Implementing Temporal Databases in Object-Oriented Systems

A. Steiner
Institute for Information Systems
ETH
CH-8092 Zurich, Switzerland
steiner@inf.ethz.ch

M. C. Norrie
Institute for Information Systems
ETH
CH-8092 Zurich, Switzerland
norrie@inf.ethz.ch

Abstract

We present a temporal object data model, query language and system that support temporal database applications. We then show how equivalent temporal constructs and operations could be provided in existing object-oriented database management systems (OODBMS) and describe how we did this in the O_2 system. A comparison of the two resulting systems highlights the current limitations to the notions of extensibility supported in existing OODBMS.

Keywords Object-Oriented Databases, Temporal Databases

1 Introduction

There is an on-going debate as to whether it is necessary to extend data models with specific temporal constructs and operations to support temporal databases, or, it is sufficient to model temporal properties using date/time attributes (e. g. [18]). In the case of object-oriented database systems (OODBMS), the latter case is argued more strongly given the inherent extensibility of these systems.

We believe that it is important to study both approaches to determine their relative merits and appreciate how far one can go in supporting the development of temporal databases without making changes to the underlying data model and system. Actually, this second point is important, not only for temporal databases, but also for other special forms of database such as spatial databases and domain specific databases such as those for engineering or medical applications.

Further, even though, as we show in this paper, there are potential gains in both convenience and optimisation by incorporating temporal constructs and operations into the model and system, it is still necessary to consider how best to support such applications in current OODBMS.

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1–4, 1997.

In this paper, we report on investigations of both approaches to implementing temporal databases in object-oriented systems. First, we present a system based on an object-oriented data model, OM, which we extended with temporal constructs and operations. The extended temporal model, TOM, is orthogonal in that it allows valid times to be associated with objects, collections of objects and constraints over these collections such as disjointness. The OM model has an associated algebra over collections of objects and this was extended with temporal equivalents. A prototype system for the extended model, TOM, has been implemented in Prolog and we give examples to show how applications can be modelled and temporal queries expressed in the system. As we will describe, this model can be considered as a generalization of many proposed temporal object-oriented data models and therefore its implementation is typical of our first approach.

Second, we describe how we implemented a temporal application in the O_2 OODBMS. The O_2 system has a `Date` class, with associated methods, which can be imported from the library schema O_2Toolkit, but otherwise has no explicit support for temporal applications. We define a `TempObject` class and also temporal equivalents of query operations and use these to implement a temporal database. We show how temporal queries can then be expressed in terms of the O_2 query language OQL. Further, we discuss the issues of making these temporal constructs and functions available for other applications by means of O_2 schema importation.

We discuss the advantages of the first approach in terms of expressiveness, convenience, query processing and also storage optimisation. However, we also discuss the disadvantages of incorporating particular temporal (or spatial) models into a given OODBMS in that the resulting system can be too restrictive. Also, if this approach were adopted for all special forms of database systems, then either the resulting system would be too complex or a proliferation of specialist database systems would result. A real problem with the latter, is that many applications span specialist areas e.g. applications dealing with both temporal and spatial data. We

propose an alternative approach, where certain key constructs should be incorporated into the object data model that then allows extensibility features to be exploited to tailor the system to a specific temporal model. In particular, we make the case for extensibility of, not only class structures, but also of object identifier and querying mechanisms.

In section 2, we present the basic notions required to support temporal databases. Section 3 provides an overview of the temporal object data model, TOM, and its associated algebra and, by means of examples, describes the system we developed based on this model. We describe how some of the basic constructs and operations of TOM were implemented in the O_2 system in section 4. A comparison of the approaches of sections 3 and 4 is given in section 5. Concluding remarks are given in section 6.

2 Extensions for Time in Databases

In order to implement temporal applications, non-temporal database systems need to be enhanced in three ways. First, the data structures have to be extended to record the time information. Second, new operations using the additional temporal semantics of the data have to be provided in order to query and modify temporal data. Third, temporal constraints must be expressible.

Usually, extending the data structures with time attributes does not cause any severe problems. When timestamping data, two different time dimensions can be distinguished. *Valid time* records time when data was true in reality. *Transaction time* records when data was stored in the system. To store valid time data, two additional attributes of a type **Date**, **VTS** (Valid Time Start) and **VTE** (Valid Time End), can be added to (maybe already existing) nontemporal data structures denoting the start and the end point of a valid time interval. The same can be done for transaction time. In this paper, we concentrate on how to extend data structures with valid time. However, the ideas presented could easily be generalised to deal also with transaction time.

