A Temporal Approach to Managing Schema Evolution in Object Database Systems

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

Randal J. Peters
Advanced Database Systems Laboratory
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. However, to properly model applications that need incremental design and experimentation, the evolutionary histories of the schema objects should be traceable rather than corrective so that historical queries can be supported. In this paper we propose a method for managing schema changes, and propagating these changes to object instances 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.

Keywords: schema evolution, change propagation, temporal model, object-oriented, database

1 Introduction

In this paper, we address the issues 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 system (ODBMS) applications require both. For example:

- The results reported in [Sjø93] 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 so that historical queries can be executed.

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.

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, a method for managing schema changes and propagating the changes to underlying instances by exploiting the functionality of a temporal object model is presented. The approach described in this work is conducted within the context of the TIGUKAT temporal DBMS. However, the results reported here extend to any ODBMS that uses time to model evolution histories of objects.

Schema evolution 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, and adding and dropping subtype relationships between types. 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 schema objects (e.g., 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. By defining appropriate behaviors on the meta-architecture, the evolution of schema is supported. Any changes in schema object 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.

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 change to the schema of an object database system necessitates corresponding changes to the underlying object instances in order to ensure the overall consistency of the system. Change propagation deals with reflecting changes to the individual objects by *coercing* them to coincide with the new schema definition. Two main approaches have been proposed to deal with coercing object instances to reflect the changed schema: *immediate* and *deferred*. Immediate object coercion results in suspension of all running programs until all objects have been coerced, while deferred object coercion leads to delays each time an object is accessed.

The change propagation strategy proposed in this paper supports both deferred object update semantics and immediate object update semantics. The granularity of object coercion is based on individual behaviors. That is, individual behaviors defined on the type of an object can be coerced to a new definition for that object when the object is accessed, leaving the other behaviors to retain their old definitions. This is in contrast to other models where an object is converted in its entirety to a changed type. The approach taken in our work has two distinct advantages, depending on whether deferred or immediate update semantics are used. If deferred update semantics are used, the “behavior-at-a-time” coercion results in an even “lazier” update semantics, since a behavior application to an object results in the update of only part of the object's structure. Updates due to other behavior changes are delayed until they are needed by other behavior applications. If immediate update semantics are used, then the update can be done more quickly since the system knows that changes to the affected type are localized to the single behavior that was just changed. This is important because the major drawback of immediate update semantics is the speed of update. Another identifying characteristic of the propagation model is that a historical record of the coerced behaviors is maintained for each object so that even if behaviors are coerced to reflect an update to an object, older definitions of the behaviors can still be accessed for each object.

The remainder of the paper is organized as follows. In Section 2, we examine some of the previous work on schema evolution. We also examine the three main approaches to schema change propagation, and compare our approach to these. 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 describe how behavior implementation changes are propagated to underlying object instances, and provide algorithms that implement the semantics of our time-varying behavior dispatch process. In Section 6, we show how the immediate object coercion is implemented in our approach. Concluding remarks and results of the paper are summarized in Section 7.

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 much 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. Orion [BKKK87, KC88] 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] are 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, make revisions if necessary, and execute historical queries.

There have been many temporal object model proposals (for example, [RS91, SC91, WD92, KS92, CITB92, BFG97]). 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.

In the context of relational temporal models, Ariav [Ari91] examines the implications of allowing data structures to evolve over time, 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 meta-database 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.

In addition to schema modifications, a system must define how schema changes are reflected in the instances. In order for the instances to remain meaningful, 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. Three main approaches have been identified and employed in the past. *Immediate* (*conversion*) and *deferred* (*lazy, screening*) propagate changes to the instances - only at different times - while *filtering* is a solution for versioning that attempts to maintain the semantic differences between versions of schema. A fourth approach is to combine the above three methods into a *hybrid* model. The various techniques are summarized below.

- **Immediate:** Each schema change initiates an immediate conversion of all objects affected by the change. This approach causes delays during the modification of schema, but no delays are incurred during access to objects. GemStone [PS87] and O_2 [FMZ$^+$95] systems report the use of immediate conversion for schema change propagation. In O_2, immediate conversion is implemented using the algorithm defined for deferred conversion.

- **Deferred:** Schema changes generate a conversion program that is capable of converting objects into the new representation. The conversion is not immediate; but is delayed until an instance of the modified schema is accessed. Object access is monitored and whenever an object is accessed, the conversion program is invoked, if necessary, to convert the object into the new definition. The conversion programs resulting from multiple independent changes to a type are composed, meaning access to an object may invoke the execution of multiple conversion programs where each one handles a certain change to the schema. Deferred conversion causes delays during object access. ORION [BKKK87] uses this approach and OTGen [LH90] uses it for database reorganization. In O_2 [FMZ94, FMZ$^+$95], implementation strategies are defined for conversion functions implemented as deferred database updates.

- **Filtering:** In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. When the schema is changed, the old objects remain with the old version of the schema and new objects are created as instances of the new one. The filters define the consistency between the old and new schema versions and handle the problems associated with behaviors written according to one version accessing objects of a different version. Error handlers are one example of filters. They can be defined on each version of the schema to trap inconsistent access and produce error and warning messages. The Encore model [SZ86] uses type versioning with error handlers as a filtering mechanism. The Avance [BH89] system adopts a similar approach to Encore. Exception handlers are defined as filters to cope with mismatches between different versions. Both Encore and Avance use emulation to present old instances as if they are new ones. It is not possible to associate additional storage with existing attributes since all objects are strictly connected to the version in which they were created. As such additional attributes would necessarily be read-only and have a fixed, default value. This problem is remedied in CLOSQL [MS92] where objects are allowed to dynamically change the class version with which they are connected. Each attribute of an object has update and backdate functions (provided by the user) for converting objects into different formats. However, the overhead of the conversion process and the added responsibility on the user are quite significant in CLOSQL.

- **Hybrid:** A hybrid approach combines two or more of the above methods. GemStone mentions an effort to incorporate a hybrid approach, but currently we are unaware of such a system implementation. In Sherpa [NR89], schema changes are propagated to instances through conversion or screening, which is selected by the user. However, only the conversion approach is discussed. Change propagation is assisted by the notion of *relevant classes*. A *relevant class* is a semantically consistent partial definition of a complete class and is bound to the class. A relevant class is similar to a type version in [SZ86] and a complete class resembles a version set.

Although numerous approaches have been proposed for propagating different schema changes to object instances, the schema change that involves changing the implementation of a behavior, and how it affects the underlying object structure has not been addressed comprehensively. In this work, a deferred approach that uses a finer grained filtering based on behavior histories is used as the underlying mechanism for behavior implementation change propagation. The approach also allows for immediate behavior coercion to reflect the changed schema. This makes it feasible for the system to take a more active role by using deferred object coercion as the default and switching to immediate object coercion whenever the system is idle.

In systems that use immediate or deferred object coercion, the entire object must be converted upon coercion and in the systems that don't define versions of schema, the old state of the object is lost. The approach in this paper differs in that the granularity of object coercion is based on individual behaviors. That is, an individual behavior of an object's type can be coerced to a new definition for that object, leaving the other behaviors to retain their old definitions. Furthermore, a historical record of the coerced behaviors is maintained for each object so one can access the older definitions of the behaviors for each object. Complete object conversion takes place only if all behaviors defined in the type of the object have been coerced. This results in considerable savings of work.

3 The Temporal Object Model

3.1 Basic Object Model

We work with an object model whose identifying characteristics are its *behavioral* nature and its *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 features supported by the model include strong object identity, abstract types, strong typing, complex objects, full encapsulation, multiple inheritance, and parametric types. This is the model of the TIGUKAT ODBMS [Pet94, ÖPS+95] that is being implemented at the University of Alberta.

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[1]; 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.

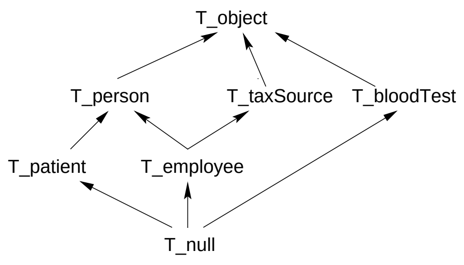


Figure 1: Simple type lattice.

