Temporal Extensions to a Uniform Behavioral Object Model*

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

Abstract

We define temporal extensions to a uniform, behavioral and functional object model by providing an extensible set of structural and behavioral abstractions to model various notions of time for different applications. We discuss the temporal semantics of inheritance by defining a lifespan behavior on objects in a collection. Finally, we give an elaborative example and show that temporal objects can be queried without adding any extra construct to the underlying query language.

1 Introduction

Most of the applications for which object management systems (OMSs) are expected to provide support exhibit some form of temporality. Some examples are the following: in engineering databases, there is a need to identify different versions of a design as it evolves; in multimedia systems, the video images are timed and synchronized with audio; in office information systems, documents are ordered based on their temporal relationships. In this paper we present temporal extensions to the TIGUKAT [1] OMS that is under development in the Laboratory for Database Systems Research of the University of Alberta.

Most of the research on temporal databases has concentrated on extending the relational model to handle time in an appropriate manner. These extensions can be divided into two main categories. The first approach uses normalized (1NF) relations in which special time attributes are added (called *tuple time-stamping*) and the history of an object (attribute) is modelled by several 1NF tuples [LJ88, Sno87]. The second approach uses non-normalized (N1NF) relations and attaches time to attribute values (called *attribute time-stamping*) in which the history of an object is modelled by a single N1NF tuple [Gad88, Tan86, TG89, Gor92].

There have been studies that have concentrated on the development of new temporal data models. One such model [SK86, SS87] models temporal data as a time sequence collection which is represented as a set of triples <surrogate, time, value>. An extension to the entity relationship (ER) model for handling time by incorporating the concept of lifespan to entities and relationships has been proposed by [EW90].

In the context of OMSs, [KC86] describes a model to handle complex objects and talks about the representation and temporal dimensions to support object identity. However, most of the emphasis is on the representation dimension. An extension to an object-based ER model to incorporate temporal structures and constraints in the data model is given in [RS91]. A corresponding temporal object-oriented algebra is given in [RS93]. A linear structural model with a continuous time domain is used to model time. Timestamps can be either time instants or time intervals. For every time-varying attribute in a class, a corresponding subclass is defined to represent the time sequence [SS87] (history) of that attribute, thus resulting in a large number of classes.

In [KS92], a state of a complex object is represented by the notion of a *time slice* which basically comprises of a time interval and the object which was valid during the interval. It is not clear however, how other timestamps and domains of time are supported for different applications and whether temporal constraints are provided to enforce the temporal semantics of inheritance.

---

*This research has been supported by the Natural Sciences and Engineering Research Council of Canada under research grant OGP0951.

[1] TIGUKAT (tee-goo-kat) is a term in the language of Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

In [DW92], variables and quantifiers are used to range over time. They base their work on abstract notions of time and talk about abstract time types to facilitate the modeling of various notions of time. However, they do not show how these abstract types fit in their primitive type lattice, neither do they formally define any behaviors on these abstract types.

Our work is conducted within the context of the TIGUKAT object management system which has a uniform, behavioral and functional object model. All access and manipulation of objects is restricted to the application of behaviors (implemented as functions) to objects. Every entity in the model is a first-class object [PÖS92]. TIGUKAT has a formal query model complete with a calculus and algebra definition. We have also designed a user language (called TQL) that loosely follows the ongoing SQL 3 standard definition [PLÖS93b].

Given the application domains TIGUKAT is expected to support, we have extended the object model to incorporate the time dimension and this is the focus of the present paper. Since TIGUKAT has similarities to OODAPLEX (in its functional nature), our approach has similarities to the work of Wuu and Dayal [DW92, WD92]. However, there are significant differences as well. The identifying characteristics of our work are the following:

1. We introduce an extensible set of primitive abstract time types and a rich set of behaviors to model the various notions of time, namely *linear*, *branching*, *discrete*, *continuous* and *dense*. Furthermore we identify three kinds of timestamps — *instants*, *intervals* and *spans* — and provide types for them. These types could be subtyped to model times with different granularities and durations. This would enable the design of a wide range of applications requiring different models of time to be carried out with ease and in a uniform manner.

2. We do not differentiate between object and attribute versioning. In our model, behavior histories are used to manage the properties of objects over time. This essentially models both approaches and alleviates the need to handle each approach in a different manner.

3. We define a more general and formal lifetime behavior which ranges not only over objects in classes, but in collections as well. This models the temporal semantics of inheritance in a uniform manner, even when multiple subtyping is involved.

4. Our temporal extensions can be incorporated within the query model without extending it with any additional constructs. Hence, the underlying object calculus remains unchanged with the temporal extensions. This enables temporal and non-temporal objects to be queried in uniform manner, which could be useful in query optimization.

The rest of the paper is organized as follows. In Section 2, we briefly describe the TIGUKAT object model. Section 3 outlines the extension to the object model to support temporal constructs and temporal semantics of inheritance. In Section 4, we describe our query language, and demonstrate its use in Section 5 by posing several queries on a geographic information system example. Finally, Section 6 concludes the paper.

2 The TIGUKAT Object Model

TIGUKAT object model [PÖS92] is *behaviorally* defined with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is restricted to the application of behaviors (operations) upon objects, and the model is *uniform* in that every entity within the model has the status of a *first-class object*.

Uniformity in TIGUKAT is similar to the approaches of DAPLEX [Shi81], its object-oriented counterpart OODAPLEX [Day89]. Furthermore, we adopt another significant aspect of these models: their functional approach in defining behaviors. However, we go further by including enhanced functionality along with a full set of precise specifications and an integration with an example structural counterpart. In this section we give only a general overview of TIGUKAT in order to introduce its fundamental concepts. For the complete and formal model specification, including the structural counterpart, we refer the reader to [PÖS92].