Since operations on intervals are not closed (e. g. the difference of two time intervals might result in a *set* of intervals), we use sets of intervals called *temporal elements* [5] for timestamping. For simplicity, we assume in our examples throughout this paper a granularity of *year* when using dates and will express them using only two digits, i.e. 90 for 1990. Of course any other granularity for dates could be chosen.

An important distinction in proposed temporal relational data models is which part of the data structure is actually timestamped. Either *tuple* (e. g. [10, 17]) or *attribute timestamping* (e. g. [5, 24]) might be applied. In the case of temporal object data models, there are three major possibilities. Firstly, timestamping may at the attribute (as in [15]) or object level. Secondly, if timestamping is at the object level, it may be either at the type level (as in [7]), in which case a special temporal attribute is included, or at the identifier level, in which case a special temporal object identifier is used. We advocate the use of temporal object identifiers and adopt these in our temporal object model, TOM. However, in existing OODBMS, object timestamping must be done at the type level.

Adding new *temporal* operations causes more problems. For relational database systems, *temporal* algebras have been defined (see e. g. [19, 21]). The *temporal* algebra operations have to be implemented either directly in the system or as an additional layer to the non-temporal relational database technology (as for example done in [22]). This approach also requires an extension of the query language supported by the database system in order to use these temporal operations together with the non-temporal functionality of the system.

OODBMS, such as O_2, allow the functionality of the system to be extended by classes, methods and/or functions. This feature can be used to add temporal classes and time functions which, together with the non-temporal operations already supported by the system, can be used to write temporal queries similar to those in proposed temporal relational data models. We discuss later in detail the limitations of this approach in existing OODBMS.

Another issue is that of support for temporal constraints. For example, for temporal relational databases, referential integrity should be checked also with respect to time. Most existing OODBMS, including O_2, leave the specification and maintenance of constraints entirely up to the application programmer. The programmer has to provide special methods which, on data modifications, check whether or not the update is allowed. In this case, special methods to check *temporal* constraints can also be expressed by the application programmer in the same way.

The object data model (OM) and system (OMS) used as the basis for our temporal object model (TOM) and system (TOMS) does have support for constraints over collections of objects and, therefore, our temporal system also has explicit support for temporal constraints.

3 A Temporal Object Model

In this section, we present the temporal object data model TOM and a system based on this model. With this approach of designing a specific temporal model, it is possible to build explicit support for the temporal constructs into the system and query language. We start by describing the main features

of our temporal object model and then, by means of examples, introduce the system and its query language.

Our temporal object data model, TOM, is based on the generic object-oriented data model, OM [12] and exhibits many of the features found in various temporal object-oriented models, e. g. [15, 25, 7, 2], but in a more generalized form, as we will demonstrate in this paper.

The OM model strictly separates typing from classification in such a way that classification structures model the roles of objects rather than their representation. 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. Collections may be either unary, in which case the members are atomic, or binary, in which case the members are pairs. Binary collections are used to represent associations between collections.

Other key features of the OM model that impact on the temporal model are its collection algebra which defines generic operations over collections of objects, the model's support for object and relationship evolution [13] 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 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

$$\texttt{toid} := \langle\langle \texttt{oid}; \texttt{ls} \rangle\rangle$$

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.

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 (o_1, o_2) is tagged with a timestamp to give elements of the form

$$\langle\langle (o_1, o_2); \texttt{ls} \rangle\rangle$$

where `ls` is a timestamp as before.

A timestamp is actually a temporal element, which means that we can model the existence of an object with respect to a particular application. For example, the timestamp of an `employee` object may represent the various periods during which that employee worked for a company.