The access and manipulation of an object's state occurs exclusively through the application of behav-

[1]Types and their extents are separate constructs in our model.

iors. 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 different implementations in each of the types. This provides direct support for behavior *overloading* and *late binding* of functions (implementations) to behaviors.

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. Types are instances of the primitive type `T_type` and behaviors are instances of the type `T_behavior`. The type `T_object` binds the type lattice from the top (i.e., least defined type) while the type `T_null` binds the lattice from the bottom (i.e., most defined type). The behavior *B_baseType* is defined on the parametric type T_X⟨T_Y⟩ to return the base (argument) type T_Y.

3.2 The Temporal Extensions

The philosophy behind adding temporality to the basic object model is to accommodate multiple applications that have different type semantics requiring various notions of time [LGÖS97, GÖS97]. Consequently, the 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 [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:

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

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 [GLÖS96]. 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 `T_timeStampedObject`$\langle$`T_X`$\rangle$ type. This type represents objects and their corresponding timestamps. Type `T_timeStampedObject` defines behaviors *B_value* and *B_timeStamp* which return the value and the timestamp (time interval or time instant) of a timestamped object, respectively.

Example 3.1 Suppose the type `T_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 bloodTests* is defined on `T_patient` to return an object of type `T_history`$\langle$`T_bloodTest`$\rangle$. Each blood test is represented by an object of the type `T_bloodTest`. Therefore, the history of the different blood tests undertaken by joe (an instance of `T_patient`) would then be retrieved using the behavior application joe.*B_bloodTests*. Let us call this history object bloodTestHistory. Now, suppose joe was suspected of having septicemia[2] and had diagnostic hematology and microbiology blood tests on 15 January 1995. As a result of a raised white cell count, 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_bloodTest` were created and then entered into the object database using the following TIGUKAT behavior applications:

$$\mathsf{bloodTestHistory}.B\_insert(\mathsf{microbiology}, 15\ January\ 1995)$$
$$\mathsf{bloodTestHistory}.B\_insert(\mathsf{hematology1}, 15\ January\ 1995)$$
$$\mathsf{bloodTestHistory}.B\_insert(\mathsf{hematology2}, 20\ February\ 1995)$$

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

$$\mathsf{bloodTestHistory}.B\_validObjects([1\ January\ 1995,\ 31\ January\ 1995])$$

This would return a collection of the two timestamped objects, $\{$timeStampedMicrobiology, timeStampedHematology1$\}$, representing the blood tests joe took in January 1995. The first timestamped object would have microbiology as its value and the second would have hematology1 as its value[3].

[2]An infection of the blood.

[3]It 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* = $15\ January\ 1995$.

To assist in clarifying the contents and structure of a history object, we give a pictorial representation of bloodTestHistory in Figure 2. 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. □

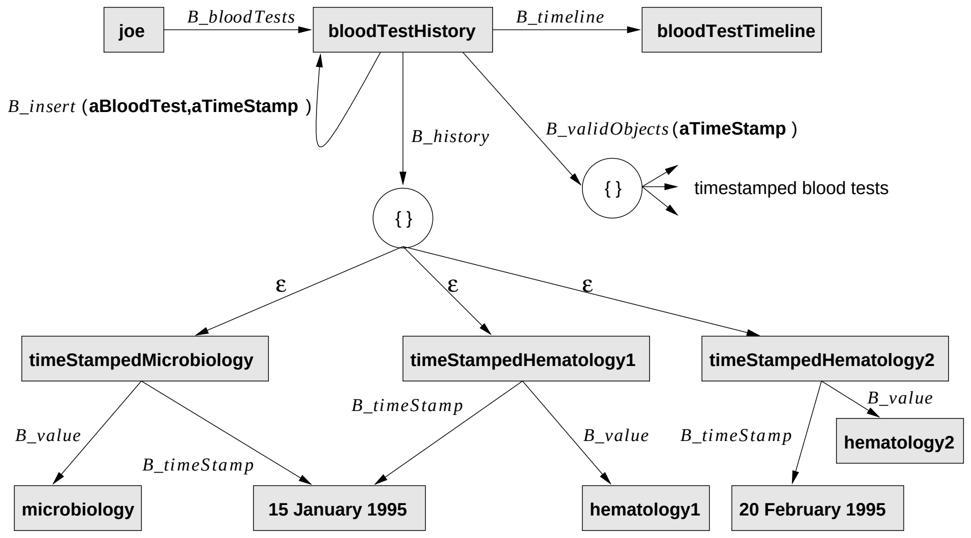


Figure 2: A pictorial representation of a patient's blood test history.

Another important behavior introduced by the temporal extensions is the *B_lifespan* behavior defined on `T_object`. This behavior is applied to an object, accepts a collection as an argument, and returns a timestamp (interval) representing the time in which the object exists in the given collection. For example, the behavior application joe.*B_lifespan*(joe.*B_mapsto*.*B_classof*.*B_shallowExtent*) returns the lifespan of the object joe in the class associated with `T_person`. The behavior *B_mapsto* is defined in `T_object` and returns the type of the receiver object. The *B_classof* behavior is defined in `T_type` and returns the class associated with the receiver type object, and the behavior *B_shallowExtent* returns all the elements of `T_person` excluding objects from the subtypes of `T_person`.

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. Schema 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 imply schema changes while the *emphasized* entries denote non-schema changes.

	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.

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 semantics of the changes to a type are discussed in the following sections. The discussion includes the neccessary behavior applications that would be needed to accomodate changes to a type. These behavior applications would all be done by the system to manage the temporal schema information. They are shown and described here to illustrate that changes to a type can be done through consistent behavioral semantics of the TIGUKAT model itself.

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 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:

$$\begin{aligned} B\_native &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \\ B\_inherited &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \\ B\_interface &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_behavior}\rangle\rangle \end{aligned}$$

Each behavior now returns a history consisting of a collection whose elements are timestamped collections of 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`.

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

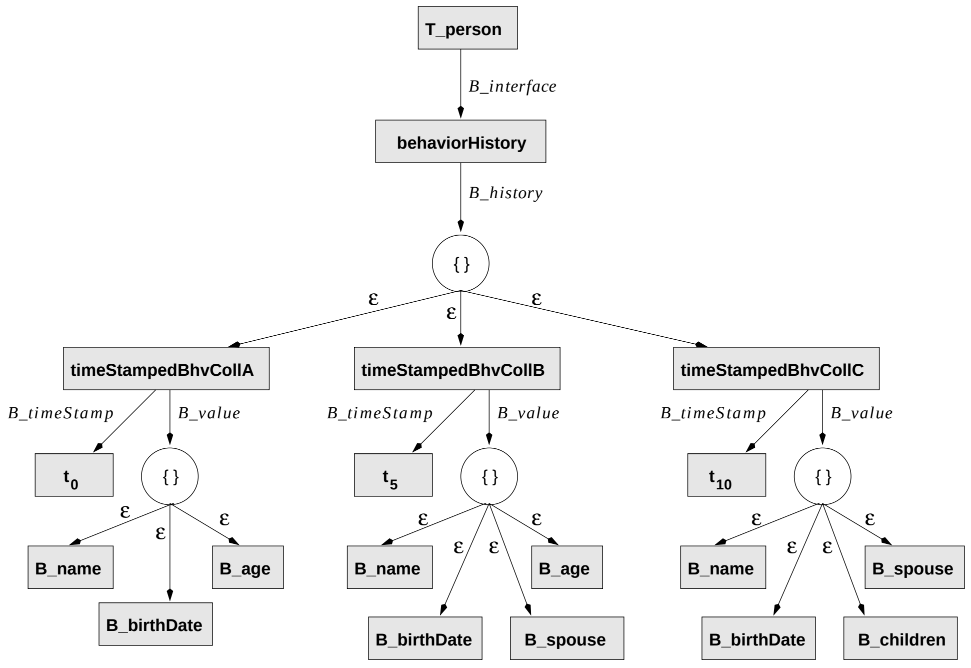


Figure 3: Interface history of type `T_person`.

time behavior *B_children* is added. The final history of the interface of `T_person` after this change is $\{<t_0, \{B\_name, B\_birthDate, B\_age\}>, <t_5, \{B\_name, B\_birthDate, B\_age, B\_spouse\}>, <t_{10}, \{B\_name, B\_birthDate, B\_spouse, B\_children\}>\}$[4]. 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 $\{B\_name, B\_birthDate, B\_age\}$, 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

[4]Note that in Figure 3 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.

on implementation histories of behaviors are given in Section 4.3). For example, if the function associated with behavior *B_spouse* is the stored function s_{spouse}, then the implementation history of *B_spouse* is updated using the behavior application *B_spouse*.*B_implementation*.*B_insert* (s_{spouse},t_5).