An *object* is the fundamental concept in TIGUKAT. Every component of information, including its semantics, is uniformly represented as objects in TIGUKAT. This means that at the most basic level, every expressible element in the model incorporates at least the semantics of our primitive notion for “object.”

The foundation of the model is constructed on a number of primitive objects which include: *atomic entities* such as *reals, integers, strings, characters, etc.*; *types* for defining and structuring the information carried by common objects, including the operations which may be performed on them, within a centralized framework for these objects; *behaviors* for specifying the semantics of the operations which may be performed on objects; *functions* for specifying the implementations of behaviors over various types; *classes* for the automatic classification of objects related through their types; and *collections* for supporting heterogeneous user-definable groupings of objects.

Objects are defined as *(identity, state)* pairs where *identity* represents a unique, immutable system managed object identity and *state* represents the information carried by the object. Thus, our model supports *strong object identity* [KC86], meaning every object has a unique existence within the model. However, this does not preclude application environments (such as object programming languages) from having many *references* (or *denotations*) to objects which need not be necessarily unique and may even change depending on the scoping rules of the application. On the other hand, the *state* of an object *encapsulates* the information carried by that object. More specifically, the state encapsulates the *denotations* of objects and hides the structure and implementation of the information carried by that object. The access and manipulation of an object's state occurs exclusively through the application of behaviors. In this way our model represents the message-based approaches such as Smalltalk [GR85] and OODAPLEX [Day89].

We separate the means for defining the characteristics of objects (i.e., a *type*) from the mechanism for grouping instances of a particular type (i.e., a *class*). A *type* specifies an object structure, behaviors and their implementations for objects created using the type as a template. Thus, a type serves as an information repository (template) of characteristics common among all objects of that particular type. Types are organized into a lattice structure using the notion of *subtyping* which promotes software reuse and incremental type development.

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental, but distinct, construct of a type responsible for managing all instances created using that type as a template (the collection of all instances of a type known as the *extent* of the type). Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. In other words, the model enforces a total (into) mapping *classof* from objects into classes and a total, injective (one-to-one and into) mapping *typeof* which maps each class to a unique type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types* (i.e., $object \Longrightarrow class \Longrightarrow type$). Another unique feature of classes is that object creation occurs only through a class using its associated type as a template for the creation. Defining object, type and class in this manner introduces a clear separation of these concepts.

In addition to classes, we define a *collection* as a more general grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, no object creation may occur through a collection; object creation occurs only through classes. This means that collections only form groupings of existing objects. Second, an object may exist in any number of collections, but its participation in classes is restricted by the lattice structure on types. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the subtype lattice whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, whereas a class groups the entire extension of a single type and its subtypes (i.e. homogeneous objects based on inclusion polymorphism), a collection may be heterogeneous in the sense that it can contain objects which may be of different types. Classes in our model are similar to the grouping constructs in Iris [FBC+87] O++ [AG89], ObjectStore [LLOW91] and Orion [BCG+87], while collections resemble those in EXODUS [CDV88], ENCORE [SZ90], GEMSTONE [MS87] and O_2 [LRV88].

In TIGUKAT, we properly define *class* as a specialization (subtype) of *collection* which introduces a clean semantics between the two and allows the model to utilize both constructs in an effective manner. For example, the targets and results of queries are typed collections of objects. Now, targets also include classes because of the specialization of classes on collections.

Two other fundamental notions of TIGUKAT are *behaviors* and the objects which implement them called *functions* (also known as *methods*). In the same way as object specifications (types) are separated from the groupings of their instances (classes and collections), we separate the definition of a behavior from its possible implementations (functions/methods). Behaviors provide the only means of operating upon objects and define a semantics which describe their functionality. Objects supporting the functionality of a particular behavior must have that behavior incorporated into the interface of their type in order for it to be applicable. Functions implement the semantics of behaviors; we say they provide the *operational* semantics of the behavior. The implementation of a particular behavior may vary over the types which support it. Nonetheless, the semantics of the behavior remains constant and unique over all types supporting that behavior. The implementation of a behavior may consist of runtime calls to executable code which is known as a *computed* function. Alternatively, it may simply be a reference to an existing object in the database in which case it is called a *stored* function. The uniformity of TIGUKAT object model considers each behavioral application as the invocation of a function, regardless of how the function is implemented (i.e., stored, computed, etc.).

The primitive type system of TIGUKAT is shown in Figure 1 with the type `T_object` as the root of the lattice. We use the following prefixes: type objects by `T_`, class objects by `C_`, collection objects by `L_` and behavior objects by `B_`