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, a company may have representatives in several countries and have different collections to represent the corresponding semantic groupings. In the event that the company ceases to trade with a given country, or even that a country ceases to exist, a collection may not be valid any more. At a later date, trading may resume and the collection is once more valid. Similarly, associations, which represent relationships 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. For example, a person may be a member of collection `Persons` during his whole life and, for certain periods, also be a member of collections `TennisTeam` and `Employees`. The `visibility` of an object c in a given collection C is given by

$$\texttt{ls}_o \cap \texttt{ls}_C \cap \texttt{t}_{user}$$

where $\texttt{ls}_o$ is the lifespan of o, $\texttt{ls}_C$ is the lifespan of C and $\texttt{t}_{user}$ is a user-specified membership time. Our approach contrasts with that of [15, 25, 2], where the lifespan of an object is derived as the union of all class membership timespans of this object. While their approach only applies to systems in which database objects must belong in at least one collection, ours is more typical of current commercial OODBMS where objects may persist independently.

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. Further, since constraints are also represented as objects in our system, they also can be timestamped.

In our current system, timestamping is supported only at the level of objects and the question then arises as to how to associate valid times with the attribute values of objects. For example, how the salary history of an employee could be represented. The general rule is that an attribute must be promoted to the level of objects or relationships in order to be timestamped. The salary history of an employee could be represented as a set of timestamped objects of type `SalaryHistory` where each

object refers to a specific salary period. We believe this approach is sufficiently expressive and there are great benefits obtained from the resulting simplicity and uniformity of the model.

So far, we have introduced the temporal constructs of TOM. The other aspect of the model is the extension of the collection algebra of OM with 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. In the remainder of this section, we will consider only unary collections for the sake of simplicity.

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 [12, 11].

The TOM model specifies temporal equivalents for these operations. There are only two operations which refer to type information or attribute values, namely the *projection* (special case of *map*) and *selection* operations. All other operations do not refer to any attribute values and work on the object level. Thus, most of the calculations can be performed handling only temporal object identifiers. It is beyond the scope of this paper to describe the resulting algebra in full, but details are given in [23]. The following examples will sketch some of the ideas behind these temporal operations.

Having presented the main features of the temporal model, TOM, we now describe how these constructs and operations are made available to the application programmer through our system and its associated query language.

Consider the classic example of representing the history of employees in a department.

Example 1 *The type definitions for department and employee objects could be defined as follows:*

```
create type department(
                DNo:     integer,
                Name:    string,
                Members: collection(employee));
create type employee(
                Name:   string,
                Salary: integer,
                Dept:   department);
```

Then timestamped collections of department objects and of employee objects belonging to Information Systems could be defined as:

```
create collection Departments
  type department
  lifespan {[80-inf)};

create collection IS_Staff
  type employee
  lifespan {[90-inf)};

create collection Math_Staff
  type employee
  lifespan {[80-inf)};
```

Since these collections are timestamped objects, they have an object identifier plus timestamp associated with them. For example, `IS_Staff` *might have a temporal object identifier* $\ll 9; \{[90-\infty)\} \gg$.

■

Now we show some examples of queries formulated in the query language of TOM. The language has a syntax similar to OQL, but the keyword *valid* at the beginning of a query denotes that it must be evaluated temporally, as proposed for SQL/Temporal in [21, 20].

Example 2 *Assume again collections* `IS_Staff` *and* `Math_Staff` *as defined in example 1. Then the query for the highest salaries is*

```
valid
  select s1.Name, s1.Salary
  from   s1 in IS_Staff
  where  not exists
           ( select *
             from   s2 in IS_Staff
             where  s1.Salary < s2.Salary );
```

Finding employees which are members of both `IS_Staff` *and* `Math_Staff` *can be expressed as*

```
valid
  IS_Staff intersect Math_Staff
```

■

Besides the selection and projection operations, attribute values of objects are accessed when a result is presented to the user. Depending on the resulting timespans of the objects in the result collection, the corresponding value histories of the objects are printed out.

4 Temporal Databases in O_2

In this section, we show how the object-oriented database system O_2 might be extended to support temporal applications. The system architecture of O_2 is divided into several layers. The base of O_2 is the O_2Engine which provides all the features of a database system and all the features of an object-oriented system [14]. Several programming interfaces are built on top of the O_2Engine. We use the O_2C and OQL interfaces to support temporal functionality (see figure 1). O_2C is a fourth generation language based on the programming language C. OQL is an SQL-like query language.

