A Temporal Extension to a Generic Object Data Model

Andreas Steiner and Moira C. Norrie
Institute for Information Systems
ETH Zürich, CH-8092 Zürich, Switzerland

May 7, 1997

TR-15

A TimeCenter Technical Report

Title	A Temporal Extension to a Generic Object Data Model
Author(s)	Andreas Steiner and Moira C. Norrie Institute for Information Systems ETH Zürich, CH-8092 Zürich, Switzerland
Publication History	A shorter version appears in *Proceedings of the 9th Conference on Advanced Information Systems Engineering (CAiSE), 1997*

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, Polytechnic University, New York, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

We present a temporal object model capable of representing the lifespan of objects and also the history of the roles and associations in which they participate. 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. Our model allows objects to participate in several roles simultaneously. Further, metadata may also have temporal properties allowing the lifespan of the object roles themselves, and constraints over these roles, to be modelled. The model is based on the generic object data model OM and the associated 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 database structures to enable entities or entity properties to be timestamped – whether it be with valid or transaction times [LJ88, Wuu91, RS91, SC91, NA93, Sar93, Sno93, EWK93, WD93, GÖ93, BFG96]. Typically, the query language is then extended with temporal properties in the sense of selections with temporal predicates or temporal joins [Wuu91, NA93, EWK93, WD93, 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 [Sno93, Sno95, SBJS96b] and for constraints [RS91, WD93, SBJS96b] 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 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 [SBJS96b, SBJS96a].

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 and query language and constraints. Additionally, metadata can have temporal properties allowing the modelling of role, association and constraint lifespans.

There is an ongoing debate whether or not object-oriented data models should be extended or generalised since, after all, their inherent extensibility can be used to support temporal applications. We have investigated this approach in [SN97a] where we show that it is possible to use an abstract data type for time to model temporal databases and specify temporal queries. Temporal queries however turn out to be rather complex in their form, the specification of temporal constraints is left to the user, and the database system cannot use the special semantics of time for query optimization. We argue that the form of extensibility provided by current object-oriented database systems is not sufficient to support temporal databases. What we would like to have additionally are, for example, extensible object identifiers, extensible query languages and the possibility to overwrite algebra operations. These concepts would help to implement temporal databases with non-temporal object-oriented database management systems, but their usefulness would not be restricted to temporal databases. Applications using spatial databases or versioning could also be supported. For this reason, we have developed a full temporal object-oriented data model.

Our temporal object data model TOM 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. Classifications are represented by the bulk type constructor *collection* and classification structures are built from collections linked by means of *subcollection*, *disjoint*, *cover* and *intersection* constraints over these collections. Explicit support for the representation of relationships between objects is given by association constructs based on a special form of collection referred to as a binary collection. As a result, classification structures may be used to model relationship roles as well as entity roles.

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. As an example of its orthogonality, collections are themselves objects and this enables arbitrary nesting of structures.

Our temporal generalisation of OM is based upon the concept of generalising objects into temporal objects rather than adding temporal properties to objects. Thus, in contrast to most other approaches to temporal object models, we do not extend object types with temporal attributes, but rather extend the fundamental notion of an object identifier into a temporal object identifier. Since collections are also objects, this leads naturally to a representation of the lifespans of both objects and object roles. Further, associations – including the membership associations between objects and the collections to which they belong – are timestamped allowing both object role histories and association histories to be modelled.

Experiences in developing the model TOM, together with a prototype implementation, show that the 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.

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, discussion of further work and also comments on the impact of current extensions to the OM model on its temporal generalisation are given in section 6.

2 Object Data Model OM

The OM data model is used to specify semantic groupings of objects and their interrelationships and thus deals with issues of classification and associations between objects at the same level of abstraction. The characteristics of objects in terms of interface and implementation would be specified by the type system associated with a particular system in which the model is used to support object data management. This separation of typing from classification is beneficial, not only in terms of the universality of the model, but also in distinguishing issues of representation from those of data semantics. Further, it allows an Entity-Relationship style of conceptual modelling to be combined with the power and flexibility of object-oriented systems.