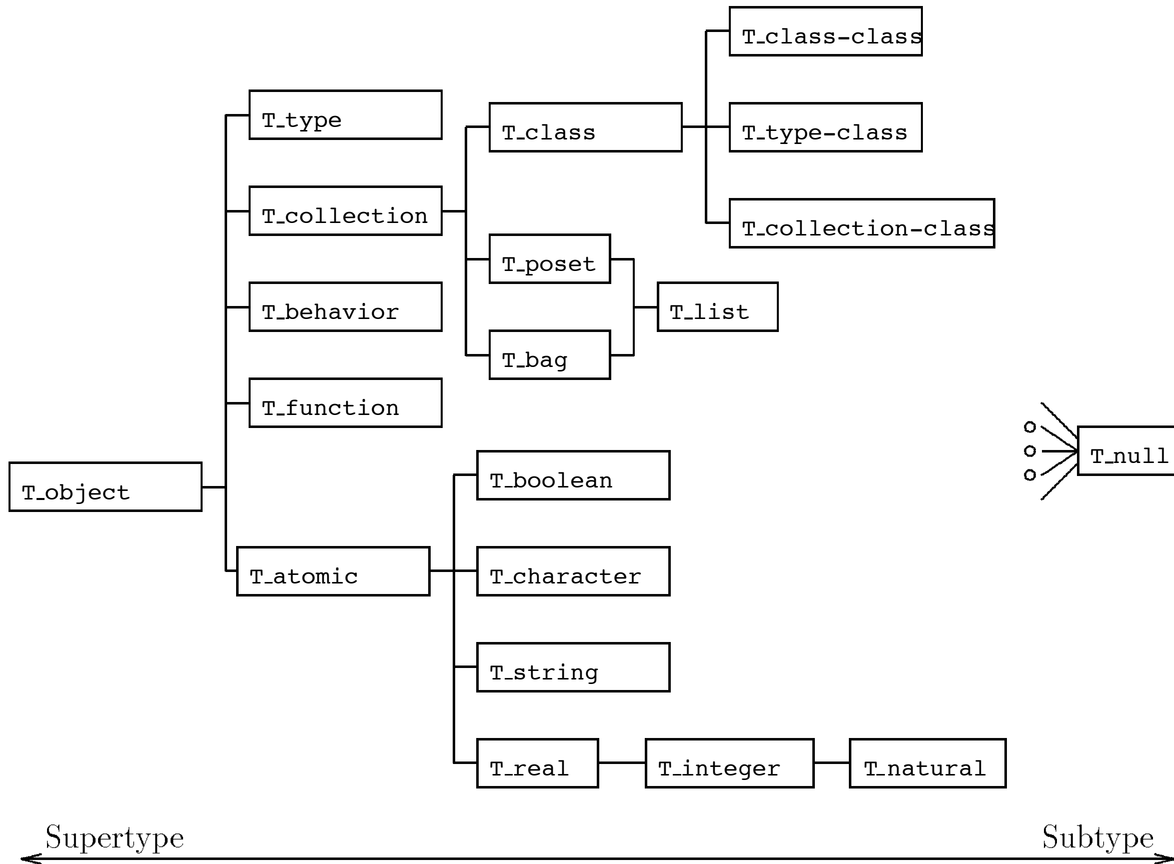

Figure 1: Primitive type system of TIGUKAT.

3 Temporality in the Object Model

3.1 Object Model Extensions

Applications that require the functionality of object management systems also require an extensible type system. Applications built on top of TIGUKAT may have different type semantics. Consequently, we provide a rich and extensible set of types to support various models of time. The inclusion of the time abstract types into the primitive type lattice is shown in Figure 2.

There are two aspects of modeling time: *structural models* of time and the *density* of these structural models. Two basic structural models of time can be identified [Ben83]:

- *Linear*: In the linear model, time flows from the past to the future in a totally ordered manner.
- *Branching*: In the branching model, time is linear in the past upto the present time (*now*), at which point it branches out into the future. In other words, the two predecessors of a given time must be relatable. The structure of the branching model can be thought of as a tree with *now* as its root defining a partial order of times. This model is useful in applications where alternate evolutions of versions and their variants are to be kept.

The density in a structural model of time defines the domain over which time is perceived in the model. In other words, it defines a scale for time in the model. We identify three *scales* (*domains*) of time:

1. *Discrete* domains map time to the set of integers (or to the set of natural numbers when combined with the linear model).
2. *Dense* domains map time to the set of rational numbers.
3. *Continuous* domains map time to the set of real numbers.

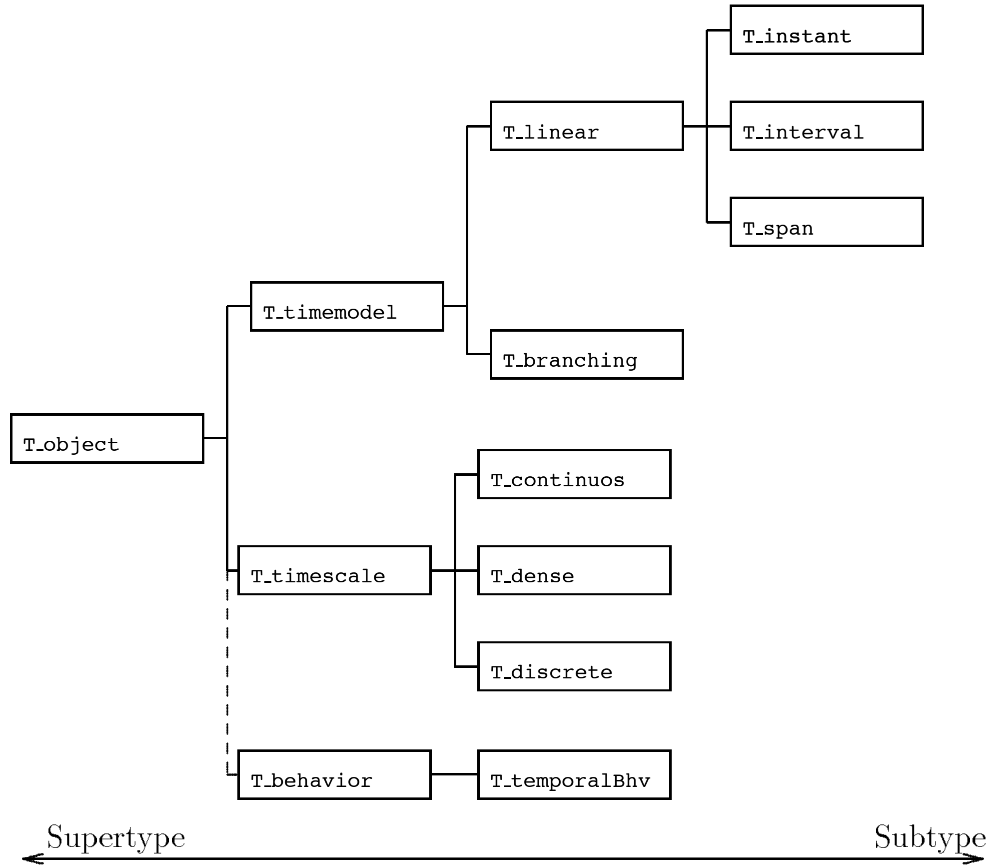

Figure 2: Primitive type system with abstract time types.