The approach we have chosen to extend O_2 with time is based on the idea of a root class supporting time attributes and special methods operating

on them. These methods, used together with the non-temporal query language OQL, allow temporal queries to be written. Another approach would be to use the C interface of O_2 and supply temporal functionality and maybe even a new *temporal* query language by adding a library written in C. However, we want to consider the general application programming level of such a system, and not consider extensions at lower levels which are more the task of the database engineer. In any case, such extensions would only result in significant changes to support for temporal databases if major components, such as the query processor, were replaced.

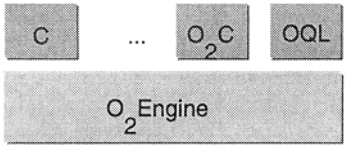

Figure 1: Part of the System Architecture of O_2

We also would like to supply our temporal extension to other users. In O_2, a schema consists of named objects, class definitions, global functions and applications programs. However, only classes and named objects may be exported from schemas. This means that all of the temporal functions have to be methods of an exported class. However, as we will see, this leads to an unnatural, asymmetric way of writing temporal queries.

First, we describe the structural part of the root class `TempObject`. We then show how we implemented functions operating on timestamps and give a few examples of temporal queries written in OQL.

4.1 Timestamps in O_2

The time intervals, our basic time units for timestamps, are defined as follows:

```
type Interval : tuple(VTS : Date, VTE : Date);
```

Time intervals, closed at the lower and open at the upper bound, consist of a starting (*VTS*, Valid Time Start) and an ending year (*VTE*, Valid Time End). For example, [90-96) denotes the time period of January 1, 1990 to December 31, 1995.

Since it is not possible to change object identifiers in O_2, we choose to have object timestamping but must do so at the type level.

In O_2, an object which is a member of two different sets has the same attribute values in both sets. As with our temporal model, TOM, we would like to be able to model the fact that an object's roles vary over time and, further, it may have many roles at the same time. To do this, we must be able to represent the logical entity and its temporal instances as shown in figure 2.

Assume for example a university having research staffs `Math_Staff` and `IS_Staff` among others. Each

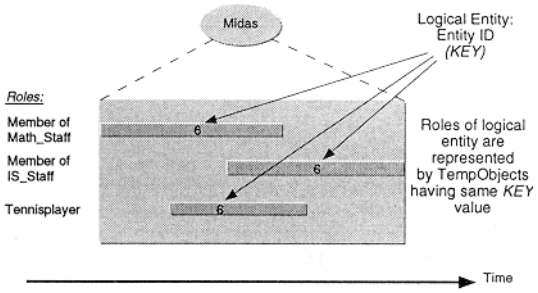

Figure 2: Roles represented in O_2

staff contains a set of employees. We want to add employee *Midas* to `IS_Staff` and `Math_Staff` because he was, or still is, a member of both `IS_Staff` and `Math_Staff`. To do this, we have to create two employee objects *Midas*, set the attribute values and add it to `Math_Staff` for example with a valid time [93-97) and to `IS_Staff` with [96-∞).

The problem now is to determine which objects in O_2 denote the same real world entity. We could, for example, add a set valued attribute to each object which contains references to other objects which actually stand for other roles of the real world entity. Or we could have objects denoting a role pointing to a root object, which is similar to what has been proposed for views in O_2 [3]. Another approach is to add a key value to objects. This key value is unique for one real world entity. Objects in the database system with the same key value refer to the same real world entity. The first and second approaches lead to quite a lot of pointer chasing and it is hard to keep the references consistent. For simplicity, we use the third approach.

The structural part of our root class `TempObject` for temporal objects thus is defined the following way:

```
class TempObject inherit Object
  public type
    tuple(VALID : set(Interval),
          KEY   : integer)
  method
    ...
end;
```

where `KEY` is some form of system-generated entity identifier.

We now derive any class whose instances shall be timestamped from class `TempObject`. Our example is the same as that of the previous section, namely, recording employee histories.

Example 3 *We define* `Departments` *to be a set of department objects. For each department, we keep track of the department number, the name of the department and its members. The members of a department are represented as a staff object consisting of a set of employees.*

```
class Departments inherit Object
```

```
  public type set(Department)
end;

class Department inherit TempObject
  public type
    tuple(DNo     : integer,
          Name    : string,
          Members : Staff)
end;

class Staff inherit TempObject
  public type tuple(members : set(Employee))
end;

class Employee inherit TempObject
  public type
    tuple(Name   : string,
          Salary : integer,
          Dept   : Department)
end;
```

We define class **Departments** *to be a set of timestamped* **Department** *objects. Class* **Department** *contains an attribute* **Members** *which is a* **Staff** *object. Objects of type* **Staff** *contain a set of timestamped* **Employee** *objects and a timestamp denoting when the staff itself existed.* ■

We timestamped objects of classes **Department** and **Employee** and **Staff**. Of course, we could have defined class **Departments** to be a subclass of **TempObject**, too.