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

  $\forall T^{'} \mid T^{'}$ 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 types `T_employee` and `T_patient` (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. 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_implementation* behavior with time-varying properties. The definition of the extended behavior is as follows:

$$B\_implementation : \texttt{T\_type} \rightarrow \texttt{T\_history}\langle\texttt{T\_function}\rangle$$

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_age* on type `T_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.

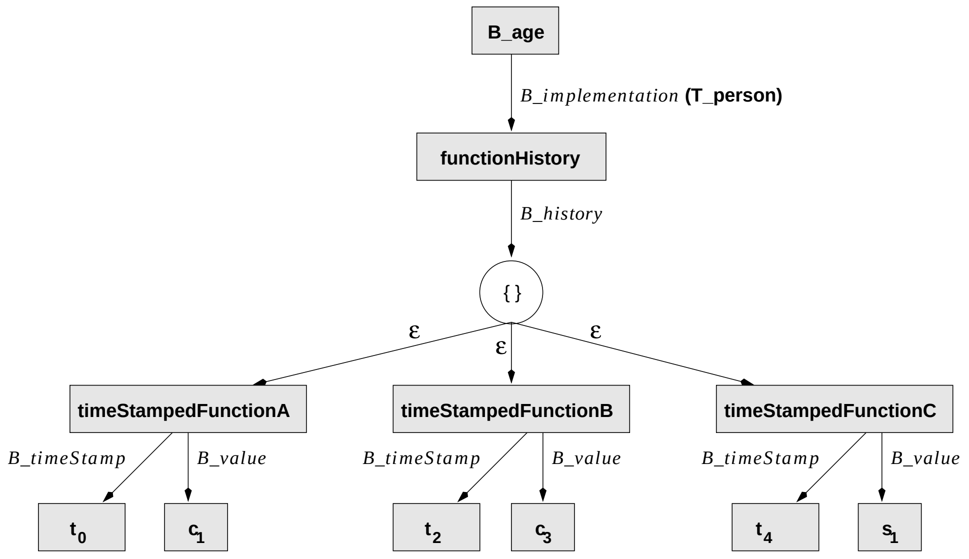


Figure 4: Implementation history of behavior *B_age* on type `T_person`.

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_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[5] 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 $\{B\_name, B\_birthDate, B\_age\}$ 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_age* at t_3 is given by *B_age*.$[t_3]$*B_implementation* (`T_person`), which is c_3, as shown in Figure 4.

[5]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 (see Section 5).

In this paper, we assume 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 with time-varying properties:

$$\begin{aligned} B\_supertypes &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_type}\rangle\rangle \\ B\_subtypes &: \texttt{T\_history}\langle\texttt{T\_collection}\langle\texttt{T\_type}\rangle\rangle \end{aligned}$$

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`:

$$\begin{aligned} B\_superlattice &: \texttt{T\_history}\langle\texttt{T\_poset}\langle\texttt{T\_type}\rangle\rangle \\ B\_sublattice &: \texttt{T\_history}\langle\texttt{T\_poset}\langle\texttt{T\_type}\rangle\rangle \end{aligned}$$

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 5 shows the 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). Include 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. The history of the super-lattice of T is adjusted accordingly. 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`. This is done by the behavior application `T_employee`.*B_supertypes*.*B_insert* (`T_taxSource`,t_0). The history of the direct supertypes of `T_employee` would then be $\{<t_0, \{\texttt{T\_taxSource}\}>\}$.

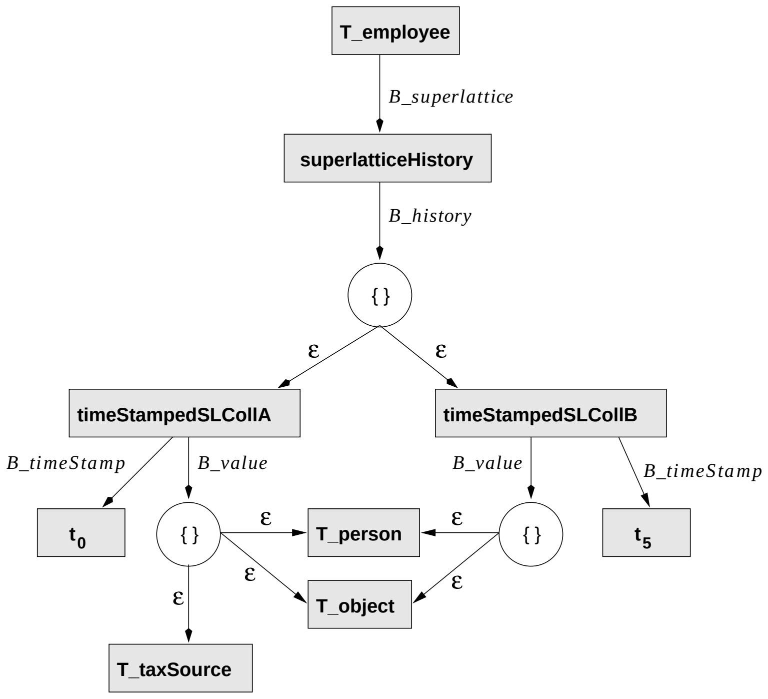


Figure 5: Supertype lattice history for type `T_employee`.

- The history of the collection of subtypes of type S is updated. This is performed by the behavior application $S.B\_subtypes.B\_insert(T, t)$. The history of the sub-lattice of S is adjusted accordingly. In this case, the history of the collection of subtypes of `T_taxSource` has to be updated using the behavior application `T_taxSource`.*B_subtypes*.*B_insert*(`T_employee`,t_0). The history of the direct subtypes of `T_taxSource` would then be $\{<t_0, \{\texttt{T\_employee}\}>\}$.
- 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.

  $\forall b \in S.B\_interface.B\_history.B\_last, \forall T' \mid T'$ subtype-of $T, T'.B\_inherited.B\_insert(b, t)$

  Behavior *B_last* is defined on type `T_history`$\langle$`T_X`$\rangle$ and it 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 `T_employee`.*B_inherited*.*B_insert*(*B_taxBracket*,t_0) behavior application performs this update. 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 3), 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 at time t (a direct supertype link to `T_object` cannot be dropped). Consider types T and S where S is the direct supertype of T. 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 supertypes of `T_employee` and the history of subtypes of `T_taxSource`. This is carried out using the behavior applications `T_employee`.*B_supertypes*.*B_remove*(`T_taxSource`,t_5) and `T_taxSource`.*B_subtypes*.*B_remove*(`T_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*,t_5) performs this update.

4.5 Queries

In this sub-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 that 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 are query objects) and meta-queries (which are query schema objects).

4.5.1 The TIGUKAT Query Language

The TIGUKAT Query Language (TQL[6]) is based on the SQL paradigm [Dat87] and its semantics is defined in terms of an 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:

select $< object\ variable\ list >$
[**into** $< collection\ name >$]
 from $< range\ variable\ list >$
[**where** $< boolean\ formula >$]

[6]TQL was developed before the release of OQL [Cat94]. It is quite similar to OQL in structure.

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 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 an existing collection or a collection returned as a result of a subquery; a subquery can be 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.

