TOOBIS

Temporal Object Oriented Databases within Information Systems

TODM, Specification and Design

OVERVIEW

MATRA CAP SYSTEMES

O2 Technology

December 1996

Table of Contents

Introduction

This document is an overview of the entire deliverable T31.TR1 dedicated to the specifications and design of the Temporal Object Data Model (TODM) of the TOOBIS project.

The following pages skim through the main points of TODM, and refer to the TODM document for detailed descriptions and explanations.

The objective of the Temporal Object-Oriented Data Model is to be upwards compatible to ODMG-93 standard, more precisely to the version 1.1 of the model of ODMG-93, and to support the main concepts of temporal databases about time varying data.

In the first part, the model of Time used within TODM is described. Then, the different temporal dimensions supported by TODM are presented. The next part details the new types presented in the data model for the manipulation of the model of Time described previously. Thereafter the model for temporal support is introduced. New data types are provided to handle time varying information over the time dimensions described previously. Temporal instances properties and temporal objects are introduced.

Model of Time

TODM is based on a classical and almost standard model of time.

Standard Definitions

This model is a temporal and linear structure where a total order is defined using the 'inferior to' operator. From a theoretical point of view, within the model the time can be continue and isomorphic to the real structure **R**. But of course at the implementation level it will be discrete and isomorphic to the integer structure **Z**.

The definitions of the temporal quantities correspond to the definitions in the consensus of temporal relational databases TSQL2, and also in the drafts of the temporal extension of SQL3, SQL/Temporal.

- an **instant** is an isolated point of the time axis; for example '1996-06-16 17:00:00' represents the 16th of June of 1996 at seventeen o'clock.
- a **period** is an anchored quantity of time between two instants called the boundaries of the period; for example '[1996-06-16 , 1996-07-16)' represents a period from the 16th of June of 1996 to the 16th of July of 1996 not included.
- an **interval** is an unanchored quantity of time, this quantity appears as a time-window along the time axis; for example '3 hours' is an interval of three hours duration.

An instant can be considered as a period of zero duration which only contains this instant.

Boundaries of Time

In the philosophical literature the time is bounded by the «Big Bang» far in the past and by the «Big Crunch» somewhere in the future. TODM will also use two boundaries for the time axis: **beginning** and **forever**. These two specific instants are also commonly used in the temporal databases community.

Calendars & Granularities

The basic international unit of time is the second. A lot of temporal units have been defined in order to locate and measure temporal events. Then calendars have been introduced. A **cal-**

endar is a human abstraction of the physical time line. A calendar is well-defined by its name, its origin, a collection of granules, and a corresponding conversion system between the different units.
The default calendar is the Gregorian one. Its default granularities are year, month, day, hour, minute, and second.

All the previously defined quantities of time are expressed in a specific calendar and at a precise granularity. For instance '1996-06-16 17:00:00' is an instant of the Gregorian calendar expressed at the second-sized granularity which represents the 16th of June of 1996 at seventeen o'clock with zero minute and zero second. Whereas the '1996-06-16' is also an instant from the Gregorian calendar but expressed at the day-sized granularity which represents an instant on the 16th of June of 1996 with no more precision concerning this instant.

On one hand the theoretical conversion of an instant into a finer granularity results in an indeterminate instant where a probability mass function describes the probability for each point that the instant occurred in it. On the other hand, converting an instant to a coarser granularity results in a discard of precision about when the instant occurred.
TODM will not support indeterminate timestamps, and adopts the semantics of the `CAST` operator from TSQL2: the conversion of a timestamp into a finer granularity results into the first value that appears in the theoretical indeterminate timestamp. The two below examples illustrates this policy:

- `CAST(DATE ´1994-06-01´ AS MONTH) = ´1994-06´`
- `CAST(DATE ´1994-06-01´ AS SECOND) = ´1994-06-01 00:00:00´`

Regarding the casting of a period, the beginning boundary is always converted as described above. But for the ending boundary, if the destination granularity is finer than the original one, then the last instant contained in the period is converted to the destination granularity as described above, and then its successor is used as the ending boundary of the casted period.