This means that we are also able to timestamp *collections* of objects by making these collections into objects. We express, for example, when a staff object itself existed. Of course, a staff object may only have members during its own lifespan.

We are now able to timestamp objects, but not attributes as with the TOM model. This means that, as with TOM, if we want to record the salary history of employees then these must be stored as objects. This is shown in the following example:

Example 4 *We store the salary history of an employee as a set of timestamped salary objects :*

```
class Salary_history inherit TempObject
  public type
    tuple(Salary : integer)
end;

class Employee inherit TempObject
  public type
    tuple(Name     : string,
          Salaries : set(Salary_history),
          Dept     : Department)
end;
```

■

In the following, we will refer to the approach shown in example 3 to keep queries as simple as possible.

4.2 Operations on Timestamps

The next step is to come up with functions which refer to the timestamps and can perform temporal calculations on them. As mentioned before, we implement these functions as methods of class **TempObject** in order to be able to export them.

We basically support the methods **T_INTERSECT**, **T_MINUS** and **T_FLATTEN** to write queries equivalent to those that can be expressed in temporal relational algebra. **T_INTERSECT** calculates the intersection of the time intervals in two sets. For example, the intersection of {[94-∞)} and {[90-96)} is the set {[94-96)}. **T_MINUS** calculates the temporal difference of sets of time intervals. The temporal difference of the interval sets {[94-∞)} and {[90-96)}) is the interval set {[96-∞)}. **T_FLATTEN** is used to flatten sets of temporal elements which may result from queries on sets of temporal objects where only the timestamp attributes of these objects are returned.

The signatures of these methods added to class **TempObject** look like

```
class TempObject inherit Object public type
  tuple(VALID : set(Interval),
        KEY   : integer)
  method
    public T_INTERSECT(T : set(Interval)) :
             set(Interval),
    public T_MINUS(T : set(Interval)) :
             set(Interval),
    public T_FLATTEN(S : set(set(Interval))) :
             set(Interval)
  ...
end;
```

Additionally, we support the temporal comparison predicates *before*, *meets*, *overlaps* and so on as proposed by [1]. They are implemented as functions on two intervals, returning a Boolean value.

With the above methods and Boolean functions, we can now express temporal queries in O_2 *OQL*.

Example 5 *Assume an instance* **IS_Staff** *of class* **Staff** *containing employee objects with the following values*

Name	Salary	Dept	KEY	VALID
Andreas	10000	IS	1	{[93-∞)}
Alain	9000	IS	2	{[95-∞)}
Antonia	11000	IS	3	{[96-∞)}
Martin	8000	IS	4	{[92-94)}
Martin	10500	IS	4	{[94-96)}
Moira	20000	IS	5	{[94-∞)}
Midas	30000	IS	6	{[96-∞)}

and objects of instance **Math_Staff** *having values*

Name	Salary	Dept	KEY	VALID
Moira	8000	Math	5	{[86-90)}
Midas	40000	Math	6	{[93-97)}
John	15000	Math	7	{[94-∞)}

An instance of class **Departments** *shall contain two objects (assuming object identifiers* **IS** *and* **Math***), with the following values:*

DNo	Name	Members	KEY	VALID
3	Inf Systems	IS_Staff	10	{[90-∞)}
9	Mathematics	Math_Staff	11	{[80-∞)}

We would like to know the history of the highest salaries of IS_Staff. *This* temporal *query is expressed as*

```
select
 tuple(Name   : s1.Name,
       Salary : s1.Salary,
       VALID  : s1.T_MINUS
                (s1.T_FLATTEN
                  (select s1.T_INTERSECT(s2.VALID)
                   from s2 in IS_Staff.members
                   where s1.Salary < s2.Salary)))
from s1 in IS_Staff.members;
```

