# Temporal Object Role Modelling*

Andreas Steiner and Moira C. Norrie

Institute for Information Systems
ETH Zürich, CH-8092 Zürich, Switzerland

**Abstract.** We present a temporal object model capable of representing object lifespans and also the history of their roles and associations. We adopt an approach of *temporal generalisation* rather than *temporal extension* in which a model in its entirety is given a temporal semantics through an orthogonal generalisation of all model concepts – including the lifespan of object roles themselves and constraints over these roles. The model is based on the generic object data model OM and its algebra has been generalised into a full temporal algebra over object roles.

## 1 Introduction

Full support for temporal databases requires a complete generalisation of a data model with a temporal dimension. This means that all aspects of the model – constructs, operations and constraints – must have temporal generalisations. Further, the temporal dimension should apply not only to data, but also to metadata.

Many existing proposals for temporal data models and systems – both relational and object-oriented – tend to focus on one particular aspect of a model and extend it with temporal properties. For example, many proposals deal with extensions to data structures to enable entities or entity properties to be timestamped – whether it be with valid or transaction times (e. g. [Wuu91, RS93, GÖ93, BFG96] or in [TCG+93]). Typically, the query language is then extended with temporal properties in the sense of selections with temporal predicates or temporal joins (e. g. [Wuu91, GÖ93]). However, in many cases, the underlying algebra is not fully generalised in the sense of providing temporal semantics to all operations of the algebra. Temporal negation for example is frequently neglected. Proposals for models and systems with temporal semantics for all operations (e. g. [Sno95, SBJS96]) and for constraints (e. g. [WD93, SBJS96]) do exist, but tend to be considered separately from each other. The issue of timestamped metadata has received little consideration.

We advocate an approach of *temporal generalisation* rather than *temporal extension* in which a model in its entirety is given a temporal semantics through an orthogonal generalisation of all concepts of the model. Previously, we considered this approach to a limited extent in the context of relational database systems and developed a prototype temporal database system TimeDB [Ste95] which

* In *Proceedings of the 9th International Conference on Advanced Information Systems Engineering (CAiSE '97), Barcelona, Spain*

supports both valid and transaction times, has a full temporal query language and, in contrast to many other implemented systems, also supports temporal updates, integrity constraints and views. The prototype system TimeDB is based on the query language ATSQL2 [SBJS96].

More recently, we have fully exploited the generalisation approach in the context of object-oriented systems and developed a temporal object model and system, TOM, capable of modelling object role and association histories. TOM also supports a full temporal algebra, query language and constraints. Additionally, metadata can have temporal properties allowing the modelling of role, association and constraint lifespans.

Our temporal object data model is based on the generic object-oriented data model, OM [Nor93]. The OM model strictly separates typing from classification in such a way that classification structures model the roles of objects rather than their representation. The model is therefore independent of any particular type system and programming language environment. Other key features of the OM model that impact on the temporal model are its collection algebra which defines generic operations over collections, the model's support for object and relationship evolution [NSWW96] and the orthogonality with which the constructs of the model may be applied.

In section 2, we present the main features of the OM model in terms of a simple example application used throughout the paper. Section 3 then presents the basic constructs of the temporal object model TOM in terms of temporal objects, roles and associations. Section 4 discusses the temporal generalisation of classification and association constraints. The temporal algebra and query language is presented in section 5. Concluding remarks are given in section 6.

## 2 Object Data Model OM

In OM, object roles are semantic groupings of objects represented by collections. The properties of an object in terms of attributes and methods are specified by the underlying type system. An object may be in many collections simultaneously and a collection may contain objects of different types as long as they have some minimal set of properties as specified by an associated member type. We thus distinguish between issues of representation and semantics by separating the notions of typing from those of role modelling.

Associations model relationships between objects of certain roles and are represented by a special form of collection – binary collection – in which each element is a pair identifying the related objects. Since binary collections are a specialisation of collections, all operations and constraints which apply over object roles also apply over associations. An association can be regarded as modelling relationship roles and therefore, generally, collections represent roles – be they object or relationship roles.

Collections are grouped into classification structures each of which describes related roles in terms of a specialisation graph. We illustrate this by means of the example schema for a property leasing company shown in figure 1.