- `CAST(PERIOD ‘[1996–01, 1996–07)’ AS YEAR) = ‘[1996,1997)’`

Gregorian Calendar

The default calendar of the TOOBIS project is the Gregorian calendar, as resulted from the User-Requirements study of the task T22. In fact the Gregorian calendar is the most used worldwide.
TODM provides a quite complete and useful interface of the Gregorian calendar and of its standard granularities.

- Gregorian granularities: **year**, **month**, **day**, **hour**, **minute** and **second**.
  The different mapping between these granularities are not all regular. For instance from year to month the mapping is regular: there are always 12 Gregorian months in 1 Gregorian year. But irregular mappings also exist as for instance from month to day since there are from 28 to 31 Gregorian days in 1 Gregorian month.
  TODM deals with all these conversions, included the curious and not always well-known management of **leap seconds**. In fact since the creation of the UTC time (Universal Time Coordinated) in the early seventies, some leap seconds has been added to UTC in order

to keep it with one second with the GMT time (Greenwich Mean Time) which corresponds to the Earth rotation time. These seconds occur on specific dates, just before the first January or just before the first July, following the decision of the International Earth Rotation Service (IERS) in Paris. For instance the last leap second has been added on '1995-12-31 23:59:59' and the last minute of the 1995 year contained 61 seconds in order to fill the delay of the Earth.
Thanks to the IERS, a list of all the leap second occurences is available, and TODM will use it in checkiing phase of input and ouput operations.

- Time zones and Daylight Saving Time (DST)
  The world is split in several time zones with delay either positive or negative with regards to the UTC or GMT time. In addition to these time zones, daylight saving times are adopted or not by each country, with preceise rules. Thenin order to manage time zones and DST, TODM has to rely on precise definition rules for them.
  In Solaris Unix System, several data files contain the precise descriptions for the time zone definitions and the DST rules ('/usr/share/lib/zoneinfo'). TODM will use these files or a part of these data to support and check time zone and DST specifications within the instant and period timestamps.

- Fixed and variable size for intervals expressed in months
  Intervals, also called spans, can own fixed or variable size, as for instance the intervals expressed in month-sized granularities in Gregorian calendar. A fixed interval for one month contains thirty one days. But when month intervals are added or substracted to instants, a fixed size is not suitable and leads to errors such as '1996-01-31 + 1 month = 1996-03-02' where the intuitive result is '1996-02-29'. Then thee size of month intervals used with instants depends of the location of this instant. For instance, '1996-01-31 + 1 month' will give as result '1996-02-29'.

For more details regarding the model of time and the Gregorian calendars rules, cf. the whole TODM document, "Model of Time", page 3, a/o cf. the annex of TODM document dedicated to time culture, "Annex 1: Calendars, Clocks, Times …", page 213.

Temporal Dimensions

TODM will support the four time dimensions of the temporal databases. Their definitions given below are extracted from the «Consensus Glossary of Temporal Database Concepts» from the temporal database community.

- the **user-defined time** enables the user to manipulate the quantities of time such as instants, periods and intervals, as any other defined type such as integer.
- the **valid time** is the time when the information was, is or will be true in the world modelled by the database. Entities which deal with valid time are called **historical** entities.
- the **transaction time** allows us to know when the information was current in the database. Entities which deal with transaction time are called **rollback** entities.
- the **bitemporal dimension** is a combination of both valid time and transaction time dimensions. Entities which deal with both valid and transaction times are called **bitemporal** entities.

Valid Time

The valid time timestamps are supplied by the user who can also choose the calendar and the granularity at which they are expressed. If he/she does not, the system uses the current time expressed in the Gregorian calendar at the second-sized granularity as default valid time timestamps.

The histories maintain using valid time can use either instant or period timestamps with no overlapping. But TODM provides also the way to maintain historical entities using overlapping periods; these entities are not really histories but allow to store in a same container data with different semantics such as an average on month and an average on semester.

In order to maintain maximum not contiguous timestamps, TODM supports set of timestamps such as set of non contiguous periods.