First, we find out for each employee, during which time periods there were other employees earning more than himself. In a second step, we project objects in IS_Staff *to attributes* Name *and* Salary *and calculate the temporal difference of the valid time of each object and the time periods found when other employees earn more. This results in*

Name	Salary	VALID
Andreas	10000	{[93-94)}
Alain	9000	{ }
Antonia	11000	{ }
Martin	8000	{[92-93)}
Martin	10500	{ }
Moira	20000	{[94-96)}
Midas	30000	{[96-∞)}

Empty timestamp sets denote that these employees never earned more than everyone else. ■

Results having empty timestamp sets usually are not of interest. If they should not be presented, we have to add a corresponding selection condition. In order to do *selections* on temporal attributes, we either have to repeat the time calculations of the **select** clause in the **where** clause or do the temporal selections as the last step on the result of a subquery.

Example 6 *We want to find those employees who earned more than anyone else for* more than a year. *The query of example 5 is used as a subquery and the temporal selection is done on the result of the subquery. We can express this in* OQL *as*

```
select s
from s in
 (select
   tuple(Name   : s1.Name,
         Salary : s1.Salary,
         VALID  : s1.T_MINUS
                  (s1.T_FLATTEN
                    (select s1.T_INTERSECT(s2.VALID)
                     from s2 in IS_Staff.members
                     where s1.Salary < s2.Salary)))
  from s1 in IS_Staff.members)
where exists T in s.VALID:T.VTE-T.VTS > 1;
```

The resulting values of this query are

Name	Salary	VALID
Moira	20000	{[94-96)}
Midas	30000	{[96-∞)}

■

So far we have used the time methods only in **select** and **where** clauses. We also would like to support set operations such as **intersect**, **except** and **union** with temporal semantics. In temporal relational algebras, a *temporal intersection* of two sets of tuples is defined as determining during which periods of time a tuple is a member of both sets. In the case of a temporal object algebra, we rather want to calculate the intersection of two sets of objects to find out which objects belonged to both sets for some time period. For example, we might want to find out which employees were members of several staff groups, simultaneously.

To express temporal intersection in *OQL*, we introduce a *generic* function **TO_INTERSECT** which takes two sets of temporal objects and returns a new set of temporal objects:

```
function TO_INTERSECT(
       s1 : set(TempObject),
       s2 : set(TempObject)) : set(TempObject);
```

Example 7 *We assume the staff objects* IS_Staff *and* Math_Staff *with the values given in example 5. To find those employees who are members of both staff groups at the same time, we can now write the query*

```
TO_INTERSECT(IS_Staff.members, Math_Staff.members);
```

which returns a single temporal object with the following values:

Name	Salary	Dept	KEY	VALID
Midas	30000	IS	6	{[96-97)}

Function **TO_INTERSECT** first checks if the classes of the elements of both argument sets **s1** and **s2** are compatible, which means that it checks whether the two element classes are equal or if one class is a subclass of the other. To do this, it has to access meta data of sets **s1** and **s2** and their elements and this is done using the imported Meta Schema provided by O_2. Next those attributes which are common to both element classes are determined again by accessing the meta data. Having found the common attributes, the temporal intersection is rewritten as a join of the two sets with the condition that the values of attributes KEY are equal. This means that we look for objects in both sets which have the same KEY value and thus refer to the same real world entity. The valid time period of the resulting temporal objects is calculated using the method T_INTERSECT. The attribute values of the resulting objects are copies of the values in **s1**.

The query of example 7 is rewritten as

```
select Employee(
         KEY    : e1.KEY,
         Name   : e1.Name,
         Dept   : e1.Dept,
         Salary : e1.Salary,
         VALID  : e1.T_INTERSECT(e2.VALID))
from e1 in IS_Staff.members,
     e2 in Math_Staff.members
where e1.KEY = e2.KEY and
      e1.T_INTERSECT(e2.VALID) != set({});
```

Unfortunately, since functions are not exportable, `TO_INTERSECT` cannot be made available as part of an imported schema. Adding it as a method to class `TempObject` means that we add a method operating on sets of instances of class `TempObject` to each single instance. A more natural way would be to create a new class `SetTempObject` and add methods `TO_INTERSECT`, `TO_EXCEPT` and `TO_UNION` to this class. Functions to calculate temporal union and difference of two sets of temporal objects can be implemented accordingly.