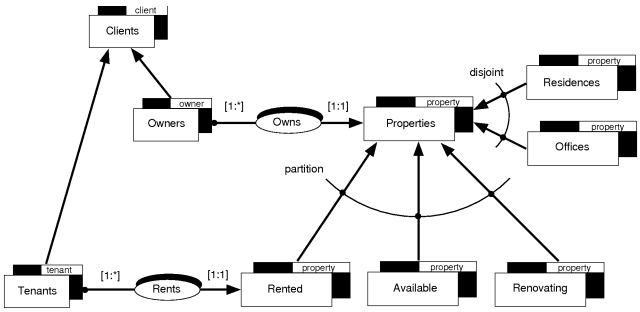


**Fig. 1.** An Example OM Schema Diagram

Figure 1 includes four classification structures. The classification structure on the right represents properties and consists of the collection `Properties` and its subcollections `Residences`, `Offices`, `Rented`, `Available` and `Renovating`. Shaded boxes are used to denote collections with the name of the collection in the unshaded region and the type of the member values specified in the shaded region. Subcollections `Residences` and `Offices` are constrained to be *disjoint* meaning that, at any point in time, no property can be categorised for use as both a residence and an office. Subcollections `Rented`, `Available` and `Renovating` form a *partition* in that they are pairwise *disjoint* and form a *cover* of `Properties` in that, at any point in time, every property must have exactly one of these three roles.

A second classification structure represents clients and their roles. `Clients` has subcollections `Owners` and `Tenants`. Note that the member type `tenant` of `Tenants` is a subtype of the member type `client` of `Clients`. Likewise, `owner` is also a subtype of `client`. It is possible that a `client` object belongs to both `Owners` and `Tenants`.

The third and fourth classification structures consist of the single associations `Owns` and `Rents`, respectively. Associations are represented by oval-shaped boxes, with links to the related collections and their respective cardinality constraints. `Owns` would be a collection of pairs of object values such that the first elements of the pairs belong to `Owners` and the second to `Properties`. We refer to `Owners` as the *source collection* and to `Properties` as the *target collection*.

OM supports object evolution in that objects may change their roles over the course of time. Such forms of evolution require changes in collection membership and this in turn may involve changes in the type of an object. For example, if a `tenant` object becomes an `owner` object, then the object must gain additional owner attributes. OM supports changes in object types through *dress* and *strip* operations. Further, the model includes mechanisms to control object evolution. For example, objects can only migrate within a classification structure, thereby preventing absurd evolutions such as an object in `Tenants` becoming an object in `Properties`. The issue of object (and relationship) evolution and further forms of control over migration are discussed in detail in [NSWW96].

The operational model of OM is based on an algebra of collections. Descriptions of some of these operations are given in section 5 where the temporal equivalents are described. Further details of the OM model and its algebra are given in [Nor93].

We have developed our own object-oriented database management system, OMS, based on the OM model. An interesting aspect of the OMS system with respect to temporal generalisation is its representation of *all* information – both data and metadata – as objects. Our approach of generalising objects to temporal objects, therefore leads immediately to the possibility of timestamping all information within the system.

Note that, in this paper, we deal only with set collections, but the concepts generalise to other forms of collections of OM, for example bag collections.

## 3 Temporal Object Role Model

Our temporal model TOM is based on *object-timestamping*. We add timestamps to the *names* of instances. In other words, we do not extend the *types* with timestamp attributes but rather extend the *object identifiers* with a lifespan. In this paper, we focus on the valid-time aspects of our model and the lifespan expresses when an object was *valid* (existent) in the real world.

Since object roles are represented by collections which are themselves objects, collections may also be timestamped. As a result, we can model the fact that roles also exist for limited lifespans and, further, that they may appear and disappear with respect to the current state of an application domain.

Adding timestamps to objects leads naturally to a more general model than the usual relational temporal models in that, not only entities and their roles, but also the roles themselves can have temporal properties. By timestamping objects (and object-pairs in binary collections), a direct comparison can be made between lifespans of objects, relationships, object roles and associations. We now go on to consider these various aspects of our model in more detail.