The basic construct of the OM model is the collection in that both objects and relationships are classified into collections of a given membertype. Unary collections are those which have atomic values as elements and represent object roles. Binary collections are those which have pair values as elements and represent relationships between entities. Relationships are actually represented by associations. An association consists of a binary collection together with constraints that specify the roles of objects that may participate in the relationship and the corresponding cardinalities. Note that a collection is itself an object which provides the capability to have collections of collections and so on.

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

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.

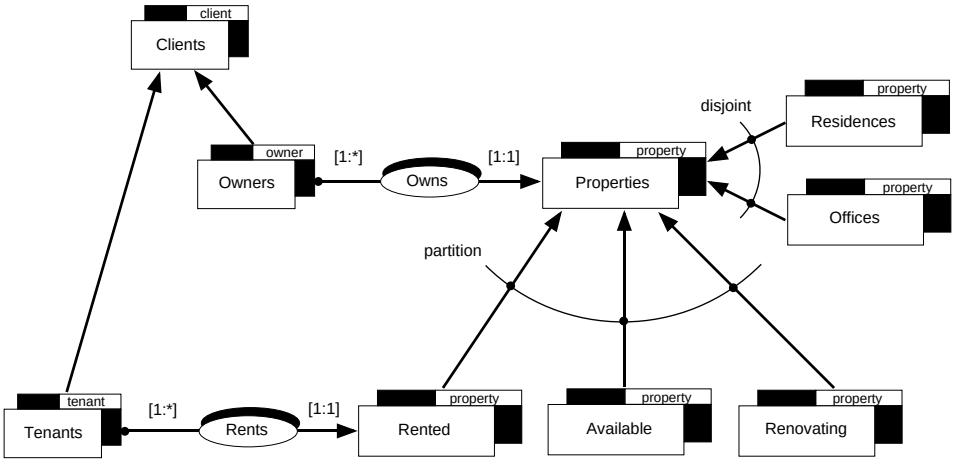

Figure 1: An Example OM Schema Diagram

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 membertype `tenant` of `Tenants` is a subtype of the membertype `client` of `Clients`. Likewise, `owner` is also a subtype of `client`. It is possible that a client may be both a tenant and an owner and hence 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, together with links to the collections related by the association and their respective cardinality constraints. In order that we can specify operations over associations, it is necessary to specify a direction of such a relationship. For example, `Owns` would actually be represented by a collection of pairs of object values such that the first elements of the pairs belong to `Owners` and the second elements belong to `Properties`. We refer to `Owners` as the *source collection* and to `Properties` as the *target collection* of `Owns`.

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 which we call *object metamorphosis*. For example, if a `tenant` object becomes an `owner` object, then the object must gain additional owner properties. OM supports object metamorphosis 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. The algebra includes *select*, *map*, *reduce*, *flatten*, *union*, *intersection* and *difference* operations as well as special operations over binary collections such as *compose*, *inverse*, *domain* and *range*. Descriptions of most of these operations are given in section 5 where the temporal equivalents are described.

In a generalised form of the OM model, collections of different behaviours are supported (as is typical in many object-oriented data models). Collections (unary or binary) may exhibit set or bag properties depending upon whether or not the collection may contain duplicate elements. In addition, they may have an associated ordering. In this paper, we deal only with set collections but note that the concepts generalise to other forms of collections. Further details of the OM model and its algebra are given in [Nor93, Nor92].

As stated above, the OM model is generic and it has formed the basis for object data management on a variety of platforms including C++-based persistent systems [CBHdP93] and the programming language

Oberon [SN97b]. Additionally, 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.

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* but rather extend the *object identifiers* with a timestamp to give temporal object identifiers of the form

$$\texttt{toid} := \ll \texttt{oid}; \texttt{ls} \gg$$

where `oid` is an object identifier and `ls` is a timestamp referred to as the *lifespan* of an object. It expresses, for example, when an object was *valid* (existent) in the real world. Thus, we do not timestamp the values of an object, but the object itself with its overall time of existence and we keep track of the history of its values separately. This means that it is not necessary to calculate the union of all timestamps of values of an object in order to establish the time period during which the object existed.