4.5.2 Query Examples

Example 4.1 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 4.2 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 4.3 Return all implementations of behavior *B_age* in type `T_person` before or 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 6. □

Example 4.4 Return all super-lattices of type `T_employee` before or 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 5. □

Example 4.5 Return the types that define behavior *B_age* with the same implementation as one of their supertypes at exactly the same time.

select T
from T **in C_type**, V **in** T.*B_supertypes*.*B_history*, S **in** V.*B_value*,
    i **in** *B_age*.*B_implementation*(T).*B_history*,
    j **in** *B_age*.*B_implementation*(S).*B_history*
where b **in** S.*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. □

5 Schema Change Propagation

5.1 Changing Implementations of Behaviors

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. Thus, a behavior with a computed function implementation can be considered an abstraction of a method in classical object models, whereas a behavior with a stored function implementation is an abstraction of an attribute (with "set" and "get" operations). The valid implementation changes for behaviors are shown in Table 2. The notation *computed*$_i$ (c_i) and *stored*$_i$ (s_i) refer to computed and stored functions respectively. The subscripts i and j are used to denote distinct functions. The term *undefined* is for the case when the behavior is undefined. The combinations *computed*$_i$ to *computed*$_i$ and *stored*$_i$ to *stored*$_i$ (which imply changes to the function code) are not included in the table because these do not reflect changes in function association. The *emphasized* entries represent user-level changes (i.e., by the schema designer) and the **bold** entry is a system-level change for reorganizing the internal representation of objects.

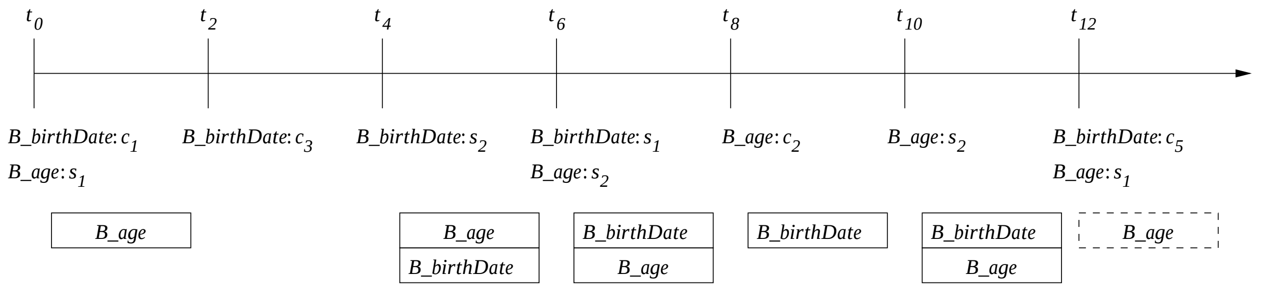
	Old Implementation	New Implementation
CC	*computed*$_i$	*computed*$_j$
CS	*computed*$_i$	*stored*$_j$
SS	**stored**$_i$	**stored**$_j$
SC	*stored*$_i$	*computed*$_j$
US	*undefined*	*stored*$_j$
UC	*undefined*	*computed*$_j$

Table 2: Valid implementation changes of a behavior in a type.

With the *B_implementation* behavior (defined in Section 4.3) we can determine the implementation of a behavior defined on a type at any time of interest. For example, Figure 6 shows the history of the implementations for behaviors *B_birthDate* and *B_age* on type `T_person`. A timeline representation and the result of

B_birthDate.*B_implementation*(`T_person`).*B_history* and *B_age*.*B_implementation*(`T_person`).*B_history* are shown. The implementation histories of *B_birthDate* and *B_age* return a collection of timestamped function objects. The value of each timestamped function is a computed or stored function. The timestamp of each timestamped function denotes the time interval during which the particular implementation is valid. The interface history of `T_person` is also shown for clarity. The *B_interface* behavior is defined in `T_type` and returns a history of the evolution of behaviors in a type. Each timestamped object in the history consists of a collection of behaviors that are valid during the associated time interval.

In the timeline representation, *B_X*:c_i or *B_X*:s_i denotes the association of a computed or stored function with behavior *B_X*. Moreover, for stored functions the subscript i refers to a location (e.g., a slot number) in an object representation that the stored function accesses. Each association is valid at a certain time t and remains valid until it is modified or removed. An object representation (i.e., the state of an object) consists of a number of slots for holding information carried by the object. The representations of objects at different times according to the stored functions associated with behaviors at those times are depicted by the boxes labeled with behaviors. For example, between times t_4 and t_6, the object representation consists of two slots – the first slot is for the stored implementation of behavior *B_age* and the second is for *B_birthDate*. Between times t_8 and t_{10}, the object representation consists of only one slot which is for *B_birthDate*, since during this interval, *B_age* is associated with the computed function, c_2.



Implementation history of behavior *B_birthDate* for type T_person:
{<[t_0, t_2), c_1 >, <[t_2, t_4), c_3 >, <[t_4, t_6), s_2 >, <[t_6, t_{12}), s_1 >, <[t_{12}, *now*], c_5 >}

Implementation history of behavior *B_age* for type T_person:
{<[t_0, t_6), s_1 >, <[t_6, t_8), s_2 >, <[t_8, t_{10}), c_2 >, <[t_{10}, t_{12}), s_2 >, <[t_{12}, *now*], s_1 >}

Interface history of type T_person:
{<[t_0, *now*], {*B_birthDate* , *B_age*}>}

Figure 6: Implementation histories of behaviors *B_birthDate* and *B_age* for type `T_person` and object representations.

Figure 6 is used to describe how the implementation changes in Table 2 are maintained by implementation histories. Prior to time t_0 both behaviors are undefined and at time t_0, *B_age* is defined as stored (US) and *B_birthDate* is defined as computed (UC). At time t_2, the implementation of *B_birthDate* changes from the computed function c_1 to the computed function c_3 (CC). At time t_4, the implementation of *B_birthDate* changes from the computed function c_3 to the stored function s_2 (CS). At time t_6, the implementation of

B_birthDate changes from the stored function s_2 to the stored function s_1 (SS) and *B_age* changes from s_1 to s_2 (SS). At time t_8, the implementation of *B_age* changes from the stored function s_2 to the computed function c_2 (SC).

Note that at time t_{12} the binding of the behavior *B_birthDate* changes from the stored function s_1 to the computed function c_5. Since all object representations at time t_{12} require only one slot, the change to *B_birthDate* forces a change to *B_age* so that at time t_{12} behavior *B_age* accesses slot one instead of slot two. Furthermore, the implicit implementation change of *B_age* is from a stored function to a stored function (SS) which is a system managed change and therefore is transparent to the user. The implicit implementation change of *B_age* is reflected in its history by the two entries $<[t_{10}, t_{12}), s_2>$ and $<[t_{12}, now], s_1>$. In general, the slots of an object representation are reorganized (i.e., an implicit change occurs) whenever a stored to computed implementation change removes a slot other than the last slot of an object's representation. The system can also rearrange slots as part of an implementation change, necessitating internal system organization as at t_6.

Using the results of this section, one can reconstruct the implementations of behaviors, and the object representations for any type at any time t. The implementation of *B_birthDate* at time t_7 (where $t_6 < t_7 < t_8$) is given by *B_birthDate*.$[t_7]$*B_implementation*(`T_person`) which is s_1. Similarly, the implementation of *B_age* at time t_7 is given by the behavior application *B_age*.$[t_7]$*B_implementation*(`T_person`) which is s_2. Since there are two stored functions, this implies a two slot representation for objects at time t_7. That is, *B_birthDate* accesses slot one using stored function s_1 and *B_age* accesses slot two using stored function s_2.

5.2 Change Propagation

The behaviors applicable to an object at its creation is the set of behaviors defined on the type of the object. The implementations of these behaviors are those that exist in the implementation histories for the type at creation time (which can be obtained by means of the *B_created* behavior defined on `T_object`).

When changes occur to the type definition and behavior implementations, we do not immediately propagate them to the instances. Instead, the old version of the schema is maintained and the change is recorded in the proper behavior histories. For adding and dropping behaviors, we make changes to the interface histories (*B_native*, *B_inherited*, *B_interface*) of the type [GSÖP97]. For changes in the implementation of a behavior we make changes to the implementation history (*B_implementation*) of the behavior (as described in Section 5.1).

The propagation of changes to the instances is delayed until the instances are accessed. In our model this occurs when a behavior is applied to an object. At that point in time, the behavior is coerced to reflect the implementation changes that have occurred on the behavior since the last behavior application. These changes are recorded in the *B_changes* behavior which is defined in `T_type`. The signature for *B_changes* is as follows:

$$B\_changes : \texttt{T\_list}\langle \texttt{T\_timeStamp}, \texttt{T\_behavior}\rangle$$

The result of *B_changes* is a list of (*timestamp*, *behavior*) pairs. Each pair denotes the time at which the implementation for the behavior has changed. The *B_changes* list is used by our behavior dispatch routine (defined in Section 5.3) to determine the most recent coercion time of the behavior that is applied to an object. The time is used as a reference point for finding an appropriate implementation of the behavior.