### 3.1 Temporal Object Identifiers

Our notion of a lifespan is similar to that proposed in [CC87]. The smallest non-decomposable time unit assumed in a temporal database, for example a *second*, is called a *chronon* [TCG+93]. Let $\mathcal{T} = \{t_0, t_1, \ldots\}$ be a set of chronons, at most countably infinite. The linear order $<_{\mathcal{T}}$ is defined over this set, where $t_i <_{\mathcal{T}} t_j$ means that $t_i$ occurs before $t_j$.

A lifespan `ls` is any subset of the set $\mathcal{T}$. [GV85] called this sort of timestamp *temporal elements*. We assume that $\mathcal{T}$ is isomorphic to the natural numbers. Thus we can represent a lifespan also as a set of non-overlapping intervals, closed at the lower bound and open at the upper bound. Lifespans are closed under the usual set-theoretic operations *union*, *intersect* and *difference*. If $ls_1$ and $ls_2$ are lifespans, then $ls_1 \cap ls_2$, $ls_1 \cup ls_2$ and $ls_1 - ls_2$ are also lifespans.

This definition of *lifespan* reflects the fact that an object may appear and disappear several times during its overall time of existence. A lifespan contains all those time points at which an object existed. For example, the timestamp of a `property` object may represent the various periods during which that property was managed by the leasing company.

**Definition 1. (temporal object identifier)** Let $O$ be the set of all possible non-temporal object identifiers. A temporal object identifier `toid` consists of an object identifier $oid \in O$ and a lifespan $ls$, $\texttt{toid} := \ll oid; ls \gg$.

In the following, we use the notation $lifespan(\texttt{toid})$ to reference the lifespan contained in the temporal object identifier `toid`. $O^v$ shall denote the set of all *temporal object identifiers*, whereas $O$ is the set of non-temporal object identifiers. Value $\omega$ represents the *undefined* object identifier.

**Definition 2. (snapshot of a temporal object identifier)** Let $\texttt{toid} \in O^v$ be a temporal object identifier containing $\texttt{oid} \in O$ as object identifier. Let $t \in \mathcal{T}$ be a time instant. Then the *snapshot of a temporal object identifier* at a time instant $t$, $\tau^t(\texttt{toid})$, is defined as

$$\tau^t(\texttt{toid}) := \texttt{IF } t \in lifespan(\texttt{toid}) \texttt{ THEN oid ELSE } \omega$$

The snapshot of a temporal object identifier at time instant $t$ returns the object identifier `oid` if the object exists at $t$, otherwise the special value $\omega$ is returned.

### 3.2 Valid-Time Objects

We use the term *value* to mean any form of data item that can be described by the underlying type system, e.g. a base value such as an integer or a complex value such as an object value. For simplicity of presentation, we assume here simply integers and strings as base values.

Let $V_I$ be the set of all integer values and $V_S$ the set of all string values. The values $v \in (V_I \cup V_S)$ have an implicit lifespan $[0-\infty)$. The snapshot operation $\tau^t$ evaluated on an integer or string value thus always returns the integer or string value itself. We define the set of *values* available in our temporal data model as

$$V^v := V_I \cup V_S \cup O^v .$$

In this paper, we focus only on temporal values although we have both non-temporal and temporal values in our system, along with conversion operations.

With definition 2, we can express the snapshot of values $v \in V^v$, $\tau^t(v)$. If $v$ is an integer or string value, the integer or string value itself is returned. If $v$ is a *temporal* object identifier, an *non-temporal* object identifier (or $\omega$) is returned.

*Valid-time objects* are objects having a temporal object identifier. In the following, we refer to the *temporal object identifier* of a valid-time object *obj* by $toid(obj)$, the *lifespan* of this object by $lifespan(obj)$ and the *object identifier* by $oid(obj)$.

*Example 1.* When creating a valid-time object in our system, a set of valid-time periods expressing the object's lifespan has to be provided by the user:

```
create object andreas lifespan { [1964 - inf) };
create object apart1 lifespan { [1980 - 1995) };
```

As mentioned before, these objects will be dressed with a type when added to a collection. Note that, for example, name `andreas` is just used as a reference to the corresponding object. Time instant `inf` denotes that the object is valid until further notice. Non-temporal objects are created by leaving away the lifespan specification.