Timestamps may also be associated with relationships between objects which are represented by member pairs of binary collections. In this case, each pair of object identifiers $(\texttt{oid}_1, \texttt{oid}_2)$ is tagged with a timestamp to give elements of the form

$$\ll (\texttt{oid}_1, \texttt{oid}_2); \texttt{ls} \gg$$

where `ls` is a timestamp as before.

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. For example, we assume that the property leasing company initially only managed properties owned by its parent company. Later, it decided to generalise and also lease properties owned by others. After a while, it ceased to manage other owner's properties while the general leasing market declined. When the market picked up a few years later, it resumed leasing of other people's properties. This can be modelled through the lifespan of the `Owners` collection. Similarly, associations, which represent relationship roles between objects, may also be timestamped.

The next stage to consider is how to model the times at which a particular entity has a particular role, i.e. that an object is a member of a collection. An object may be in several collections at one time and may migrate between collections. Looking at our example schema in figure 1, a tenant is a member of collection `Tenants`. This means that the property leasing company helped him to find a property to rent. It is possible that over some interval, a tenant is also an owner of a property leased by the company. For example, they may own a property in one city and lease another in a different city. In this case, the tenant would appear in both the `Tenants` and `Owners` collections of the company's database during this time period.

An object's *visibility* in a time collection is determined by the overall lifespan of the object, the collection's own lifespan and a membership time specified by the user. Thus, if we look at a `client` object through the `Owners` role, it is only visible during the time period he is actually an owner, even though the object existed before its membership in `Owners`.

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 $ls_1 \Leftrightarrow ls_2$ 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 :*

$$\texttt{toid} := \ll \texttt{oid}; \; ls \gg$$

■

In the following, we use the notation $lifespan(\texttt{toid})$ to reference the (visible) 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 ω represents the *undefined* object identifier.

Definition 2 (snapshot of a temporal object identifier) *Let* `toid` $\in O^v$ *be a temporal object identifier containing* `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 ω is returned. Definition 2 is required for later definitions.

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 $[0 \Leftrightarrow \infty)$. The snapshot operation τ^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 will focus only on temporal values although we suggest to have both non-temporal and temporal values available in a system, along with adequate conversion operations.

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

Valid-time objects are objects having a temporal object identifier. Depending on the role they play (meaning the collection they are member of), they show corresponding property values. In the following, we refer to the *temporal object identifier* of a valid-time object *obj* by *toid*(*obj*), the *(visible) 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 antonia lifespan { [1969 - inf) };
create object moira lifespan { [1970 - inf) };
...
create object herbert lifespan { [1964 - inf) };
...
create object apart1 lifespan { [1980 - 1995) };
create object apart2 lifespan { [1970 - inf) };
...
```

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

A collection contains objects which are of the same membertype. In fact, in its full generality, a collection can contain values of any type – including object values – but, in our discussion here, we focus on the case of collections of objects. Recall that a collection is itself an object.

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 figure 1 as valid-time collections, we first have to define the corresponding membertypes:*

```
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,* $\tau^t(ext(C))$*, is defined as*

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

■

Definition 4 will be needed to define the temporal subcollection relationship (definition 7) and the temporal membership relation (definition 9). 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. We can now define the snapshot of a valid-time collection at a particular time instant to be a non-temporal collection which is valid at this time instant.

Definition 5 (snapshot of a valid-time collection) *Let C be a valid-time collection with a temporal object identifier* `toid` *and an extension $ext(C)$. The* snapshot of the valid-time collection C, $\tau^t(C)$*, is defined as*

$$\tau^t(C) = [\tau^t(toid(C)), \tau^t(ext(C))]$$

■

Note that if the temporal object identifier `toid` of a valid-time collection C is undefined (ω) at time instant t, then the extension of this collection is also undefined at t. This means that the objects in the extension are not visible at t.