A novel characteristic of our model is that the basic unit of object coercion is individual behaviors. More specifically, objects from the older schema are coerced to the newer schema one behavior at a time. Thus, portions of an object (i.e., some behaviors) may correspond to older schema, while other portions correspond to newer schema.

In order to model the representations of an object over time (resulting from changes to its structure), we use the T_history mechanism that is described in Section 3.2. For example, type T_person $=$ T_history$\langle$T_person$'\rangle$ is created to maintain the representations of a person over time. Therefore, if joe is an object of type T_person, then joe represents the history of its different structural changes over time. The value of each timestamped object in an object o of type T_X $=$ T_history$\langle$T_X$'\rangle$ is called a *representation object* of o, and is of type T_X$'$.

In this paper we are using the notation T_history$\langle$T_X$'\rangle$ to denote a type whose schema changes we wish to record. However, an actual user of the ODBMS would simply use the notation T_X and indicate at type creation time that the schema changes should be recorded for this type. The ODBMS would then create T_X as T_history$\langle$T_X$'\rangle$ and the user would never deal directly with T_X$'$. However, we will continue to use the notation T_history$\langle$T_X$'\rangle$ in this paper since we wish to show how our model and algorithms for schema changes can be defined using only our existing temporal model and without introducing any new concepts. The user does not actually see or use T_history.

Whenever a change to the representation of an object occurs due to coercion of one of the behaviors of its base type[7], the change is recorded by updating the history of its structural changes. Thus, an object of type T_history$\langle$T_X$'\rangle$ is generic in the sense that it consists of all its representation objects over time. This is called the *generic instance* of the object. The default representation object of a generic instance is the most current representation object in the history of its structural changes. The individual representation objects in the history denote how the object existed at certain times in the past. Each of these representation objects is called a *structural instance* of the object and has type T_X$'$. In essence, the *B_changes* list of the type T_X$'$ and the objects of type T_history$\langle$T_X$'\rangle$ (potentially) "grow" with each behavior application if that behavior has been modified since its last application to the object.

Example 5.1 Consider Figure 7, which contains the object joe created as an instance of type T_history $\langle$T_person$'\rangle$. Assuming no behavior application has occurred, the figure shows the created time and the representation objects of joe. It also shows the changes list of T_person[8]. The notation $o@t_i$ is used to denote the structural instance of an object o at time t_i. Object joe is created at time t_0. The default properties and implementations for this object are those that exist at time t_0, namely, *B_birthDate*:c_1 and *B_age*:s_1 (see Figure 6). There are no entries in the changes list of T_person since no coercion of any behaviors of T_person$'$ has taken place yet. Therefore, joe has only one structural instance $joe@t_0$, the representation object that existed at the creation time of joe.

Now suppose joe is accessed at time t_7 through the behavior application joe.$[t_7]B\_birthDate$. The *B_birthDate* behavior is coerced to a version at t_7, and joe is updated. These changes are shown in Figure 8.

[7]The base type of an object o of type T_history<T_X'> is the type T_X'.

[8]Although we say the changes list of T_person, it is actually computed from the base type as T_person$'$.*B_changes*.

$$\begin{aligned}
\text{joe}.\textit{B\_created} &= t_0 \\
\text{joe}.\textit{B\_history} &= \{<[t_0, now], joe@t_0>\} \\
\texttt{T\_person}'.\textit{B\_changes} &= \{\,\}
\end{aligned}$$

Figure 7: Initial representation of joe and changes list of `T_person`.

$$\begin{aligned}
\text{joe}.\textit{B\_created} &= t_0 \\
\text{joe}.\textit{B\_history} &= \{<[t_0, t_4), joe@t_0>, <[t_4, t_6), joe@t_4>, <[t_6, now], joe@t_6>\} \\
\texttt{T\_person}'.\textit{B\_changes} &= \{<t_0, B\_birthDate>, <t_2, B\_birthDate>, <t_4, B\_birthDate>, \\
&\quad <t_6, B\_birthDate>\}
\end{aligned}$$

Figure 8: The representation objects of joe and the changes list of `T_person` after behavior application of *B_birthDate* at time t_7.

Since this is the first behavior application of *B_birthDate* on object joe, the *B_changes* list of `T_person` is updated with the times of all implementation changes that took place on behavior *B_birthDate* prior to time t_7. From Figure 6 we see that these times are t_0, t_2, t_4, and t_6. The behavior coercions at times t_4 and t_6 lead to changes in the representation of object joe. At t_4, the implementation of *B_birthDate* changes from a computed to a stored function and at t_6, the implementation changes from a stored to a stored function. These changes in structural representation are recorded in joe as shown in Figure 8. Note that changes to *B_age* are not yet recorded since we use deferred coercion and *B_age* has not yet been applied at t_7. □

5.3 Temporal Behavior Dispatch

The preceding sections establish mechanisms for maintaining the histories of behavior implementations and the representations of temporal objects. We now illustrate the behavior dispatch process that occurs when a behavior b is applied to an object o at a given time t. We denote this application as $o.[t]b$. The time component is optional and if left out the current time *now* is assumed.

Figure 9 provides an overview of the dispatch process. Detailed explanations of the various steps are given in the sections that follow. In general, a dispatch mechanism takes a type and a behavior and returns the function associated with the behavior for the given type [HS97]. In this paper, the dispatch mechanism is extended to take a third argument; namely, time.

A behavior application is first checked for temporal validity. It is considered valid if the object o exists at time t and behavior b is defined in the interface of o's base type at time t. A temporally invalid behavior application generates the only possible error while dispatching. As illustrated in Figure 9, this error is caught early in the dispatch process, which is a good feature of the design.

For a valid application, the *B_changes* list of the base type of o is updated. A search is made in b.*B_implementation* for implementation changes that took place before or at the same time as t. The *B_changes* list of the base type of o is then updated with all implementation changes that have not yet been recorded in *B_changes*. Object o is then updated if neccssary.

The appropriate representation object $o@t$ of o, and the appropriate implementation f of b for the base

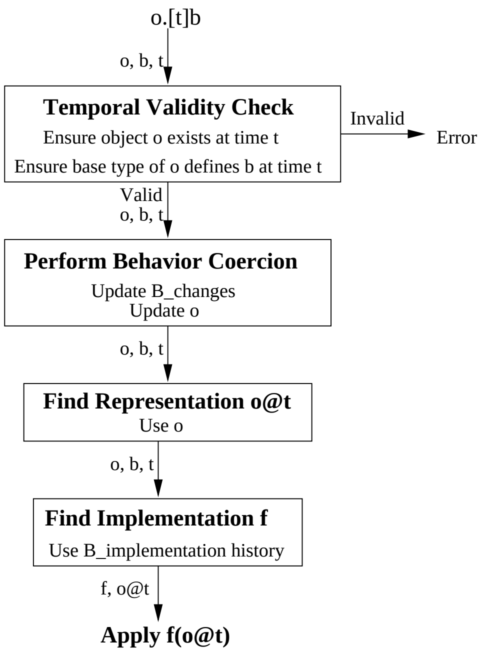


Figure 9: Dispatch process for applying a behavior b to an object o at time t.

type of o at time t are then retrieved by indexing into o and the *B_implementation* history of b, respectively. Finally, function f is applied to the representation object $o@t$. Examples of this process are given in Section 5.3.2; after the algorithms for the dispatch semantics are discussed.

5.3.1 Dispatch Semantics

In order for a behavior application to be valid, object o must exist at time t and behavior b must be defined in the interface of the base type of o at time t. The temporal validity check algorithm, Algorithm 5.1, performs this test in the form of a logical expression.

Algorithm 5.1 *TemporalValidity:*

Input: An object o, a behavior b and a time t

Output: True if the application is valid, false otherwise

Procedure:

$$\begin{aligned}
\textbf{return} \quad & (t.B\_within(o.B\_lifespan(o.B\_mapsto.B\_classof.B\_shallowExtent)) && (1)\\
& \wedge\ \exists x(x \in o.B\_mapsto.B\_baseType.B\_interface.B\_history && (2)\\
& \quad \wedge\ t.B\_within(x.B\_timeStamp) && (3)\\
& \quad \wedge\ b \in x.B\_value)) && (4)
\end{aligned}$$

The first part of the expression (1) checks that o exists at time t by testing whether time t lies within[9] the lifespan of o in the class of its associated type. In the second part of the expression, *o.B_mapsto.B_baseType*.