To model the structural models, we define type `T_timemodel` as a subtype of `T_object`. Similarly, to characterize the density of the structural models, we introduce the `T_timescale` type as a subtype of `T_object`.

Type `T_timemodel` has subtypes `T_linear` and `T_branching` which define the linear and branching structural models of time respectively. Each of these has the *B_timescale* behavior defined on them (see Table 1). In `T_timemodel`, this behavior returns a collection of `T_timescale` objects, thereby defining the *density* of the time model. The *B_timescale* behavior is refined in `T_linear` to return a list of `T_timescale` objects due to the total ordering on the linear model, and a collection of `T_linear` objects in `T_branching` since a branching model can be visualized as a union of linear branches.

The `T_timescale` type has the normal comparison behaviors such as *B_lessthan*, *B_greaterthan*, etc (see Table 1 for full behavior definition). Furthermore, it has subtypes, `T_discrete`, `T_dense` and `T_continuous` which define the respective domains of time. Type `T_discrete` has additional behaviors *B_next* and *B_previous* which return the next and previous time points of a particular time point. Type `T_dense` has the behavior *B_incr* which essentially finds a rational time between two rational times. Finally, type `T_continuous` has behaviors to *B_round* or *B_truncate* a continuous time to a discrete one.

Objects can be timestamped with either a time *instant* (*moment*, *chronon*, etc.), a time *interval* or a time *span* (*duration*). A time instant is a specific instant on the time axis and can be compared with another time instant with transitive comparison operators. A time interval can be defined as an ordered pair of time instants, a lower bound and an upper bound where the former is less than the latter. A time span is an unanchored, relative duration of time (e.g., an event taking 5 months). It is independent of any time instant or interval.

The `T_linear` type can be subtyped into three abstract types, `T_instant`, `T_interval` and `T_span` which basically identify the timestamps for objects[2].

[2]The `T_branching` type could be subtyped as well to model versions and their variants. Conceptually, these would be a collection of their linear counterparts. In this work however, we concentrate on the `T_linear` type.

Type	Signatures	
T_timescale	*B_lessthan*:	T_timescale → T_boolean
	B_lessthaneqto:	T_timescale → T_boolean
	B_greaterthan:	T_timescale → T_boolean
	B_greaterthaneqto:	T_timescale → T_boolean
	B_noteqto:	T_timescale → T_boolean
T_continuos	*B_round*:	T_discrete
	B_truncate:	T_discrete
T_dense	*B_incr*:	T_dense → T_dense
T_discrete	*B_next*:	T_discrete
	B_previous:	T_discrete
T_timemodel	*B_timescale*:	T_collection<T_timescale>
T_linear	*B_timescale*:	T_list<T_timescale>
T_branching	*B_timescale*:	T_collection<T_linear>
T_instant	*B_timescale*:	T_timescale
	B_precedes:	T_interval → T_boolean
	B_follows:	T_interval → T_boolean
	B_within:	T_interval → T_boolean
	B_add:	T_span → T_instant
	B_sub:	T_span → T_instant
T_interval	*B_timescale*:	T_list<T_timescale>
	B_lb:	T_timescale
	B_ub:	T_timescale
	B_length:	T_span
	B_precedes:	T_interval → T_boolean
	B_meets:	T_interval → T_boolean
	B_overlaps:	T_interval → T_boolean
	B_during:	T_interval → T_boolean
	B_starts:	T_interval → T_boolean
	B_finishes:	T_interval → T_boolean
	B_union:	T_interval → T_interval
	B_intersection:	T_interval → T_interval
	B_difference:	T_interval → T_interval
T_span	*B_timescale*:	T_timescale
	B_add:	T_span → T_span
	B_sub:	T_span → T_span
	B_mult:	T_integer → T_span
	B_div:	T_integer → T_span
	B_mod:	T_integer → T_span

Table 1: Behavior signatures of the time abstract types.

Behaviors are defined on `T_instant` to check if a time instant is before, after, or within a time interval. Furthermore, behaviors *B_add* and *B_subtract* are provided which connect spans with instants.

A rich set of behaviors is defined on `T_interval` which includes interval comparison behaviors [All84] (*B_overlaps*, *B_meets*, *B_during*, etc) and set-theoretic behaviors (*B_union*, *B_difference*, and *B_intersection*).

Behaviors on `T_span` allow time spans to be added to (subtracted from) each other to give other time spans. A time span can also be multiplied or divided by an integer to give another time span. To model absolute times, like *dates*, we can easily extend our time type hierarchy by defining a subtype, `T_date` of the `T_instant` type. Furthermore, we can subtype type `T_span` to model year, month and day spans (durations). These can be further subtyped to model finer granularities of time.

To manage temporal information of various properties of objects, we introduce a subtype, `T_temporalBhv`, of the `T_behavior` type. `T_temporalBhv` has an additional functionality in that its instances maintain a history of updates with respect to a particular object to which they are applicable. We model this history of updates by defining the *B_history* behavior in the interface of type `T_temporalBhv` which returns a collection of <`T_timemodel`, `T_object`> objects. More specifically,

$$B_history: \texttt{T_object} \rightarrow \texttt{T_collection}\langle\texttt{T_timemodel}, \texttt{T_object}\rangle$$

For example, if $e \in$ **C_employee**, and *B_salary* is defined in the interface of `T_employee` and is an instance of **C_temporalBhv**, then $B_salary.B_history(e)$ gives the salary history of employee e.

The following definitions formally distinguishes between temporal and non-temporal objects.

Definition 3.1 *Object temporality:* An object o is temporal iff $t = o.B_mapsto$ is temporal, where t is a type object.

