases, the intermediate results are partially ordered. Figure 4 shows the supertype lattice history for type `T_employee`.

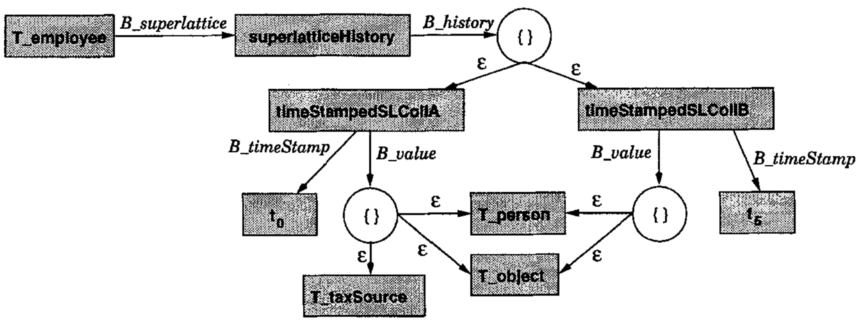

Fig. 4. Supertype lattice history for type `T_employee`.

At time t_0, the superlattice history of type `T_employee` included the types `T_person`, `T_taxSource`, and `T_object`. At time t_5, the supertype link between `T_employee` and `T_taxSource` is dropped. To reflect this change, the superlattice history of `T_employee` is updated to $\{<t_0, \{\texttt{T\_person}, \texttt{T\_taxSource}, \texttt{T\_object}\}>, <t_5, \{\texttt{T\_person}, \texttt{T\_object}\}>\}$.

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 $T.B\_supertypes.B\_insert(S,t)$ performs this update. For example, adding the supertype link between `T_employee` and `T_taxSource` at t_0 necessitates an update to the history of supertypes for `T_employee`.
- The history of the collection of subtypes of type S is updated. In this case, the history of the collection of subtypes of `T_taxSource` has to be updated.
- 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. Formally, $\forall b \in S.B\_interface.B\_history.B\_last$, $\forall T' \mid T'$ subtype-of $T, T'.B\_inherited.B\_insert(b,t)$. Behavior *B_last* returns the collection of behaviors that are currently valid from the interface history of S. Let us assume `T_taxSource` has the behavior *B_taxBracket* defined at t_0. *B_taxBracket* then has to be added to the history of inherited behaviors of `T_employee`. The history of the inherited behaviors would then be $\{<t_0, \{B\_name, B\_birthDate, B\_age, B\_taxBracket\}>\}$. Behaviors *B_name*,*B_birthDate*,*B_age* are inherited from type `T_person` (see Figure 2), while behavior *B_taxBracket* is inherited from type `T_taxSource`.

Modify Type - Drop Supertype Link (MT-DSL). Drop a direct supertype link between two types (a direct supertype link to `T_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_employee` and `T_taxSource` at t_5 requires updating the history of `T_employee`'s supertypes and the history of subtypes of `T_taxSource`.
- The **MT-ASL** operation is carried out from T to every supertype of S, unless T is linked to the supertype through another chain.
- The **MT-ASL** operation is carried out from each subtype of T to S, unless the subtype is linked to S through another chain.
- The native behaviors of S are dropped from the interface of T. That is, the history of inherited behaviors of T is adjusted.

5 Queries

In this section we show how queries can be constructed using the TIGUKAT query language (TQL) [10] 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). TQL is based on the SQL paradigm [4] and its semantics is defined in terms of the object calculus. Hence, every statement of the language corresponds to an equivalent object calculus expression. We now give several example queries which illustrate how temporal objects can uniformly be queried using behavior applications without changing any of the basic constructs of TQL.

Example 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 2.

Example 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 (*b*1 **in** *T.B_interface.B_history* **and** *B_age* **in** *b*1*.B_value*) **or**
(*b*2 **in** *T.B_interface.B_history* **and** *B_taxBracket* **in** *b*2*.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 2), 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 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_lessthaneqto*(t_1)
The behavior *B_lessthaneqto* is defined on type `T_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 3.

Example 4. Return the super-lattice of type `T_employee` at time t_3.
select *r.B_value*
from *r* **in** `T_employee`*.B_super-lattice.B_history*
where *r.B_timestamp.B_lessthaneqto*(t_3)
The super-lattice of `T_employee` at t_3 consists of the types `T_person`, `T_taxSource`, and `T_object`. This is shown in Figure 4.

Example 5. Return the types that define behavior *B_age* with the same implementation as one of their supertypes.
select T
from T **in** **C_type**, S **in** *T.B_supertypes.B_history*,
  i **in** *B_age.B_implementation(T).B_history*,
  j **in** *B_age.B_implementation(S.B_value).B_history*
where b **in** *S.B_value.B_interface.B_history* **and** *B_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_employee`, `T_patient`, and `T_null`, assuming the implementation of behavior *B_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.

References

1. J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 311–322, May 1987.

2. E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In *Proc. 5th Int'l Conf. on Extending Database Technology*, March 1996.
3. C. Combi, F. Pinciroli, and G. Pozzi. Managing Different Time Granularities of Clinical Information by an Interval-Based Temporal Data Model. *Methods of Information in Medicine*, 34(5):458–474, 1995.
4. C.J. Date. *A Guide to SQL Standard.* Addison Wesley, 1987.
5. S. Gibbs, D.C. Tsichritzis, E. Casais, O.M. Nierstrasz, and X. Pintado. Class Management for Software Communities. *Communications of the ACM*, 33(9):90–103, September 1990.
6. I.A. Goralwalla, M.T. Özsu, and D. Szafron. Modeling Medical Trials in Pharmacoeconomics using a Temporal Object Model. *Computers in Biology and Medicine - Special Issue on Time-Oriented Systems in Medicine*, 1997. In Press.
7. W. Kim, J. Banerjee, H.T. Chou, and J.F. Garza. Object-oriented database support for CAD. *Computer Aided Design*, 22(8):469–479, 1990.
8. J.Z. Li, I.A. Goralwalla, M.T. Özsu, and Duane Szafron. Modeling Video Temporal Relationships in an Object Database Management System. In *Proceedings of Multimedia Computing and Networking (MMCN97)*, February 1997.
9. G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, 4:43–67, 1989.
10. M.T. Özsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal*, 4:100–147, August 1995.
11. D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 111–117, October 1987.
12. R.J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Management System.* PhD thesis, University of Alberta, 1994.
13. R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. 12th Int'l Conf. on the Entity Relationship Approach*, pages 37–49, December 1993.
14. R.J. Peters and M.T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions on Database Systems*, 22(1):75–114, March 1997.
15. E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. 10th Int'l Conf. on the Entity Relationship Approach*, pages 205–229, October 1991.
16. Dag Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, January 1993.
17. A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 483–495, September 1986.
18. J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.
19. S.Y.W. Su and H.M. Chen. A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T. In *Proc. 17th Int'l Conf. on Very Large Data bases*, 1991.
20. G. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proc. 8th Int'l. Conf. on Data Engineering*, pages 584–593, February 1992.