We have seen that it is possible to extend an existing object-oriented database system such that it can do complex temporal queries. It is not necessary to extend a temporal object-oriented query language syntactically to express temporal algebra operations as was done for relational database systems. Temporal algebra operations can be expressed using *non-temporal* algebra operations and functions.

5 Comparing the approaches

5.1 Query Languages

The examples 5, 6 and 7 show that it is quite akward to write temporal queries in O_2 *OQL*. It would be better if the temporal set operations could be written as infix operators rather than as functions with arguments. The application programmer also has to know exactly how the temporal queries can be formulated using the methods of `TempObject`. The temporal queries he writes usually look totally different from their non-temporal counterparts.

We have also seen that it is much easier just to specify that all algebraic operations should be evaluated *temporally* by writing a special keyword in front of a *legal non-temporal query* (see example 2). This is not only less error prone and easier to understand for a programmer, but also helps in migrating non-temporal to temporal queries.

Using the approach described in [21, 20], the query of example 5 could be expressed in *temporal* OQL as

```
valid
 select tuple(Name   : s1.Name,
              Salary : s1.Salary )
 from s1 in IS_Staff.members
 where not exists s2 in IS_Staff.members:
                  s1.Salary < s2.Salary;
```

which clearly is much simpler.

Instead of changing the syntax of the query language, the possibility of overriding algebra operations would help to meet the same goal. Queries, written in a query language like OQL, could be translated into algebraic expressions. If the algebra operations were allowed to be overridden, we could implement a set of temporal algebra operations and use them to override some of the non-temporal operators. The system then would, depending on the context in which the query is executed, execute the temporal operations instead of the overridden ones after translating the query into algebra operations. Thus, we could change the semantics of a query such that it meets the special demands an application such as a temporal database (or others) might have.

5.2 Constraints

Constraints are used to define which states of a database are legal. By adding a time dimension to data, constraint checking needs to be enhanced, too (see e. g. [25, 6, 2]).

The semantics of *referential integrity* must be changed with respect to time. We not only have to check if a referenced object does exist at some point in time, we have to check whether this object exists during the whole timespan it is referenced. In example 3, we have to make sure that a referenced staff object, for example `IS_Staff`, in an object of type `Department`, for example `IS`, actually exists during the time it is referenced.

Special treatment of the *subclass relationship* is also needed in temporal databases. An instance of a subclass may only exist during the time it is also an instance of its superclass. For example, an object of class `Employee` may only exist during the time it is also an object of its superclass `Person`.

Another constraint which might need to be tested depending on how extensively data is timestamped is the *set membership constraint*. This is a constraint similar to the temporal referential integrity constraint. Assume we also timestamp `Staff` objects, which means we also record when a staff group existed. Then an object of type `Employee` may only be a member of an instance of type `Staff` during its own lifespan, and an instance of type `Staff` may only contain objects of type `Employee` during its own lifespan. We call this the *temporal membership constraint*.

Additionally, we can define *partition*, *cover* and *intersection* constraints for sets of objects. They also need to be adapted with respect to time. A *temporal partition constraint* for example needs to check whether two sets of objects never contain the same object at the same time during their lifespan.

Using the database system O_2, we can add methods to objects which check the above (and other)

constraints when the attribute values are updated. This means that the programmer has to provide these general constraints. It would be useful, however, if they were supported by the system itself, since they are general and are considered *model inherent* in most object-oriented data models.

TOM supports this kind of constraints. *Temporal* referential integrity, *temporal* subclass relationship, *temporal* membership, *temporal* partition, cover and intersection constraints are supported by the system directly and checked at commit time.

5.3 Optimisation

Temporal databases should never delete data physically. This soon leads to huge amounts of data which have to be stored and managed. Besides enhancing the expressive power, temporal operations also increase the complexity of query processing. Accessing and querying temporal data thus need to be optimized in order to provide reasonable answer times.

Object-oriented database systems, extended with time functions and temporal comparison predicates, cannot make use of optimisation techniques for temporal data. They simply do not know about the temporal semantics of the data and functions and how they could be exploited for optimisation.

As was shown in the case of temporal relational databases (e. g. [4, 8, 9, 16]), adding temporal data structures and operations to the system allows them to be used in optimisation strategies.