### 3.3 Valid-Time Collections

In this section, we introduce *valid-time collections* as collections having a *lifespan* and containing *valid-time objects* which have their own lifespan. We then define the snapshot of the extension of a valid-time collection and use this notion for further definitions.

**Definition 3. (valid-time collection)** A *valid-time collection* $C$ consists of a temporal object identifier $\texttt{toid} \in O^v$, $toid(C) = \texttt{toid}$, and an extension $ext(C) \subseteq V^v$

We write $C = [\texttt{toid}, \texttt{ext}]$ to denote a valid-time collection. Since a valid-time collection is also a valid-time object, we can reference the *temporal object identifier of a valid-time collection* $C$ by $toid(C)$, the *object identifier* by $oid(C)$ and its *lifespan* by $lifespan(C)$.

*Example 2.* In order to create the collections depicted in the figure 1 as valid-time collections, we first have to define the corresponding member types:

```
create type client(name : string);
create type tenant(profession : string) subtype of client;
create type owner(bank_account : string) subtype of client;
create type property(price : integer; street : string; city : string);
```

Now we can create the main valid-time collections `Clients` and `Properties`. Assume that the property leasing company started to exist in 1980.

```
create collection Clients type client lifespan { [1980 - inf) };
create collection Properties type property lifespan { [1980 - inf) };
```

We define the snapshot of an extension $ext(C)$ of a valid-time collection $C$ at a time instant $t$ to be the set of those values in the extension of $C$, which exist at time instant $t$. Note that these snapshot values have no time information attached.

**Definition 4. (snapshot of an extension)** The *snapshot of the extension* $ext(C) \subseteq V^v$ at a time instant $t$, $\tau^t(ext(C))$, is defined as

$$\tau^t(ext(C)) := \{v | \exists v^v \in_{set} ext(C) \wedge v = \tau^t(v^v) \wedge v \neq \omega\}$$

Definition 4 will be needed to define the valid-time subcollection relationship (definition 5) and the temporal membership relation (definition 8). With definitions 2 and 4, we can also define notions of collection *identity* and *equality* at a time instant and extend these with temporal semantics.

### 3.4 Valid-Time Subcollection Relationship

Our generalisation approach makes it also necessary to redefine the subcollection relationship for valid-time collections. In this section, we introduce the valid-time subcollection relationship used in TOM. We start with its time instant definition.

**Definition 5. (subcollection relation at a time instant)** Let $C_1$ and $C_2$ be valid-time collections. $C_1$ is a *subcollection* of $C_2$ at time instant $t \in \mathcal{T}$, $C_1 \preceq^t C_2$, if and only if all of the following conditions hold:

1. $\tau^t(toid(C_1)) \neq \omega$
2. $\tau^t(toid(C_2)) \neq \omega$
3. $\tau^t(ext(C_1)) \subseteq \tau^t(ext(C_2))$

Using definition 5, we can now define the subcollection constraint for our temporal object data model. In our system, this constraint is used to trigger actions such as update propagations to ensure that database consistency is maintained [NSWW96].

**Definition 6. (valid-time subcollection relationship)** Let $C_1$ and $C_2$ be valid-time collections. The *valid-time subcollection relationship* $C_1 \preceq^v C_2$ holds if and only if the following holds: $\forall t \in lifespan(C_1) \ : C_1 \preceq^t C_2$

Note that the valid-time subcollection relationship is defined over the lifespan of the subcollection. The valid-time subcollection relationship demands that, for each time instant subcollection $C_1$ exists, supercollection $C_2$ also has to exist. So, in our example, the lifespan of any subcollection of valid-time collection `Clients` must be contained in the lifespan $[1980 - \infty)$.