3.4 Valid-Time Subcollection Relationship

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

Definition 6 (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 6, 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 7 (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, super-collection C_2 also has to exist, but not vice versa. So, in our example, the lifespan of any subcollection of valid-time collection `Clients` must be contained in the lifespan $[1980 \Leftrightarrow \infty)$.

Example 3 *We show how the subcollections 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. During the first five years, the company only dealt with residential properties. In 1985, they started to deal also with properties used as offices.*

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

create collection Residences
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Offices
   subcollection of Properties type property lifespan{ [1985-inf) };

create collection Rented
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Available
   subcollection of Properties type property lifespan { [1980 - inf) };
create collection Renovating
   subcollection of Properties type property lifespan { [1980 - 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. As stated previously, 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.

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 $\mathtt{v}$ of the added object is the intersection of the lifespan $\mathtt{ls}_O$ of the object with the lifespan of the collection $\mathtt{ls}_C$, intersected with the user specified membership time $\mathtt{t}_{user}$:

$$\mathtt{v} := \mathtt{ls}_C \cap \mathtt{ls}_O \cap \mathtt{t}_{user}$$

Example 4 *We now want to add the valid-time objects of example 1 to valid-time collections created in example 2:*

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

insert object antonia into Tenants during { [1984 - 1996) };
...
insert object moira into Tenants during { [1992 - inf) };
...
insert object herbert into Owners during { [1982 - inf) };
...
insert object apart1 into Rented during { [1980 - 1995) };
...
insert object apart2 into Rented during { [1987 - inf) };
...
```

Andreas is a client of the company since 1980. He found a property 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 membertype (if it is not already dressed with it) and asks for attribute values (e. g. name and profession). Additionally, objects are propagated automatically to supercollections 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 membertype. 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*. Object evolution thus consists of the following steps: firstly change an object's type (or let it gain a new type if necessary) within the type hierarchy (object metamorphosis), possibly adding values for additional properties, and then add the the object to a new collection in the classification structure (object migration).

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 $\mathtt{t}_{user}$ 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

$$\mathtt{v} := \mathtt{ls}_{\mathtt{Owners}} \cap \mathtt{ls}_{object} \cap \mathtt{t}_{user},$$

where $\mathtt{ls}_{\mathtt{Owners}}$ represents the lifespan of collection `Owners` and $\mathtt{ls}_{object}$ corresponds to the lifespan of the client object to be added to collection `Owners`.

Example 5 *Assume Moira and Andreas both decided to buy properties, but remained in the properties already rented and asked the leasing company to find tenants for their properties. Thus, they also became owners in the company's database.*

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

insert object moira into Owners during { [1982 - 1987, 1991 - inf) };
...
```

Andreas bought a property in 1982 and in 1995 he decided to have another company manage his property. Moira actually had her first property managed by the company in 1982. She sold the property in 1987 and bought another one in 1991. When inserting objects, the system again dresses the objects with the corresponding membertype (if needed) and asks for property 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 8 (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. As mentioned previously, given a valid-time binary collection C, then an element of $ext(C)$ may be of the form $\ll (\texttt{oid}_1, \texttt{oid}_2); \texttt{v} \gg$ where $\texttt{oid}_1, \texttt{oid}_2 \in O$ and $\texttt{v}$ is the visible lifespan of the relationship,

$$\texttt{v} := \texttt{ls}_\texttt{C} \cap \texttt{v}_{oid_1} \cap \texttt{v}_{oid_2} \cap \texttt{t}_{user},$$

where $\texttt{ls}_\texttt{C}$ is the lifespan of collection C, $\texttt{v}_{oid_1}$ and $\texttt{v}_{oid_2}$ are the visible lifespans of objects $\texttt{oid}_1$ and $\texttt{oid}_2$ in the source and target collections respectively, and $\texttt{t}_{user}$ the user-specified membership time.