However, instead of supporting specialist database systems handling, for example, temporal or spatial database applications, an approach should be chosen which allows the system to be modified or extended such that special semantics of the application domain can be used for efficient data processing.

In query processing strategies, temporal algebra operations such as the temporal cross product can be used in the same way as the corresponding non-temporal algebra operations. Overriding these operations with different semantics would in most cases still allow the use of the implemented query optimisation algorithm.

6 Conclusions

We have presented two general approaches to implementing temporal databases in object-oriented systems. In both cases, we were aiming at supporting the temporal constructs and operations typical of temporal object data models and, specifically, of our general model TOM. The first approach is to implement a system based on the model, thereby supporting efficient storage and processing of temporal data as well as an ability to handle both temporal and non-temporal data in a uniform manner. The second approach is to extend an existing OODBMS with classes and methods for the management of temporal data. This approach is the only one available in current commercial environments.

We have developed temporal databases using both approaches and found that, while it is much more convenient to develop and query application databases using our own system, we accept that the particular instantiation of the TOM model that we implemented may not be appropriate for all temporal applications. Further, we recognise the need for many different forms of extended database systems to manage spatial data, versions, variants etc.

In conclusion, we advocate an approach that lies somewhere between in that certain basic constructs to support temporal models should be built into the system and then specific instantiations of temporal models should be implemented using the extensible features of the system. In fact, the basic constructs built into the system, such as support for multiple instances of objects, can be used to support not only temporal models, but also version models and other extended forms of database systems. Also, existing systems need to be extensible in a more general sense. In particular, as we have shown, it would be useful to have extensibility of object identifiers and also of the query system. We are currently investigating these issues in the context of the OMS system.

References

[1] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, Volume 16, Number 11, 1983.

[2] 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.

[3] C. S. dos Santos. Design and Implementation of Object-Oriented Views. In *Proceedings of the Database and Expert Systems Applications (DEXA) Conference*, pages 91–102, 1995.

[4] R. Elmasri, G.T.J. Wuu and V. Kouramajian. The Time Index and the Monotonic B^+-tree. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (editors), *Temporal Databases: Theory, Design, and Implementation*, Chapter 18, pages 433–456. Benjamin/Cummings Publishing Company, 1993.

[5] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, Volume 13, Number 4, pages 418–448, 1988.

[6] 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.

[7] W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In *SIGMOD RECORD*, pages 266–275, 1992.

[8] C.P. Kolovson. Indexing Techniques for Historical Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (editors), *Temporal Databases: Theory, Design, and Implementation*, Chapter 17, pages 418–432. Benjamin/Cummings Publishing Company, 1993.

[9] T.Y.C. Leung and R.R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (editors), *Temporal Databases: Theory, Design, and Implementation*, Chapter 14, pages 329–355. Benjamin/Cummings Publishing Company, 1993.

[10] 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.

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

[12] 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*, pages 390–401. Springer-Verlag, LNCS 823, 1993.

[13] M. C. Norrie, A. Steiner, A. Würgler and M. Wunderli. A Model for Classification Structures with Evolution Control. In *Proc. of the 15th Int. Conf. on Conceptual Modelling (ER'96)*, 1996.

[14] O2 Technology. *The O_2 System, Release 4.6; Manuals.*

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

[16] A. Segev. Join Processing and Optimization in Temporal Relational Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass (editors), *Temporal Databases: Theory, Design, and Implementation*, Chapter 15, pages 356–387. Benjamin/Cummings Publishing Company, 1993.

[17] R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, Volume 12, 1987.

[18] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim (editor), *Modern Database Systems*, Chapter 19, pages 386–408. ACM Press, 1995.

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

[20] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. Adding Transaction Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-152r, ISO/IEC JTC1/SC21/WG3 DBL MCI-143*, May 1996.

[21] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. Adding Valid Time to SQL/Temporal. *SQL/Temporal Change Proposal, ANSI X3H2-96-151r1, ISO/IEC JTC1/SC21/WG3 DBL MCI-142*, May 1996.

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

[23] A. Steiner and M. C. Norrie. A Temporal Extension to a Generic Object Data Model. Technical report, ETH Zürich, 1996.

[24] A. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, Volume 11, Number 4, pages 343–355, 1986.

[25] 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.