[9]The *B_within* behavior is defined on `T_timeStamp` and checks whether one timestamp is within another timestamp.

B_interface.*B_history* returns the interface history for the base type of object o. This history is searched for an entry x that satisfies (3), which checks that time t lies within the timestamp of entry x, and (4), which checks that behavior b is part of the collection of behaviors defined in the interface of the type at this time. If all conditions are satisfied, the behavior application is valid.

If the validity test is satisfied, the next step is to coerce behavior b to the implementation changes that took place prior to time t. Algorithm 5.2 performs this operation.

Algorithm 5.2 *Coerce:*

Input: An object o, a behavior b and a time t

Procedure:

$$o.B\_mapsto.B\_baseType.B\_updateChanges(t, b.B\_implementation(o.B\_mapsto.B\_baseType)) \tag{5}$$

$$o.B\_updateRep(t, b.B\_implementation(o.B\_mapsto.B\_baseType)) \tag{6}$$

In step (5), the *B_changes* list of the base type of o is updated with the implementation changes that took place on behavior b at or before time t. The *B_updateChanges* behavior, defined on the `T_type` type, performs this update by taking t and *B_implementation* of b as arguments. It searches *B_implementation* of b for implementation changes that took place before or at the same time as t and updates the *B_changes* list with all implementation changes that have not yet been recorded in *B_changes*. For example, the behavior application $\texttt{T\_person}'.B\_updateChanges(t_7, B\_birthDate.B\_implementation(\texttt{T\_person}'))$ updates the *B_changes* list of the base type of joe ($\texttt{T\_person}'$) during the behavior application joe.$[t_7]B\_birthDate$. The updated *B_changes* list is shown in Figure 8. The object o is then updated if neccessary (6). The *B_updateRep* behavior, defined on base type objects, performs this update. For each behavior implementation change at time t_i that leads to a change in the representation of o, *B_updateRep* updates o with the appropriate representation object with respect to time t_i and the time interval during which it was valid. The behaviors applicable to the representation object are those that exist in the interface of its type at t_i. The implementations of these behaviors are those that exist in the implementation histories for the type at t_i. The stored functions at t_i determine the initial state of the representation object.

Algorithm 5.3 performs the simple task of returning the appropriate representation object of o at time t.

Algorithm 5.3 *Representation:*

Input: An object o and time t

Output: An object with its representation at time t

Procedure:

```
    return $o.B\_validObject(t).B\_value$
```

The appropriate implementation f of b for the base type of o at time t is then retrieved from the *B_implementation* history of b. Algorithm 5.4 finds and returns this implementation.

Algorithm 5.4 *Implementation:*

Input: An object o, a behavior b and a time t

Output: The function that implements behavior b for object o at time t

Procedure:

```
    return $b.B\_implementation(o.B\_mapsto.B\_baseType).B\_validObject(t).B\_value$
```

A final step of the dispatch mechanism is the execution of the function returned from Algorithm 5.4 to the representation object returned by Algorithm 5.3. We use the *B_execute* behavior on functions to accomplish this. The relationships between all the algorithms are shown in Algorithm 5.5.

Algorithm 5.5 *Dispatch:*

Input: An object o, a behavior b and a time t

Output: An object resulting from the application $o.[t]b$

Procedure:

```
    if TemporalValidity(o,b,t) then
        $Coerce(o, b, t)$
        $o@t \leftarrow Representation(o, t)$
        $f \leftarrow Implementation(o, b, t)$
        $f.B\_execute(o@t)$
    else
        INVALID: object $o$ does not exist at time $t$
                 or behavior $b$ not defined in the interface of $o$'s base type at time $t$
```

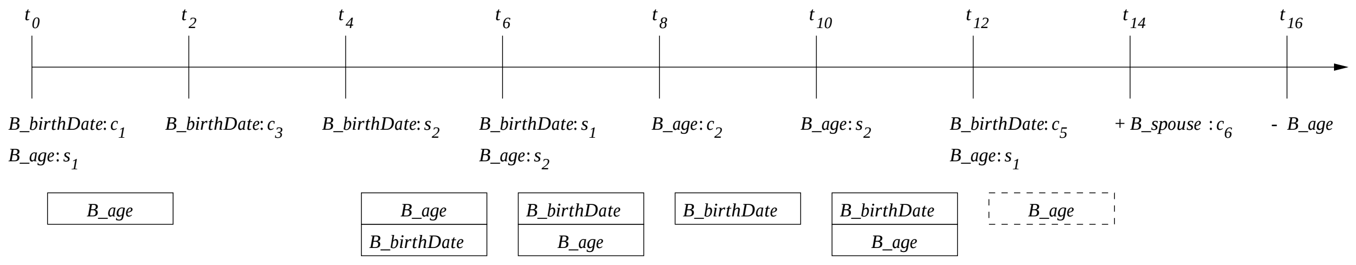
5.3.2 Dispatch Examples

For the following examples, consider Figure 10, which extends the timeline of type `T_person` in Figure 6 by adding a behavior *B_spouse* with the computed implementation c_6 at time t_{14} and dropping the behavior *B_age* at time t_{16}. Note that an object representation will not change by adding behavior *B_spouse* and the representations will be empty after behavior *B_age* is dropped. For this example, *now* $> t_{16}$.

Several example behavior applications using time are presented to show how the dispatch process is followed in order to determine the proper implementation and state instance that are appropriate at the given time of interest. We assume the behavior applications take place in chronological order.

Example 5.2 Behavior application joe.$[t_7]B\_birthDate$ (assuming no previous behavior application has taken place)

Validity: Object joe was created at time t_0 and exists at time *now*. Therefore, the lifespan of joe is the time interval $[t_0, now]$. Since t_7 in within this interval (i.e., lifespan), the object part of the behavior application is valid. The base type of joe is `T_person`$'$. The interface of `T_person`$'$ at time t_7 is $\{B\_birthDate, B\_age\}$. Since *B_birthDate* is part of this interface, the behavior part of the application is valid and, thus, the validity test is satisfied.



B_birthDate.B_implementation (T_person').*B_history*
$\{<[t_0, t_2), c_1>, <[t_2, t_4), c_3>, <[t_4, t_6), s_2>, <[t_6, t_{12}), s_1>, <[t_{12}, now], c_5>\}$

B_age.B_implementation (T_person').*B_history*
$\{<[t_0, t_6), s_1>, <[t_6, t_8), s_2>, <[t_8, t_{10}), c_2>, <[t_{10}, t_{12}), s_2>, <[t_{12}, t_{16}], s_1>\}$

B_spouse.B_implementation (T_person').*B_history*
$\{<[t_{14}, now], c_6>\}$

T_person'. *B_interface.B_history*
$\{<[t_0, t_{14}), \{B\_birthDate, B\_age\}>, <[t_{14}, t_{16}), \{B\_birthDate, B\_age, B\_spouse\}>, <[t_{16}, now], \{B\_birthDate, B\_spouse\}>\}$

Figure 10: Example showing effects on implementation histories of first adding and then dropping a behavior.

Coerce: The next step is to update the *B_changes* list of the base type of o and the representation history of o. These updates are performed by the behavior applications $\texttt{T\_person}'.B\_updateChanges(t_7, B\_birthDate.B\_implementation(\texttt{T\_person}'))$ and joe.$B\_updateRep(t_7, B\_birthDate.B\_implementation(\texttt{T\_person}'))$, respectively. The updated *B_changes* list and representation history of o is shown in Figure 8.

Representation: The behavior application joe.$B\_validObject(t_7).B\_value$ returns joe$@t_6$, which is the appropriate representation object of joe at time t_7 (see Figure 8).

Implementation: The behavior application $B\_birthDate.B\_implementation(\texttt{T\_person}').B\_validObject(t_7).B\_value$ returns the appropriate implementation of *B_birthDate* for type `T_person` at time t_7, which is the stored function s_1.