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 an association. This is checked by the system.*

```
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:

```
insert binary object (andreas, apart1) to Rents during { [1980 - 1993) };
insert binary object (herbert, apart2) to Owns during { [1987 - inf) };
...
```

■

In the case that a temporal association references an object which is not visible in the referenced collection (source or target) during the specified time period or does not exist at all, an error message is produced. This means we check for *temporal referential integrity* on both collection and object level.

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. As mentioned before, in this paper we only discuss collections which exhibit set properties.

We present the temporal generalisations of the cover and disjoint constraints. Firstly, we have to redefine the membership relation for a time instant. Then, we define the *valid-time cover* and *valid-time disjoint* constraints over *valid-time collections*.

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 9 (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, ω is never a member of a set of values $\tau^t(ext(C))$. With definition 9, we can now define the valid-time disjoint and cover constraints.

Definition 10 (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*

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

■

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 9.

Definition 11 (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 in the time period during which any of the collections exist.

Definition 12 (cover constraint at a time instant) *Let $t \in \mathcal{T}$ be a time instant, C a valid-time collection, and CS a set of valid-time collections. The* cover constraint at time instant t, *$cover^t(C, CS)$, then is defined as*

$$cover^t(C, CS) :\Leftrightarrow \forall x \in^t_{set} C \ \exists C_j \in CS : x \in^t_{set} C_j$$

■

With definition 12, we can now define the valid-time cover constraint:

Definition 13 (valid-time cover constraint) *Let C be a valid-time collection and CS a set of valid-time collections. The* valid-time cover constraint, *$cover^v(C, CS)$, is defined as*

$$cover^v(C, CS) :\Leftrightarrow \forall t \in lifespan(C) : cover^t(C, CS)$$

■

A set of valid-time collections CS is a valid-time cover of a valid-time collection C, if each member of CS is a subcollection of C and each element of C appears in at least one collection of CS during each time instant of its existence.

The valid-time intersection constraint or the semantics of temporal cardinality constraints can be defined accordingly. Temporal partition constraints can be expressed by a combination of a temporal cover and a temporal disjoint constraint.

Example 7 *The temporal disjoint constraint used in figure 1, demands that at each time point the subcollections* `Residences` *and* `Offices` *exist, they must be disjoint. The valid-time partition constraint demands that the three valid-time collections* `Rented`*,* `Available` *and* `Renovating` *are partition valid-time collection at each time point valid-time collection* `Properties` *exists.*