*Example 3.* We show how the subcollections `Tenants` and `Owners` depicted in figure 1 can be created. Assume that at first, the property leasing company only dealt with renting properties owned by a parent company. In 1982, the company decided to generalise their operations and also lease properties owned by others.

```
create collection Tenants
   subcollection of Clients type tenant lifespan{ [1980-inf) };
create collection Owners
   subcollection of Clients type owner lifespan{ [1982-inf) };
```

### 3.5 Adding and Removing Valid-Time Objects to Valid-Time Collections

Adding an object to a valid-time collection restricts the object's *visibility* in the collection in several ways. The object is *visible* only during a certain time period in the collection as determined by the collection's lifespan, the object's own lifespan and a membership time specified by the user. An object's *maximal visibility* in a collection is the collection's lifespan. The notion of visibility contrasts, for example, with the approach proposed in [GÖ93] where an object's lifespan has to be contained in the lifespan of the collection to which it is added.

The resulting visible lifespan `ls` of the added object is the intersection of the lifespan $\texttt{ls}_O$ of the object with the lifespan of the collection $\texttt{ls}_C$, intersected with the user specified membership time $\texttt{t}_{user}$: $\texttt{ls} := \texttt{ls}_C \cap \texttt{ls}_O \cap \texttt{t}_{user}$.

*Example 4.* We now want to add the valid-time object `andreas` of example 1 to valid-time collection `Tenants` created in example 3:

```
insert object andreas into Tenants during { [1980 - inf) };
Give a value for name: Andreas
Give a value for profession: Assistant
```

Andreas is a client of the company since 1980. He found a property to rent with the help of this company and thus is a member of collection `Tenants` in the company's database. When inserting objects into a collection, the system dresses the object with the corresponding member type (if it is not already dressed with it) and asks for attribute values (e. g. name and profession). Additionally, objects are propagated automatically to super-collections if needed.

### 3.6 Object Evolution

As stated in section 2, objects must be allowed to evolve and change roles during their lifespan. This accounts for the fact that entities in the real world change their roles during their life. For example, a tenant buys a property in another city which is then leased by the company. This client plays the role of a tenant and then gains the role of an owner. Such changes and accumulation of roles is reflected in our model by the possibility that an object can migrate from one collection to another and may also be a member of several collections at the same time.

Each collection has an associated member type. This means that, for a given collection $C$ and a given type $Type$, if $member\_type(C) = Type$, then for any value x in the extension of $C$, x must be an instance of type $Type$. Thus, to change collection membership, an object must also be able to change its type while retaining the same object identity. This is referred to as *object metamorphosis*.

Assume classification structures as depicted in figure 1. If an object in collection `Clients` is also added to subcollection `Owners`, then we first have to `dress` the object with membertype `owner` of collection `Owners`. Then a valid-time period $\texttt{t}_{user}$ has to be specified by the user which expresses the time the object

was a property owner in the real world. The visibility of this object in the valid-time collection `Owners` then results in $\texttt{ls} := \texttt{ls}_{\texttt{Owners}} \cap \texttt{ls}_{object} \cap \texttt{t}_{user}$, where $\texttt{ls}_{\texttt{Owners}}$ represents the lifespan of collection `Owners` and $\texttt{ls}_{object}$ corresponds to the lifespan of the client object to be added to collection `Owners`.

*Example 5.* Assume Andreas decided to buy a property, but he remained in the property already rented and asked the leasing company to find tenants for his property. Thus, he also became owner in the company's database.

```
insert object andreas into Owners during { [1982 - 1995) };
Give a value for bank_account: SBG 123-456
```

Andreas bought a property in 1982 and in 1995 he decided to have another company manage his property. When inserting objects, the system again dresses the objects with the corresponding member type (if needed) and asks for attribute values.

### 3.7 Temporal Associations

As described previously, relationships between objects are represented by associations. Relationships may also have valid-times associated with them and these are represented by temporal associations. A temporal association is a valid-time binary collection together with constraints specifying the source and target collections and their respective cardinality constraints as before.

**Definition 7. (valid-time binary collection)** A *valid-time binary collection* $C$ consists of a temporal object identifier $\texttt{toid} \in O^v$, $toid(C) = \texttt{toid}$, and an extension $ext(C) \subseteq (V \times V)^v$ where $V$ is the set of non-temporal values $V_I \cup V_S \cup O$.