Transaction Time

The transaction time timestamps are provided by the system with no intervention of the user. It only uses the Gregorian calendar and periods expressed at the chronon-sized granularity. The transaction time dimension is continue, without any gap.

Non contiguous periods are also used to maintain maximum timestamps.

Bitemporal Dimension

This is an orthogonal combination of the two previous times, where each time dimension keeps the properties described above.

Data Model for Time Manipulation

In this part, new abstract data types are introduced to represent and manage the different notions of time described previously in the “Model of Time” part.

In ODMG few types are described to manipulate time, namely Interval, Date, Time and Timestamp. Unfortunately they are not sufficient and powerful enough for temporal databases and TOOBIS purposes.

Interval

In order to represent unanchored quantities of time which are called intervals, for instance «2 hours and 10 minutes», we add a new data type: **Interval**[1].

The operations on intervals include comparison and arithmetic operations.

- the classical comparison operators are overridden in order to handle intervals
  ==, !=, >, <, >=, <=
- the legal arithmetic operations on intervals are listed in the following table:

operand 1	**operator**	**operand 2**	**result**
	-	interval	interval
	+	interval	interval
	abs	interval	interval
interval	+ or -	interval	interval
interval	* or /	*numeric*	interval
numeric	*	interval	interval
interval	/	interval	*numeric*

NB: ‹numeric› represents a number.
NB: The unary ´+´ is defined for completeness.

1.This name is already used in ODMG-93, in the C++ interface to ODMG, but in order to not complicate the model, we will keep this meaningful name for the new type introduced in TODM

Instant & Period

TODM introduces the two types: **Instant** and **Period**.

Analogously to intervals, the operations applied on anchored quantities of time contain a lot comparison and arithmetic operations.

- the comparison operations include the classical operators (==,!=,<,>,<=,>=) which are overridden for instants and periods, but also some other one which are useful in the temporal databases: `PRECEDES, CONTAINS, MEETS, OVERLAPS, ADJACENT, MERGES`.
- the legal arithmetic operations are listed in the following table:

operand 1	operator	operand 2	result
instant	+ or -	interval	instant
interval	+	instant	instant
period	+ or -	interval	period
interval	+	period	period
instant	-	instant	interval

The set operations are also defined on periods. The theoretical result is not always a period, as for instance the union of two disjoint periods. Then in Period type, the set operations are simplified to only return a period, and TODM introduces the **Period_Set** type which allow to handle sets of non contiguous periods. Comparison operators are also overridden in Period_Set in order to manipulate a set of disjoint periods as a period. Sets of instants are not introduced because an instant can be seen as a period of zero duration with only this instant.

For more details regarding Instant, Period and Period_Set types, cf. the whole TODM document, “Anchored quantity of time: Instant & Period”, page 43.

Relative Time

When a user wants to locate an event on the time dimension, he can use two kinds of temporal references:

- an anchored quantity of time on the time line, called «**absolute dating**»
- a reference to another temporal event, called «**relative dating**».

To deal with relative time, TODM introduces two specialisations of the type Instant defined above: **Absolute_Instant** and **Relative_Instant**.

A relative instant is related to a temporal mark, and delayed by a given quantity of time from it, either in the future or in the past. This quantity of time is in fact an interval. A basic tem-

poral mark can be composed of an instant and of its name. TODM introduces the **Temporal_Mark** type to represent a basic temporal mark. A relative instant will be linked to a temporal mark, with an interval as delay.
A relative timestamp can also be seen as a calculated literal. The calculation is the addition of the interval delay, either in the past or in the future, and of another timestamp. Instead of a temporal mark, we can use a request which represents the way to access the timestamp from which is related the relative instant. The only condition on this request is that it must return a timestamp.
Let us take a small example. A type `Project` is defined with three attributes: `name` is a String; `begin_bound` is an Instant; `end_bound` is a Relative_Instant. An instance is created to represent the TOOBIS project: the `name` is set to 'TOOBIS'; the `end_bound` is initialized with a positive delay of thirty months and it is using the request "`select p.begin_bound from p in Projects where p.name='TOOBIS'`".