```
create constraint disjointOR disjoint([Offices, Residences]);
create constraint coverRAR cover(Properties, [Rented, Available, Renovating]);
create constraint disjointRAR disjoint([Rented, Available, Renovating]);
```

■

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 of this. Currently, our system treats constraint objects as non-temporal objects, meaning, that they are checked over a lifespan $[0 \Leftrightarrow \infty)$ as long as they are present in the system.

The above definitions of temporal constraints do 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 Collection Algebra

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. A full description of the algebra is given in [Nor93, Nor92].

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, SC91, RS93]) or a temporal join, operations such as temporal set difference (temporal negation) or intersection 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 operations in our temporal algebra. The *first category* contains those operations which calculate new lifespans for both the result collection and the objects contained in it. For example, this category includes the *temporal composition 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.

Since collections contain values of a specific *type* in their extensions, we have to specify the membertype of a resulting collection for a given collection operation. [Nor93, Nor92] use the notion of *least common supertype* and *greatest common subtype* to determine the membertype of the result collection.

The *common supertype (upper bound)* of two types t_i and t_j is any type t_k such that $t_i \leq_t t_k$ and $t_j \leq_t t_k$, where $\leq_t$ denotes a subtype relationship. If t_k is a common supertype of t_i and t_j, such that for any other common supertype t_l of t_i and t_j, $t_k \leq_t t_l$, then t_k is the *least common supertype (least upper bound)* of t_i and t_j which is written as $t_k = t_i \sqcup t_j$.

Similarly, the *greatest common subtype (greatest lower bound)* of t_i and t_j is defined as a type t_k such that for any other common subtype t_l, $t_l \leq_t t_k$, which is written as $t_k = t_i \sqcap t_j$. It can be easily shown that both $t_i \sqcup t_j$ and $t_k = t_i \sqcap t_j$ are unique.

We will now discuss the temporal algebra operations in more detail by considering some example queries, explaining how they are evaluated and giving definitions for the temporal operations. In our system, it is possible to either use algebra expressions or an SQL-like syntax for querying.

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 in the system as a query using either the algebra expression

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

or an SQL-like statement

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

The extent of a resulting collection, having its own lifespan $[1982 \Leftrightarrow \infty)$*, contains objects with the following visible lifespans and property values:*

VALID	Profession	Name
{[1995-inf)}	Assistant	Andreas
{[1992-inf)}	Professor	Moira

■

Operations in an algebra expression having 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 [SBJS96b, SBJS96a], we use the keyword *valid* to denote that temporal evaluation semantics should be applied. 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. The valid-time selection is defined the following way:

Definition 14 (valid-time selection in a valid-time collection) *Let* C_1 *be a valid-time collection of type* `t` *and* P *be a function that maps each element of* C_1 *to one of the Boolean values* `true` *or* `false`*. The* valid-time selection *of* C_1 *using function* P*,* $C = \sigma^v_P(C_1)$*, mapping collection* C_1 *to a collection* C *of type* `t` *and valid-time* $lifespan(C) = lifespan(C_1)$*, is then defined as*

$$\forall t \in lifespan(C_1), \forall x \in C_1 : P(\tau^t(x)) = \texttt{true} \Leftrightarrow x \in^t_{set} C$$

■

We then combine the temporal result collection of the selection operation with binary collection `Rents`. The valid-time composition 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 composition operation is

Definition 15 (composition of two valid-time binary collections) *Let* B_1 *and* B_2 *be two valid-time binary collections of types* $(\mathtt{t}_1, \mathtt{t}_2)$ *and* $(\mathtt{t}_3, \mathtt{t}_4)$ *respectively. The valid-time composition of* B_1 *and* B_2*,* $B_1 \circ^v_{set} B_2$*, is of type* $(\mathtt{t}_1, \mathtt{t}_4)$ *and has a lifespan equal to* $lifespan(B_1) \cap lifespan(B_2)$ *and an extension*

$$ext(B_1 \circ^v_{set} B_2) = \{\ll (x,z); v \gg | \exists y : \ll (x,y); v_1 \gg \in_{set} ext(B_1) \wedge \ll (y,z); v_2 \gg \in_{set} ext(B_2) \wedge v := v_1 \cap v_2 \wedge v \neq \{\}\}$$

where v_1 *and* v_2 *denote the* visible lifespans *of the objects in the corresponding collections.*

■

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 composition 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 15, we combine those pairs of objects where the right object of the first pair is the same as the left object of the second object 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 composition 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 16 (inverse of a valid-time binary collection) *Let B be a valid-time binary collection of type $(\mathtt{t}_1, \mathtt{t}_2)$. The valid-time inverse of B, $\mathtt{inv}^v_{set}(B)$, is of type $(\mathtt{t}_2, \mathtt{t}_1)$, has a lifespan equal to $lifespan(B)$ and an extension*

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

■

The result of the composition 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.

Definition 17 (range of a valid-time binary collection) *Let B be a valid-time binary collection of type $(\mathtt{t}_1, \mathtt{t}_2)$. The valid-time range of B, $\mathtt{range}^v_{set}(B)$, is of type $\mathtt{t}_2$ and has a lifespan equal to $lifespan(B)$ and an extension*

$$ext(\mathtt{range}^v_{set}(B)) = \{\ll y; v \gg \mid \exists x : \ll (x,y); v \gg \in_{set} ext(B)\}$$

■

The next example uses a temporal set difference and a temporal cross product operation. The temporal cross product operation is similar to the temporal composition in that it calculates the common lifespan of both collections and objects involved and it returns a valid-time binary collection. Its arguments, however, are *unary* valid-time collections.

Example 9 *We would like to find those residences and the corresponding time period during which no higher priced offices exist. The corresponding algebra expression looks like*

$$Residences \Leftrightarrow^v domain^v(\sigma^v_{left.price<right.price}(Residences \times^v Offices))$$

We can query the system either with the algebra expression