The extension of a valid-time binary collection will be a set of object value pairs together with a lifespan. Given a valid-time binary collection $C$, then an element of $ext(C)$ may be of the form $\ll (\texttt{oid}_1, \texttt{oid}_2); \texttt{ls} \gg$ where $\texttt{oid}_1, \texttt{oid}_2 \in O$ and `ls` is the lifespan of the relationship.

*Example 6.* According to the database schema depicted in figure 1, we have to create two valid-time associations `Rents` and `Owns`. The association `Rents` exists since 1980, when the company started. Since the company decided in 1982 to extend their activity and find tenants for property owners, the association `Owns` exists since 1982. Of course, both source and target valid-time collections have to exist during the lifespan of the association. This is checked by the system. Assume that collection `Rented` has a lifespan $[1980 - \infty)$.

```
create association Rents
       source Tenants target Rented lifespan { [1980-inf) };
create association Owns
       source Owners target Properties lifespan { [1982-inf) };
```

Now we can create associations between tenants and the properties they rent, and between owners and the properties they own. Assume apartment `apart1` to be in collection `Rented` from 1980 to 1995. Tenant `andreas` has rented apartment `apart1` from 1980 to 1993:

```
insert binary object (andreas, apart1) to Rents during { [1980 - 1993) };
```

If a temporal association references an object which does not exist at all or during the specified time period, an error message is produced. This means we check for *temporal referential integrity* on both collection and object levels.

## 4 Temporal Constraints

In this section, we discuss the issue of the temporal generalisation of the classification constraints in detail. We consider the conditions imposed by the constraint with respect to a particular time instant and then generalise it over time.

We present the temporal generalisations of the disjoint constraint. Firstly, we have to redefine the membership relation for a time instant. Then, we define the *valid-time* disjoint constraint over *valid-time collections*. A description of all temporal constraints can be found in [SN97].

The *non-temporal* membership relation of a value $x$ in a set $S$ is denoted by $x \in_{set} S$. The membership relation *at a time instant* can be defined as:

**Definition 8. (membership relation at a time instant)** Let $C$ be a valid-time collection of elements of type $Type$, $member\_type(C) = Type$, and let $t \in \mathcal{T}$ be a time instant. Then for any value $x : Type$, $x \in V^v$, $x$ is a member of $C$ at time instant $t$, $x \in^t_{set} C$, if and only if both of the following conditions hold:

1. $\tau^t(toid(C)) \neq \omega$
2. $\tau^t(x) \in_{set} \tau^t(ext(C))$

Note that according to definition 4, $\omega$ is never a member of a set of values $\tau^t(ext(C))$. With definition 8, we can now define the valid-time disjoint constraint.

**Definition 9. (disjoint constraint at a time instant)** Let $t \in \mathcal{T}$ be a time instant. The *disjoint constraint at time instant* $t$ over a set of valid-time collections $CS$, $disjoint^t(CS)$, is defined as

$$\forall C_i, C_j \in CS : oid(C_i) \neq oid(C_j) \Rightarrow \neg\exists x : \; x \in^t_{set} C_i \wedge x \in^t_{set} C_j$$

Note that if at least one of the two collections $C_i$ or $C_j$ is undefined at time instant $t$, then $C_i$ and $C_j$ are *disjoint* due to definition 8.

**Definition 10. (valid-time disjoint constraint)** The *valid-time disjoint constraint* over a set of valid-time collections $CS$, $disjoint^v(CS)$, is defined as

$$disjoint^v(CS) :\Leftrightarrow \forall t \in \bigcup_{C_j \in CS} lifespan(C_j) : disjoint^t(CS)$$

A set of valid-time collections $CS$ is temporally disjoint, if no pair of member collections has a common member value at any time point.

The temporal cover and intersection constraints and the semantics of temporal cardinality constraints can be defined accordingly. A temporal partition constraint can be expressed by a combination of a temporal cover and a temporal disjoint constraint.

*Example 7.* The schema depicted in figure 1 demands that the valid-time collections `Residences` and `Offices` are disjoint. In our system, the command

```
create constraint DisjointRO disjoint([Residences, Offices]);
```

creates a valid-time disjoint constraint over the valid-time collections `Residences` and `Offices`.