Dispatch: To complete the dispatch of the behavior, the stored function s_1 is executed using the representation object joe$@t_6$ as an argument. This will access the first slot of the representation of joe at t_6. The represenation of joe at t_7 is the same as the one at t_6, so the behavior application accesses the appropriate birthdate slot of joe at t_7.

$\square$

Example 5.3 Behavior application joe.$[t_3]B\_birthDate$
The validity test is satisfied. The *B_birthDate* has already been coerced to the implementations at times t_0 and t_2 since the entries $<t_0, B\_birthDate>$ and $<t_2, B\_birthDate>$ exist in the changes list of `T_person`. Therefore, *B_changes* and o remain unchanged. The representation object at t_3 is joe$@t_0$ (see Figure 8) and the implementation chosen at t_3 is the computed function c_3. The function c_3 is then applied to joe$@t_0$. $\square$

Example 5.4 Behavior application joe.$[t_{12}]B\_birthDate$
The validity test is satisfied. In Algorithm 5.2, the implementation change of *B_birthDate* at time t_{12} is recorded in the *B_changes* list, and the representation history for object joe also changes since the implementation for *B_birthDate* changes from a stored function (time t_6) to a computed function (time t_{12}). Figure 11 shows the changes in *B_changes* and the representation history of joe.

$$\begin{array}{rcl}
\text{joe}.\textit{B\_created} & = & t_0 \\
\text{joe}.\textit{B\_history} & = & \{<[t_0,t_4),joe@t_0>, <[t_4,t_6),joe@t_4>, <[t_6,t_{12}),joe@t_6>, \\
 & & <[t_{12},now],joe@t_{12}>\} \\
\texttt{T\_person}'.\textit{B\_changes} & = & \{<t_0,B\_birthDate>, <t_2,B\_birthDate>, <t_4,B\_birthDate>, \\
 & & <t_6,B\_birthDate>, <t_{12},B\_birthDate>\}
\end{array}$$

Figure 11: The representation objects of joe and the changes list of `T_person` after behavior application of *B_birthDate* at time t_{12}.

The appropriate implementation for *B_birthDate* at t_{12}, which is the computed function c_5, is then applied to joe$@t_{12}$, which is the representation object of joe at t_{12}. □

Example 5.5 Behavior application joe.$[t_{10}]B\_age$
Now suppose a different behavior (*B_age*) is applied to object joe. The validity test is satisfied. The *B_changes* list is updated with the times of all implementation changes that took place on behavior *B_age* prior to time t_{10}. From Figure 10 we see that these times are t_0, t_6, t_8, and t_{10}. The behavior coercions at all these times lead to changes in the structural representation of object joe. At t_0, the implementation of *B_age* changed from undefined to stored (UC), at t_6 the implementation changed from stored to stored (SS), at t_8 the implementation changed from stored to computed (SC), and at t_{10} the implementation changed from computed to stored (CS). The new value of *B_changes* and the structural representation history of joe are shown in Figure 12.

$$\begin{array}{rcl}
\text{joe}.\textit{B\_created} & = & t_0 \\
\text{joe}.\textit{B\_history} & = & \{<[t_0,t_4),joe@t_0>, <[t_4,t_6),joe@t_4>, <[t_6,t_8),joe@t_6>, \\
 & & <[t_8,t_{10}),joe@t_8>, <[t_{10},t_{12}),joe@t_{10}>, <[t_{12},now],joe@t_{12}>\} \\
\texttt{T\_person}'.\textit{B\_changes} & = & \{<t_0,B\_birthDate>, <t_0,B\_age>, <t_2,B\_birthDate>, \\
 & & <t_4,B\_birthDate>, <t_6,B\_birthDate>, <t_6,B\_age>, \\
 & & <t_8,B\_age>, <t_{10},B\_age>, <t_{12},B\_birthDate>\}
\end{array}$$

Figure 12: The representation objects of joe and the changes list of `T_person` after behavior application of *B_age* at time t_{10}.

Having updated the *B_changes* list and o, the representation object $joe@t_{10}$, and the implementation s_2 are returned from Algorithms 5.3 and 5.4 at time t_{10}. We can now apply s_2 to $joe@t_{10}$. □

Example 5.6 Behavior application joe.$[now]B\_age$
This fails the validity test because behavior *B_age* is not part of the interface of `T_person′` at time now. □

Example 5.7 Behavior application joe.$[t_{15}]B\_spouse$
The validity test is satisfied. An appropriate entry $< t_{14}, B\_spouse >$ is added to the changes list of T_person. The representation history of joe remains unchanged since the implementation change is from an undefined function to a computed one (UC). The representation object at time t_{14} is $joe@t_{12}$ (see Figure 12) and the implementation is the computed function c_6. The function c_6 is then applied to $joe@t_{12}$. □

Example 5.8 Behavior application jane.$[t_7]B\_age$
Suppose the object jane was created at time t_6. The validity test is satisfied. The changes list of T_person remains unchanged since it has already been updated with the implementation changes of *B_age* prior to t_7 (see Figure 12). The structural representation history of jane however, is updated to reflect the behavior coercions that took place at or after jane was created and before or at t_7. This is shown in Figure 13.

$$\boxed{\begin{array}{lcl} \text{jane.}\textit{B\_created} & = & t_6 \\ \text{jane.}\textit{B\_history} & = & \{<[t_6, now], jane@t_6>\} \end{array}}$$

Figure 13: The representation objects of jane after behavior application of *B_age* at time t_7.

The time t_7 is used to find the appropriate representation object for jane and the correct implementation of *B_age* for type T_person$'$. The representation object chosen is $jane@t_6$ and the implementation returned is the stored function s_2. This function is then applied to $jane@t_6$. □

6 Immediate Object Conversions

The temporal infrastructure proposed in this paper is sufficiently powerful to support schema change approaches other than the deferred coercion strategy that we have developed. In this section, we show how the immediate object coercion approach of schema change propagation can be implemented using the model presented in Section 5. In this case, changes are immediately propagated to the instances. In our model, this would mean that each time the implementation of a behavior changes, the behavior is coerced to the newer implementation at that time and the structural representations of all objects of that type are updated, if necessary. These changes are recorded in the *B_implementation* and *B_changes* behaviors, respectively. Figure 14 shows the changes list for T_person and the representation history of object joe when immediate object coercion is used for the behavior implementation changes shown in Figure 10.

The *B_changes* behavior for T_person$'$ shows that each time the implementation of a behavior changes after the object was created, the behavior is coerced to the newer implementation since immediate object coercion is used. For example, after joe is created, the implementation of behavior *B_birthDate* changes at times t_2, t_4, t_6, and t_{12} (see Figure 10). Subsequently, *B_birthDate* is also coerced to the newer implementations at these times. This is shown in the changes list of T_person. The representation history of an object is only updated when a change to the representation of an object occurs due to the coercion of one of its behaviors. For example, although the behavior *B_birthDate* is coerced to a newer implementation at time t_2, the representation of joe is unaffected since the implementation is changed from one computed function

$$\begin{array}{rcl}
\textsf{joe}.\textit{B\_created} & = & t_0 \\
\textsf{joe}.\textit{B\_history} & = & \{<[t_0,t_4),joe@t_0>,<[t_4,t_6),joe@t_4>,<[t_6,t_8),joe@t_6>, \\
 & & <[t_8,t_{10}),joe@t_8>,<[t_{10},t_{12}),joe@t_{10}>,<[t_{12},t_{16}),joe@t_{12}>, \\
 & & <[t_{16},now],joe@t_{16}>\} \\
\textsf{jane}.\textit{B\_created} & = & t_6 \\
\textsf{jane}.\textit{B\_history} & = & \{<[t_6,t_8),jane@t_6>,<[t_8,t_{10}),jane@t_8>,<[t_{10},t_{12}),jane@t_{10}>, \\
 & & <[t_{12},t_{16}),jane@t_{12}>,<[t_{16},now],jane@t_{16}>\} \\
\texttt{T\_person}'.\textit{B\_changes} & = & \{<t_0,B\_birthDate>,<t_0,B\_age>,<t_2,B\_birthDate>, \\
 & & <t_4,B\_birthDate>,<t_6,B\_birthDate>,<t_6,B\_age>, \\
 & & <t_8,B\_age>,<t_{10},B\_age>,<t_{12},B\_birthDate>, \\
 & & <t_{12},B\_age>,<t_{14},B\_spouse>\}
\end{array}$$

Figure 14: The representation objects of joe and jane, and the changes list of `T_person` for immediate object coercion.

to another computed function (see Figure 10). Therefore, joe is unchanged at t_2. A similar situation occurs at t_{14} for joe and jane when behavior *B_spouse* is added to type `T_person`$'$.