```
valid Residences -
      domain(select (left.price < right.price) Residences * Offices);
```

where the asterisk stands for the cross product operation, or the SQL-like statement

```
valid
  select r in Residences
  where not exists (select o in offices where r.price < o.price);
```

■

All of the operations in the algebra expression have temporal semantics. This is denoted by the keyword `valid` at the beginning of the query whose scope again is the whole query. The valid-time cross product $Residences \times^v Offices$ generates pairs of object identifiers together with their common visible lifespan. It returns a collection of valid-time binary objects containing pairs of non-temporal object identifiers together with a lifespan which is calculated by the intersection of the visible lifespans of the objects involved. Formally, the temporal cross product is defined as

Definition 18 (valid-time cross product of collections) *Let C_1, C_2 be valid-time collections of types $\mathtt{t}_1$ and $\mathtt{t}_2$ respectively. The* valid-time cross product *of C_1 and C_2, $C_1 \times^v_{set} C_2$, returns a binary valid-time collection of type $(\mathtt{t}_1, \mathtt{t}_2)$ having a lifespan equal to $lifespan(C_1) \cap lifespan(C_2)$ and an extension*

$$ext(C_1 \times^v_{set} C_2) = \{\ll (oid(x), oid(y)); v \gg | x \in_{set} ext(C_1) \wedge y \in_{set} ext(C_2) \wedge \\ v = lifespan_{C_1}(x) \cap lifespan_{C_2}(y) \wedge v \neq \{\}\}$$

The lifespans $lifespan_{C_1}(x)$ and $lifespan_{C_2}(y)$ denote the visible lifespans of the objects in the corresponding collections.

■

In the result of *Residences* $\times^v$ *Offices*, we then select those pairs of residence and office objects where the residence's price was lower than the office's price (together with the time period during which this is true). Last, the valid-time difference of the *domain* of the resulting valid-time binary collection and collection `Residences` is calculated returning those residences with time periods for which no higher priced office can be found. The *temporal domain operation* can be defined similarly to the temporal range operation (definition 17). The difference is that the temporal domain returns the objects on the left side in a binary collection.

Definition 19 (domain of a valid-time binary collection) *Let B be a valid-time binary collection of type $(\mathtt{t}_1, \mathtt{t}_2)$. The valid-time domain of B, $\mathtt{domain}^v_{set}(B)$, is of type $\mathtt{t}_1$, has a lifespan equal to $lifespan(B)$ and an extension*

$$ext(\mathtt{domain}^v_{set}(B)) = \{\ll x; v \gg | \exists x :\ll (x, y); v \gg \in_{set} ext(B)\}$$

■

The valid-time difference, union and intersection operations belong to the first category of operations. Temporal set difference in our model is defined as

Definition 20 (valid-time difference of collections) *Let C_1 and C_2 be valid-time collections of membertypes $\mathtt{t}_1$ and $\mathtt{t}_2$ respectively. The* valid-time difference *of C_1 and C_2, $C = C_1 \Leftrightarrow^v_{set} C_2$, mapping the two collections to a collection C of membertype $\mathtt{t}_1$ and $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$, is defined as*

$$\forall t \in lifespan(C) : \tau^t(ext(C)) = \{x | x \in^t_{set} C_1 \wedge x \notin^t_{set} C_2\}$$

■

Valid-time union and intersection can be defined in a similar style to definition 20. The type of the result collections of valid-time union and intersection operations are described using the notions of greatest common subtype and least common supertype.

Definition 21 (valid-time union of collections) *Let C_1 and C_2 be valid-time collections of membertypes $\mathtt{t}_1$ and $\mathtt{t}_2$. The* valid-time union *of C_1 and C_2, $C = C_1 \cup^v_{set} C_2$, mapping the two collections to a collection C of type $\mathtt{t} = \mathtt{t}_1 \sqcup \mathtt{t}_2$ and $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$, is defined as*

$$\forall c \in lifespan(C), \forall x : \tau^t(ext(C)) = \{x | x \in^t_{set} C_1 \vee x \in^t_{set} C_2\}$$

■

Definition 22 (valid-time intersection of collections) *Let C_1 and C_2 be valid-time collections of membertypes $\mathtt{t}_1$ and $\mathtt{t}_2$. The* valid-time intersection *of C_1 and C_2, $C = C_1 \cap^v_{set} C_2$, mapping the two collections to a collection C of type $\mathtt{t} = \mathtt{t}_1 \sqcap \mathtt{t}_2$ and a lifespan $lifespan(C) = lifespan(C_1) \cap lifespan(C_2)$, is defined as*