Previously, we stated that all information is represented as objects, including constraints. This means that with our object-timestamping approach, constraint objects may also be extended to temporal objects having a lifespan. At the moment, we are still investigating this idea of timestamping constraints and the impact they have.

The above definition of a temporal constraint does not lead directly to a good implementation. It is not feasible to implement a constraint checking algorithm which is based on time instants. We use an efficient implementation based on calculations of time on an interval level, using the set-theoretic operations union, intersect and difference of time intervals. Additionally, we exploit the fact that, if a database is consistent at the beginning of a transaction, only the changes made during the current transactions need to be checked.

## 5 Temporal Operations

So far, we have introduced the temporal constructs of TOM. The other aspect of the model is the generalisation of the collection algebra of OM to give equivalent temporal operations. In OM, all algebra operations work on collections of objects and return a result collection of objects. The model has an extensive set of generic operations, including convenience forms for operating over binary collections.

The algebra supports the standard set-based operations of *union*, *intersection* and *difference*. There are also operations to *map* a given function over a collection, to *select* elements of a collection based on a predicate condition and to *flatten* a collection of collections by eliminating one level of nesting. A full description of the algebra is given in [Nor93].

Proposed temporal algebras and query languages tend to neglect the fact that, besides different kinds of temporal selections (e. g. `DURING`, `WHEN`, `MOVING WINDOW` as introduced in [Wuu91, RS93]) or a temporal join, operations such as temporal set difference (temporal negation) or intersection etc. should also be

supported. The TOM model specifies temporal equivalents for all of the algebra operations in OM. Additionally, temporal comparison operators as introduced in [All83] are supported.

There exist two categories of temporal operations in our temporal algebra. The *first category* contains those operations which calculate new lifespans for both result collection and the objects contained in it, for example the *temporal compose operation*, the *temporal cross product* or *temporal set operations*. The *second category* of temporal operations only work on object identifiers while retaining lifespans. Examples are the *temporal inversion* or the *temporal domain* operations.

We will discuss these operations in more detail by considering an example query, explaining how it is evaluated and giving definitions for a few temporal operations. Definitions for all of the operations and more examples are given in [SN97]. In our system, it is possible to either use algebra expressions or an SQL-like syntax for querying. We use algebra expressions in the following examples.

*Example 8.* We would like to know the history of tenants renting one of Herbert's properties. The temporal algebra expression calculating the corresponding result looks like

$$range^v(\sigma^v_{left.name='Herbert'}(Owns) \circ^v inv^v(Rents))$$

This expression can be run as a query in the system the following way:

```
valid range(compose(select (left.name = 'Herbert') Owns, inv(Rents)));
```

Algebra operations with a superscript $v$ denote that they are evaluated using temporal semantics with respect to valid-time. In example 8, all of the operations use temporal semantics. According to the approach proposed in [SBJS96], we use the keyword *valid* to denote that temporal evaluation semantics should be applied for a query. In the above example, the scope of keyword *valid* is the whole query.

In example 8, we first select those binary objects in the temporal association `Owns` which have the object denoting owner Herbert on the left side. We then combine the temporal result collection with binary collection `Rents`. The valid-time compose operation ($\circ^v$) composes out of two binary collections, a new binary collection by taking the objects in the domain of the first and the objects in the range of the second and combining them if they have equal range and domain objects respectively. This operation belongs to the first category of operations where lifespan calculation is done. The formal definition of the temporal compose operation is

**Definition 11. (compose of two valid-time binary collections)** Let $B_1$ and $B_2$ be two valid-time binary collections. The valid-time composition of $B_1$ and $B_2$, $B_1 \circ^v_{set} B_2$, has a lifespan $lifespan(B_1) \cap lifespan(B_2)$ and an extension

$$ext(B_1 \circ^v_{set} B_2) = \{\ll (x,z); ls \gg | \exists y : \ll (x,y); ls_1 \gg \in_{set} ext(B_1) \wedge \\ \ll (y,z); ls_2 \gg \in_{set} ext(B_2) \wedge ls := ls_1 \cap ls_2 \wedge ls \neq \{\}\}$$

Since our collections also have lifespans, we have to define what the lifespan of a resulting valid-time collection shall be. A non-temporal database management system returns an error if one of the arguments of an operation does not exist. In our case, we define that a resulting temporal collection should only cover those time instants when all of the argument collections exist. Thus the result of a valid-time compose operation is valid only during the intersection of the two lifespans of the valid-time collections involved. Note that this also holds for other temporal operations of the first category.