Definition 3.2 *Type temporality:* Type t is temporal iff

$$\exists b | b \in t.B_interface \wedge$$
$$b \in \mathbf{C_temporalBhv}$$

T_employee	:	*B_ssno* : T_integer
		B_name : T_string
		BT_Dept : T_department
		BT_Salary : T_integer
T_department	:	*B_name* : T_string
		BT_Manager : T_employee

Figure 3: Employee object type

Since either all objects belonging to a particular type are temporal or all are non-temporal temporality of an object is determined by looking at its type. As shown in the above expression, a type is temporal if and only if there exists at least one behavior in its interface which is an instance of the **C_temporalBhv** class. Hence, the above rule uniformly determines whether any object is temporal or non-temporal.

For discussion purposes, in the rest of this paper we concentrate on the type T_interval to demonstrate the extensibility of our modeling of time. Extensions for other temporal notions can be provided in the same manner, but in the interest of saving space, we don't discuss them in this paper. To exemplify the notion of timestamping objects, we introduce a subtype of the T_interval type, T_DiscInterval[3], which refines all the behaviors of its supertype by fixing the time scale to be discrete. We also use the term, *interval* to mean an instance of **C_DiscInterval**. Hence, the time model part of the <T_timemodel, T_object> object is taken to be the *interval* in which the object is valid. Consequently, we represent the history of behaviors which are instances of **C_temporalBhv** by sets of triplets of the form, $< [l, u), o >$ where [l,u) is the time component (interval), with *l* and *u* representing the *lower* and *upper* bounds of the interval respectively, and *o* is the object (could be either atomic or complex) which is valid (exists) over the time interval [l,u).

The time instant *now* is the marking symbol for the current time. An interval whose upper bound is *now* expands as the clock ticks. We do not specify any time unit; this is left to the user. For these interval comparison behaviors, we assume the existence of an instance of **C_DiscInterval**, [ ], which stands for the empty interval.

The result of the *B_history* behavior is a collection of <T_DiscInterval, T_object> objects and can be represented as $\texttt{T_collection}\langle\texttt{T_DiscInterval}, \texttt{T_object}\rangle$. In other words, T_collection is made up of objects whose type is $\texttt{T_DiscInterval} \times \texttt{T_object}$. This type is automatically created as a subtype of T_product [PLÖS93b] and thereby inherits all its native behaviors. The injection behavior (ρ_i) of T_product returns the ith component of a product object. Hence, if o is a temporal product object (i.e., $o \in \texttt{T_collection}\langle\texttt{T_DiscInterval}, \texttt{T_object}\rangle$), then $o.\rho_1$ returns an object of type T_DiscInterval and $o.\rho_2$ returns an object of type T_object.

For *notational* convenience, in the rest of the paper we prefix the names of behaviors which are instances of the **C_temporalBhv** class by BT_. As an example, a T_employee object type with its respective behaviors can be represented as shown in Figure 3. If $e \in \texttt{T_employee}$, and $s \in BT_Salary.B_history(e)$, then $s.\rho_2$ gives the salary value during the interval given by $s.\rho_1$.

A point worthy of mention is the *temporality transparency* in the signatures of behaviors which are instances of the **C_temporalBhv** class. This is important especially from user perspective since behavior histories can be got simply by application of the *B_history* behavior thereby providing uniformity in the signatures of behaviors which are instances of the **C_temporalBhv** class and those that are instances of the **C_behavior** type.

Two basic aspects of time are considered in databases which incorporate time. These are the *valid* and *transaction* times. The former denotes the time when an object becomes effective (begins to model reality), while the latter represents the time when a transaction was posted to the database. The need to distinguish between valid and transaction time arises when an update to an object is posted to the database at a time which is different than the time when the update becomes valid. In this work we only consider valid time (all the notions introduced apply to transaction time as well and can easily be carried forward) and use the term "temporal objectbase" liberally to refer to a database which contains any kind of time-varying objects.

Objects in our model are either mutable (versionable) or immutable (non-versionable). Immutable objects exist indefinitely in the database. Examples of these are T_integer, T_string, etc. Mutable objects on the other hand, are simply compositions of immutable objects tagged by a sequence of continuous time intervals in ascending order. T_employee is an example of a mutable object. This categorization enables us to model both time-varying and non time-varying objects.

[3] The T_interval type can be subtyped to model any other specialized intervals according to a specific time scale.

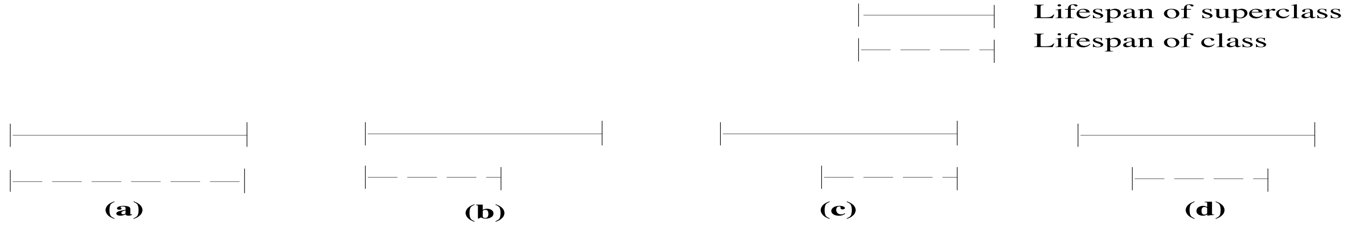

Figure 4: Temporal constructs between a class object and its superclass object

3.2 Temporal Semantics of Inheritance

We model temporal constraints to represent relationships between objects, more specifically between objects in a class and those existing in their (immediate) superclass. Figure 4 shows the four different constructs used to model temporal relationships between a class and its superclass. As seen, the *lifespan* of a class is contained in the *lifespan* of its superclass where the lifespan is the time interval over which objects of this class are valid[4]. The differentiating factor in the four cases is the start and end times of the class as compared to those of its superclass. The construct in Figure 4 (a) is directly inherited by the subclass(es), and need not be explicitly specified. We do not allow the lifespan of a class object to be larger than that of its superclass object. For example, if a person dies, it makes no sense for his existence to continue as an employee, assuming **C_employee** is a subclass of **C_person**.