$$\forall c \in lifespan(C), \forall x : \tau^t(ext(C)) = \{x | x \in^t_{set} C_1 \wedge x \in^t_{set} C_2\}$$

■

Our definitions of valid-time union, intersection and difference ensure that the operations are orthogonal to each other in the sense that for example the valid-time intersect operation can be expressed using the valid-time difference operation:

$$C_1 \cap^v C_2 = C_1 \Leftrightarrow^v (C_1 \Leftrightarrow^v C_2)$$

6 Conclusions

We propose a temporal object-oriented data model which not only generalises data model structures to support time, but considers all parts of a data model by temporally generalising data structures, constraints and collection algebra.

Rather than extending the data structures with additional properties, we adopt the approach of extending the notion of object identifiers by adding timestamps to them. The underlying model OM is *general* in the sense that entities, collections, associations and even databases are considered as objects. Further, it is *generic* in the sense that it is not based on a specific type system but can be used in a variety of programming language environments and implementation platforms. These advantages carry over to the temporal model TOM. By generalising the notion of an object identifier to a temporal object identifier, everything considered as an object can automatically be timestamped. 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 models.

OM also supports rich sets of both constraints and operations which have been temporally generalised. We claim that it is important to consider not only model-inherent, but also user-defined constraints, when defining a temporal model. Also, whereas most other proposals choose to generalise only a few of the operations, or add new temporal operations, we advocate that the algebra should be fully generalised.

The TOM system was implemented in Prolog and our experiences show that the generality and orthogonality of both the OM model and, consequently, the temporal model TOM, turned out to be very beneficial also for the implementation.

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.

[CBHdP93] V. J. Cahill, R. Balter, N. Harris, and X. Rousset de Pina, editors. *The Comandos Distributed Application Platform*. Springer-Verlag, 1993.

[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.

[EWK93] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. A Temporal Model and Query Language for EER Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 9, pages 212–229. Benjamin/Cummings Publishing Company, 1993.

[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.

[LJ88] N. Lorentzos and R. G. Johnson. TRA: A Model for a Temporal Relational Algebra. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pages 99–112, 1988.

[NA93] S. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL . In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings Publishing Company, 1993.

[Nor92] M. C. Norrie. *A Collection Model for Data Management in Object-Oriented Systems.* PhD thesis, University of Glasgow, Dept. of Computing Science, Glasgow G12 8QQ, Scotland, December 1992.

[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.

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proceedings of the 10th International Conference on the ER Approach*, 1991.

[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.

[Sar93] N. Sarda. HSQL: A Historical Query Language. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 110–138. Benjamin/Cummings Publishing Company, 1993.

[SBJS96a] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-147r2*, November 1996.

[SBJS96b] 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.

[SC91] S. Y. W. Su and H. M. Chen. A Temporal Knowledge Representation Model OSAM*/T and Its Query Language OQL/T. In *Proceedings of the International Conference on Very Large Databases*, pages 431–442, 1991.

[SN97a] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Database Systems for Advanced Applications (DASFAA)*, 1997.

[SN97b] J. Supcik and M. C. Norrie. An Object-Oriented Database Programming Environment for Oberon. In *Proc. of the Joint Modular Languages Conference (JMLC'97)*, Linz, Austria, March 1997.

[Sno93] R. Snodgrass. An Overview of TQuel. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 141–182. Benjamin/Cummings Publishing Company, 1993.

[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.