As we can see in definition 11, we combine those pairs of objects where the right object of the first pair is the same as the left object of the second pair during their common time period. In example 8, we want to find tenants of properties owned by Herbert. We combine owner objects in `Owns` with tenant objects in `Rents` through their common property objects. To be able to do that with a temporal compose operation, we first have to invert collection `Rents`. The valid-time inversion operation ($inv^v$) just switches source and target objects of a binary collection, leaving the timestamp the same. This operation belongs to what we earlier called the second category of operations in our temporal algebra. The formal definition of this operation is

**Definition 12. (inverse of a valid-time binary collection)** Let $B$ be a valid-time binary collection. The valid-time inverse of $B$, $\mathtt{inv}^v_{set}(B)$, has a lifespan $lifespan(B)$ and an extension

$$ext(\mathtt{inv}^v_{set}(B)) = \{\ll (y,x); ls \gg \mid \ll (x,y); ls \gg \in_{set} ext(B)\}$$

The result of the compose operation $\sigma^v_{left.name='Herbert'}(Owns) \circ^v inv^v(Rents)$ is a binary collection containing pairs having an owner object on its left and a tenant object on its right side together with their common time periods. Since we are interested in tenant objects of this binary collection, only the range of the binary collection is of interest. The corresponding operation is the temporal range operation ($range^v$), which can be defined similarly to the temporal inversion and also belongs to the second category of temporal operations.

## 6 Conclusions

Experiences in developing the model TOM, together with a prototype implementation, show that our generalisation approach leads naturally to more general models and systems. The generality and orthogonality of the underlying model, in this case OM, are major contributing factors and therefore essential to fully exploit the generalisation approach.

By generalising the notion of an object identifier to a temporal object identifier, everything considered as an object is automatically timestamped. In OM, not only entities, but also collections, associations and even constraints are objects, and hence all may have temporal properties. Additionally, the possibility that objects may have several roles at the same time and evolve by changing roles makes both OM and TOM very powerful for role modelling.

## References


[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11), 1983.

[BFG96] E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology*, pages 342–356. Springer, 1996.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537. IEEE Computer Society Press, 1987.

[GÖ93] I. A. Goralwalla and M. T. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In *Proceedings of the 10th International Conference on the ER Approach*, pages 110–121, 1993.

[GV85] S. K. Gadia and J. H. Vaishnav. A Query Language for a Homogeneous Temporal Database. In *Proceedings of the International Conference on Principles of Database Systems*, 1985.

[Nor93] M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proceedings of the 12th International Conference on the ER Approach*, 1993.

[NSWW96] M. C. Norrie, A. Steiner, A. Würgler, and M. Wunderli. A Model for Classification Structures with Evolution Control. In *Proceedings of the 15th International Conference on Conceptual Modelling*, 1996.

[RS93] E. Rose and A. Segev. TOOSQL - A Temporal Object-Oriented Query Language. In *Proceedings of the 10th International Conference on the ER Approach*, pages 122–136, Dallas, TX, 1993.

[SBJS96] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2*, November 1996.

[SN97] A. Steiner and M. C. Norrie. A Temporal Extension to a Generic Object Data Model. Technical report, Institute for Information Systems, ETH Zürich, 1997.

[Sno95] R. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, USA, 1995.

[Ste95] A. Steiner. The TimeDB Temporal Database Prototype. Institute for Information Systems, ETH Zürich. Available at `ftp://ftp.cs.arizona.edu/tsql/timecenter/TimeDB.tar.gz`, September 1995.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/-Cummings Publishing Company, 1993.

[WD93] G.T.J. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 10, pages 230–247. Benjamin/Cummings Publishing Company, 1993.

[Wuu91] G.T.J. Wuu. SERQL: An ER Query Language Supporting Temporal Data Retrieval. In *Proceedings of the 10th International Phoenix Conference on Computers and Communications*, pages 272–279, 1991.