Adding time to the object model should enable us to find out all existing objects of a class (collection) at a particular time interval. We introduce a *timeextent* behavior in the `T_collection` type which returns a collection of objects existing at a given time interval, when applied to a collection (class). If no interval is specified, defaults could be set by the database administrator to either return the collection of currently existing objects or the collection of objects which ever existed.

$$\textit{B_timeextent} : \texttt{T_DiscInterval} \rightarrow \texttt{T_collection}\langle\texttt{T_object}\rangle$$

From the constructs given in Figure 4, it follows that the collection of existing class (collection) objects during a given time interval (extent of the class), is a subset of its existing superclass objects during the same time interval (extent of the superclass). This is essentially same as the *temporal inclusion* constraint specified in [WD92]. The constructs in Figure 4 also restrict the behaviors inherited by a class from its superclass to span over the duration in which the class exists and not the duration in which its superclass existed.

In [WD92], a *lifespan* function is defined which takes an object as an argument and returns the time during which it existed. The lifespan of an object in the class to which it belongs is restricted to be a subset of its lifetime in the superclass, but it is not clear how the constraint is actually enforced. We extend this notion of the *lifespan* function and formally define a more general lifespan behavior, *B_lifespan* on the `T_object` type which returns the time during which an object existed in a *particular* collection (class). This definition allows us to talk about the lifespan of an object not only in a class, but in any collection as well. Collections are typed groupings of objects meaning every collection has an associated type. For a collection c, we use *B_typeof(c)* to denote the associated type for collection c.

$$\textit{B_lifespan}\text{: } \texttt{T_collection} \rightarrow \texttt{T_DiscInterval}$$

To ensure temporal consistency, we add the following rule:

$$B_lifespan(o, x) \leq B_lifespan(o, c) \leq B_lifespan(c, C_class)$$

where c is a class object, x is any collection object and *B_typeof(x)* is a subtype of *B_typeof(c)*. The above rule ensures that the lifespan of an object within a particular class c is contained in the lifespan of class c. Furthermore, the lifespan of that object in any collection x ranging over the class c, is also contained in the lifespan of c. When an object ceases to exist (becomes invalid), the extent of the class to which it belongs (and the extents of all its subclasses) is adjusted to reflect this change. A similar change is made in the respective collections ranging over the class.

The temporal inheritance semantics using a lifespan behavior hold true for multiple subtyping as well, but there are questions concerning update semantics in the primitive model which need to be addressed first. We are currently working on the update semantics for multiple subtyping in the object model and this will be directly carried forward to the temporal object model.

[4] We base our discussion of inheritance semantics on classes rather than on types since in TIGUKAT types may exist without having an extent (abstract types) modeled as classes. The lifespans of instance objects are, therefore, limited to the lifespans of their classes rather than their types.

4 The TIGUKAT Query Language

The TIGUKAT query model is a direct extension to the object model. It is defined by type and behavioral extensions to the primitive model. The languages for the query model include a complete calculus, and equivalent object algebra and a SQL-like user language.

The calculus has a logical foundation and its expressive power is outlined by the following characteristics. It defines predicates on collections (essentially sets) of objects and returns collections of objects as results which gives the language *closure*. It incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavioral applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has rigorous definitions of safety (based on the evaluable class of queries) and typing which are compile time checkable. It supports controlled creation and integration of new collections, types and objects into the existing schema.

Like the calculus, the algebra is *closed* on collections. Algebraic operators are modeled as behaviors on the primitive type `T_collection`. They operate on collections and return collections as a result. Thus, the algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. The combination of these behaviors brings closure to the algebra.

Details of the equivalence of the object calculus and algebra in both directions, reduction of the user language to the calculus and the safety of the calculus, algebra and user languages are given in [PLÖS93a]. In this section, we briefly discuss the TIGUKAT Query Language (TQL) and demonstrate how it can be used to access temporal objects since the rich set of behaviors defined on the time abstract types alleviates the need to make changes to the underlying calculus and algebra.

TQL is based on the SQL paradigm and its semantics is defined in terms of the object calculus. Hence, every statement of the language corresponds to an equivalent object calculus expression.

The basic query statement of TQL is the *select statement*. It operates on a set of input collections, and it always returns a new collection as the result. The general syntax of the select statement is as follows:

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

The *select clause* in this statement identifies objects which are 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 either an existing collection, or a collection returned as a result of a subquery, where a subquery can be either given explicitly, or as a reference to a query object. The range variable in the *from clause* is as follows:

$< range\ variable >$: $< identifier\ list >$ **in** $< collection\ reference >$ [**+**]
$< collection\ reference >$: $< term >$ | **(** $< query\ statement >$ **)**

The collection reference in the range variable definition can be followed by a '**+**' to refer to the shallow extent of a class. The *term* in the collection reference definition is either a constant reference, a variable reference, or a path expression.

The *where clause* defines a boolean formula which must be satisfied by objects returned by a query. Two additional predicates are added to TQL boolean formulas to represent existential and universal quantification. The existential quantifier is expressed by the *exists predicate* which is of the form:

$$< exists\ predicate >: \ \mathbf{exists}\ < collection\ reference >$$

The *exists predicate* is *true* if the referenced collection is not empty. The universal quantifier is expressed by the *forAll predicate* which has the structure:
$< forAll\ predicate >$: **forAll** $< rangevariablelist > < boolean\ formula >$
The syntax of the *range variable list* is the same as that in the *from* clause of the select statement. It defines variables which range over specified collections. The *boolean formula* is evaluated for every possible binding of the variables in this list. Hence, the entire *forAll predicate* is *true*, if for every element in every collection in the range variable list, the boolean formula is satisfied.