With immediate coercion, if a behavior implementation change at time t for a type T necessitates an update of the representation of an object, the change is recorded in the representation histories of *all objects* of type T that exist at time t. This is exemplified in Figure 14 where the tuples in representation histories of objects joe and jane (of type `T_person`) are updated at the same time after jane was created (from t_8 to now).

In the immediate coercion approach, Algorithm 5.2 is carried out at the time of behavior implementation change, and not during a behavior application process as was the case in deferred coercion. The only difference to the dispatch algorithm is that invocation of coerce is not necessary. The example below shows how the dispatch process is followed when immediate object coercion is used for the behavior application given in Example 5.2.

Example 6.1 Behavior application `joe`.$[t_7]B\_birthDate$
The validity test is satisfied. The appropriate representation object for joe at time t_7 is $joe@t_6$, while the appropriate implementation of *B_birthDate* for type `T_person`$'$ is the stored function s_1. We can now apply s_1 to $joe@t_6$. The function and representation are correct for joe since the implementation of behavior *B_birthDate* changed at time t_6 for this object, and *B_birthDate* was coerced to the new version at the same time. □

From the above example, we note that the function and the representation object obtained for joe using immediate object coercion are the same as those obtained in Example 5.2, in which deferred object coercion was used. The function chosen in both cases is the stored function s_1, and the representation object chosen for joe in both cases is $joe@t_6$. This equivalence of deferred and immediate object coercion strategies is neccessary.

7 Conclusion

In this paper, we present a new approach to schema evolution for ODBMS. This strategy is characterized by four novel concepts:

1. The schema evolution strategy is based on a uniform temporal model. Consequently, no special concepts are introduced for modeling schema changes that are expressed as changes to the behaviors defined on types. These changes are tracked using the same `T_history` mechanism that is used for modeling temporal changes to non-schema objects in the database.

2. In addition to schema changes, the strategy supports lossless recording of these changes, allowing historical queries.

3. The strategy supports both deferred (lazy) object update semantics and immediate object update semantics using the same basic algorithms. Only the application time of the algorithms is changed to produce the desired update semantics.

4. The granularity of schema changes is finer than traditional approaches that require a complete type change every time a single behavior changes. We handle behavior changes individually. This approach has two distinct advantages depending on whether deferred or immediate update semantics are used. If deferred update semantics are used, the finer granularity results in an even "lazier" update semantics. That is, when a modified behavior is applied to an object, only part of the object's structure needs to be updated to reflect changes for only that particular behavior. Updates due to other behavior changes are delayed until they are needed by other behavior applications. If immediate update semantics are used, then the update can be done more quickly since the system knows that changes to the affected type are localized to the single behavior that was just changed. This is important because a major drawback of immediate update semantics is the speed of the update.

Support for historical queries potentially has a profound effect on ODBMS behavior dispatch. In traditional behavior dispatch, each behavior on a type is bound to a single function (implementation) and dispatch is a mapping of behavior-type pairs to functions. With recorded schema evolution, each behavior may be bound to a different function at different times. Therefore, the dispatch process must map a three-tuple (behavior, type, time) to a function. Unfortunately, the domain of the temporal argument is very large compared to the domain of all behaviors (or all types), so standard dispatch techniques do not work very well. This paper provides a temporal dispatch algorithm to demonstrate that no new concepts need to be added to the schema evolution model to solve the temporal dispatch

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 inter-connection 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

[ALP91] J. Andany, M. Leonard, and C. Palisser. Management of Schema Evolution in Databases. In *Proc. 17th Int'l Conf. on Very Large Data bases*, pages 161–170, September 1991.

[Ari91] G. Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, (6):451–467, 1991.

[BFG97] E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera - A Temporal Object-Oriented Data Model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.

[BH89] A. Bjørnerstedt and C. Hültén. Version Control in an Object-Oriented Architecture. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 18, pages 451–485. Addison Wesley, September 1989.

[BKKK87] 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.

[Cat94] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.

[CITB92] W.W. Chu, I.T. Ieong, R.K. Taira, and C.M. Breant. A Temporal Evolutionary Object-Oriented Data Model and Its Query Language for Medical Image Management. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 53–64, August 1992.

[CPP95] 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.

[Dat87] C.J. Date. *A Guide to SQL Standard*. Addison Wesley, 1987.

[DM94] A.K. Das and M.A. Musen. A Temporal Query System for Protocol-Directed Decision Support. *Methods of Information in Medicine*, 33:358–370, 1994.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. 20th Int'l Conf. on Very Large Data Bases*, pages 261–272, September 1994.

[FMZ$^+$95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O_2 Object Database System. In *Proc. 21st Int'l Conf. on Very Large Data Bases*, pages 170–181, September 1995.

[GLÖS96] I.A. Goralwalla, Y. Leontiev, M.T. Özsu, and D. Szafron. Modeling Time: Back to Basics. Technical Report TR-96-03, University of Alberta, February 1996.

[GÖS97] 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*, 27(5):369 – 387, 1997.

[GSÖP97] I.A. Goralwalla, D. Szafron, M.T. Özsu, and R.J. Peters. Managing Schema Evolution using a Temporal Object Model. In *Proc. 16th International Conference on Conceptual Modeling (ER'97)*, pages 71–84, November 1997. Proceedings published as Lecture Notes in Computer Science, David Embley and Robert Goldstein (eds.), Springer-Verlag, 1997.

[GTC$^+$90] 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.

[HS97] W. Holst and D. Szafron. A General Framework For Inheritance Management and Method Dispatch in Object-Oriented Languages. In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, pages 276–301, 1997.

[KBCG90] 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.

[KC88] W. Kim and H-J. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. 14th Int'l Conf. on Very Large Data Bases*, pages 148–159, 1988.

[KFT91] M.G. Kahn, L.M. Fagan, and S. Tu. Extensions to the Time-Oriented Database Model to Support Temporal Reasoning in Medical Expert Systems. *Methods of Information in Medicine*, 30:4–14, 1991.

[KS92] W. Kafer and H. Schoning. Realizing a Temporal Complex-Object Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 266–275, June 1992.

[LGÖS97] J.Z. Li, I.A. Goralwalla, M.T. Özsu, and Duane Szafron. Modeling Video Temporal Relationships in an Object Database Management System. In *SPIE Proceedings of Multimedia Computing and Networking (MMCN97)*, pages 80–91, February 1997.

[LH90] B.S. Lerner and A.N. Habermann. Beyond Schema Evolution to Database Reorganization. In *ECOOP/OOPSLA '90 Proceedings*, pages 67–76, October 1990.

[MS90] E. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[MS92] S.R. Monk and I. Sommerville. A Model for Versioning of Classes in Object-Oriented Databases . In *10th British National Conference on Databases (BNCOD '92), Aberdeen, Scotland July 1992*, pages 42–58, July 1992.

[MS93] S. Monk and I. Sommerville. Schema Evolution in OODBs using Class Versioning. *ACM SIGMOD Record*, 22(3):16–22, September 1993.

[NR89] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

[ÖPS$^+$95] 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.

[Pet94] R.J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Management System*. PhD thesis, University of Alberta, 1994.

[PLÖS93] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System. In *Proc. Second Int'l. Conf. on Information and Knowledge Management*, pages 403–412, November 1993.

[PÖ93] 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 (ER'93)*, pages 37–49, December 1993.

[PÖ97] 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.

[PS87] 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.

[Rod91] J.F. Roddick. Dynamically Changing Schemas within Database Models. *Australian Computer Journal*, 23(3):105–109, 1991.

[Rod92] J.F. Roddick. SQL/SE- A Query Language Extension for Databases Supporting Schema Evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.

[Rod95] J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[RS91] 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.

[SC91] 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*, pages 431–442, 1991.

[Sjø93] Dag Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[Sno95] R. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[SZ86] 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.

[SZ87] A.H. Skarra and S.B. Zdonik. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pages 393–415. M.I.T. Press, 1987.

[WD92] 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, Tempe, USA, February 1992.