Having described TQL, and with the behaviors defined on `T_DiscInterval`, we show in the next section how temporal objects can uniformly be queried without changing any of the basic constructs of TQL.

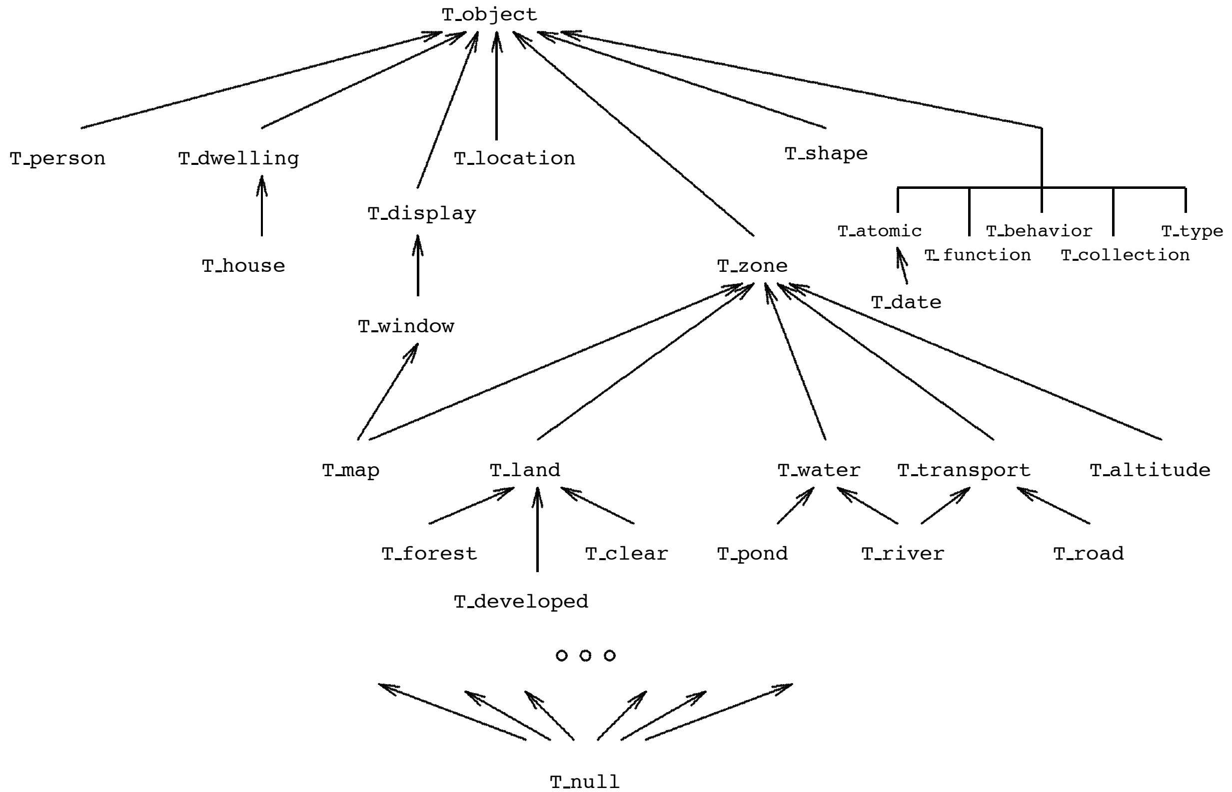

Figure 5: Type lattice for a simple geographic information system.

5 Example Objectbase

In this section, we present some examples on a geographic information system (GIS) objectbase to demonstrate how the temporal extensions to our underlying object model can easily be incorporated within the query model, without extending TQL with any extra construct to query temporal objects.

A type lattice for a simplified GIS is shown in Figure 5. The example includes the root types of the various sub-lattices from the primitive type system to illustrate their relative position in an extended application lattice. The GIS example defines abstract types for representing information on people and their dwellings. These include the types `T_person`, `T_dwelling` and `T_house`. Geographic types to store information about the locations of dwellings and their surrounding areas are defined. These include the type `T_location`, the type `T_zone` along with its various subtypes which categorize the various zones of a geographic area, and the type `T_map` which defines a collection of zones suitable for displaying in a window. Displayable types for presenting information on a graphical device are defined. These include the types `T_display` and `T_window` which are application dependent, along with the type `T_map` which is the only GIS application specific object that can be displayed. Finally, the type `T_shape` defines the geometric shape of the regions representing the various zones. Table 2 lists the signatures of the behaviors defined on GIS specific types. A type in the GIS example is temporal, if it has at least one behavior which is an instance of **C_temporalBhv**. Hence, as seen in Table 2, the types `T_zone`, `T_map`, `T_land`, `T_water`, `T_transport`, `T_person` and `T_house` are temporal.

The following examples illustrate possible queries on the GIS expressed in TQL. We use arithmetic notation for the operators like *o.greaterthan(p)*, *o.elementof()*, etc., and set inclusion notation for the operators like *overlap(o,p)*, *union(o,p)*, etc., instead of boolean path expressions. Lower case letters like o, p, a, etc., stand for object variables.

Example 5.1 Return land zones valued over $100,000 that covered an area over 1000 units.

select o
from o **in in C_land**, v **in** *BT_Value.B_history*(o), a **in** *BT_Area.B_history*(o)
where $(v.\rho_2() > 100000$ **and** $a.\rho_2() > 1000$ **and** $(v.\rho_1() \cap a.\rho_1()) \neq [\,])$

Type	Signatures	
T_location	*B_latitude*:	T_real
	B_longitude:	T_real
T_display	*B_display*:	T_display
T_window	*B_resize*:	T_window
	B_drag:	T_window
T_shape		
T_zone	*B_title*:	T_string
	BT_Origin:	T_location
	BT_Region:	T_shape
	BT_Area:	T_real
	BT_Proximity:	T_zone $\rightarrow$ T_real
T_map	*B_resolution*:	T_real
	B_orientation:	T_real
	BT_Zones:	T_collection$\langle$T_zone$\rangle$
T_land	*BT_Value*:	T_real
T_water	*BT_Volume*:	T_real
T_transport	*BT_Efficiency*:	T_real
T_altitude	*B_low*:	T_integer
	B_high:	T_integer
T_person	*B_name*:	T_string
	B_birthDate:	T_date
	BT_Age:	T_natural
	BT_Residence:	T_dwelling
	BT_Spouse:	T_person
	BT_Children:	T_person $\rightarrow$ T_collection$\langle$T_person$\rangle$
T_dwelling	*B_address*:	T_string
	B_inZone:	T_land
T_house	*B_inZone*:	T_developed
	BT_Mortgage:	T_real

Table 2: Behavior signatures pertaining to the GIS example.

Example 5.2 Return all zones which have people currently living in them.

select o
from p **in C_person**, r **in** *BT_Residence*.*B_history*(p)
where (o=p.*BT_Residence()*.*B_inzone()* **and** $r.\rho_1()$.*B_ub*$=now$)

Example 5.3 Return the maps with areas where senior citizens have ever lived.

select o
from o **in C_map**, z **in** *BT_Zones*.*B_history*(o)
where exists (**select** p
from p **in C_person**, r **in** *BT_Residence*.*B_history*(p), d **in C_dwelling**
where (p.*BT_Age*$() \geq 65$ **and** $d = r.\rho_2()$ **and** d.*B_inzone()* $\in z.\rho_2()$))

Example 5.4 Return all persons who changed their spouse in a span of less than 2 years. (assuming a time unit is equal to a year)

select o
from o **in C_person**
where forAll p **in** (*BT_Spouse*.*B_history*(o))
($p.\rho_1()$.*B_length* < 2)

Example 5.5 When was person x spouse of person y?

select $s.\rho_1()$
from y **in C_person**, x **in C_person**, s **in** *BT_Spouse*.*B_history*(y)
where $s.\rho_2() = x$

Example 5.6 Return all (`T_person`,`T_person`,`T_Set`) triples of people who have never married but have children together.

select $p,\ q,\ k$
from p **in C_person**, q **in C_person**
where $(k = p.\mathit{BT_Children}(q)$ **and** $\neg(k = \{\})$ **and**
(**forAll** s **in** $(\mathit{BT_Spouse}.\mathit{B_history}(p))$
$\neg(s.\rho_2() = q)))$

6 Conclusion

In this paper we defined temporal extensions to the TIGUKAT object model by providing an extensible set of primitive time types with a rich set of behaviors to model various notions of time elegantly. We showed how temporal objects can be timestamped with one of the time interval types and manipulated using the injection behavior of the `T_product` type.

We introduced a general lifespan notion for objects which models the lifespan of an object in any collection, and the class to which it belongs. A lifespan constraint was defined for objects which facilitates the inheritance of temporal classes.

Finally, we gave a real world GIS example and illustrated with the help of some example queries, how temporal objects can be queried without adding any extra construct to our query language, TQL.

We are currently investigating the modeling of schema versioning and other time dimensions (like transaction time) with our primitive time types in a uniform way. We also intend to look into the issues of query optimization when temporal objects are considered by investigating how best the notion of a time index [EWK90] can be applied to complex temporal objects.

References

[AG89] R. Agrawal and N.H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 36–45, May 1989.

[All84] J. F. Allen. Towards a General Theory of Action and Time. *Artifical Intelligence*, 23(123), 1984.

[BCG+87] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Ben83] J.F.K.A. Benthem. *The Logic of Time*. Reidel, 1983.

[CDV88] M. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 413–423, September 1988.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 80–102, June 1989.

[DW92] U. Dayal and G. Wuu. A Uniform Approach to Processing Temporal Queries. In *Proc. 8th Int'l. Conf. on Data Engineering*, pages 407–418, August 1992.

[EW90] R. Elmasri and G. Wuu. A Temporal Model and Query Language for ER Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 76–83, February 1990.

[EWK90] R. Elmasri, G. Wuu, and Y. Kim. The Time Index: An Access Structure for Temporal Data. In *Proc. 16th Int'l Conf. on Very Large Data bases*, August 1990.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Gad88] S. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4), 1988.

[Gor92] I. Goralwalla. An Implementation of a Temporal Relational Database Management System. Master's thesis, Bilkent University, 1992.

[GR85] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison-Wesley, 1985.

[KC86] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 406–416, September 1986.

[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, 1992.

[LJ88] N. Lorentzos and R. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 15(3), 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstien, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 424–433, September 1988.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming*, pages 355–392. M.I.T. Press, 1987.

[PLÖS93a] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. A Query Model and Language for the TIGUKAT Objectbase Management System. Technical Report TR93-01, Department of Computing Science, University of Alberta, January 1993.

[PLÖS93b] 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*, November 1993.

[PÖS92] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR-92-14, University of Alberta, October 1992.

[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.

[RS93] E. Rose and A. Segev. TOOA - A Temporal Object-Oriented Algebra. In *Proc. European Conference on Object-Oriented Programming*, 1993.

[Shi81] D.W. Shipman. The Functional Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), March 1981.

[SK86] A. Shoshani and K. Kawagoe. Temporal Data Management. In *Proc. 12th Int'l Conf. on Very Large Data Bases*, 1986.

[Sno87] R. Snodgrass. The Temporal Query Language, TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[SS87] A. Segev and A. Shoshani. Modeling Temporal Semantics. In *Temporal Aspects of Information Systems Conference*, 1987.

[SZ90] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

[Tan86] A. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, 13(4):343–355, 1986.

[TG89] A. Tansel and L. Garnett. Nested Historical Relations. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 1989.

[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, February 1992.