Late Binding: NOW

Now has been used by a lot of temporal data models:

- to indicate that a fact is valid until the current time
- to manipulate the current instant

TODM will introduce *now* as a temporal mark: not as a common temporal mark but as a late binding temporal mark. The timestamp instant will only be valuated when the get operation is performed, and will be the current instant.
LateBinding type is a sub-type of Temporal_Mark. **Current** type is a sub-type of LateBinding. One instance of Current type will exist and will be named ***´clock´***. Three specific instances of Relative_Instant type will also exist. They will be named ***´tomorrow´***, ***´now´*** and ***´yesterday´***, and they will be related to the *clock* object respectively with a delay of plus one day, a zero duration delay, and a delay of minus one day.

In the following, simpler notations will be used:

- **`NOW`**: it represents the current instant given by the *clock* object.
- **`NOW()`** : it represents the *now* object, i.e. a relative instant related to the current time with no delay.

The use of relative time to timestamp time-varying information will not be supported.
However to allow the user to express that the data he/she is storing are valid until a change appears, a specific timestamp will be available: UC ("Until Changed"). This timestamp says that the value is valid until a date in the future which not known at the present time, but which corresponds to the next change of value. This timestamp will be allowed as ending boundary of valid time period timestamps.

Calendars & Granules

Any time representation, anchored or unanchored, is expressed in a precise calendar, at a precise granularity. TODM introduces two new data types: **Calendar** and **Granule** types. A calendar is defined by a name, an origin, a collection of granules and the lattice of conversions between these granules. TODM only supports linear calendar. A granule is defined by a name and a partition of another granule. For instance, hour is defined by the partition of a day in twenty four intervals. And the smallest granule of a calendar is only partitioned in chronons which are implementation specific.

Granules

TODM has to provide a casting operation in order to move up or down a time element in the granularity lattice of a calendar. The cast operation is defined in each temporal type:

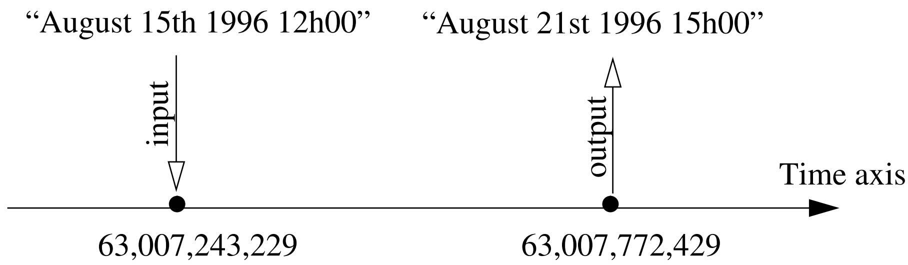
Instant: `cast(into_granule:Granule)->i:Instant`
Period: `cast(into_granule:Granule)->p:Period`
Interval: `cast(into_granule:Granule)->i:Interval`

A calendar is composed of a collection of granules. Each granule is linked to two other granules: the immediately finer one and the immediately coarser one. This represents the lattice of granularities of the calendar. And each granule provides a way to map its value to this coarser or finer granule thanks to `map_to_finer` and `map_to_coarser` defined in the Granule type.

Calendars

The policy adopted by TODM is to store all the time point timestamps in a unique internal representation: the number of chronons since the origin of the calendar in which it is expressed. That allows fast calculations for timestamp operations and simplifies the implementation since only one internal timestamp format is used and its manipulation is calendar independent. That's why, in the previous parts, comparison, arithmetic and manipulation operations of instants, periods and interval are performed without any intervention of the calendar characteristics. However, a timestamp always represents a quantity of time expressed in a given calendar and at a given granularity from this calendar.

As calendars are human abstractions of the Time axis, two crucial operations for each calendar are input and output functions to and from the time axis. The input function is a translation from a constant quantity of time described by a string, to the internal timestamp representation. Conversely, the output function is in charge of the reverse task, i.e. a translation from the internal representation to a temporal constant. The following figure illustrates the input and output operations for the Gregorian calendar.



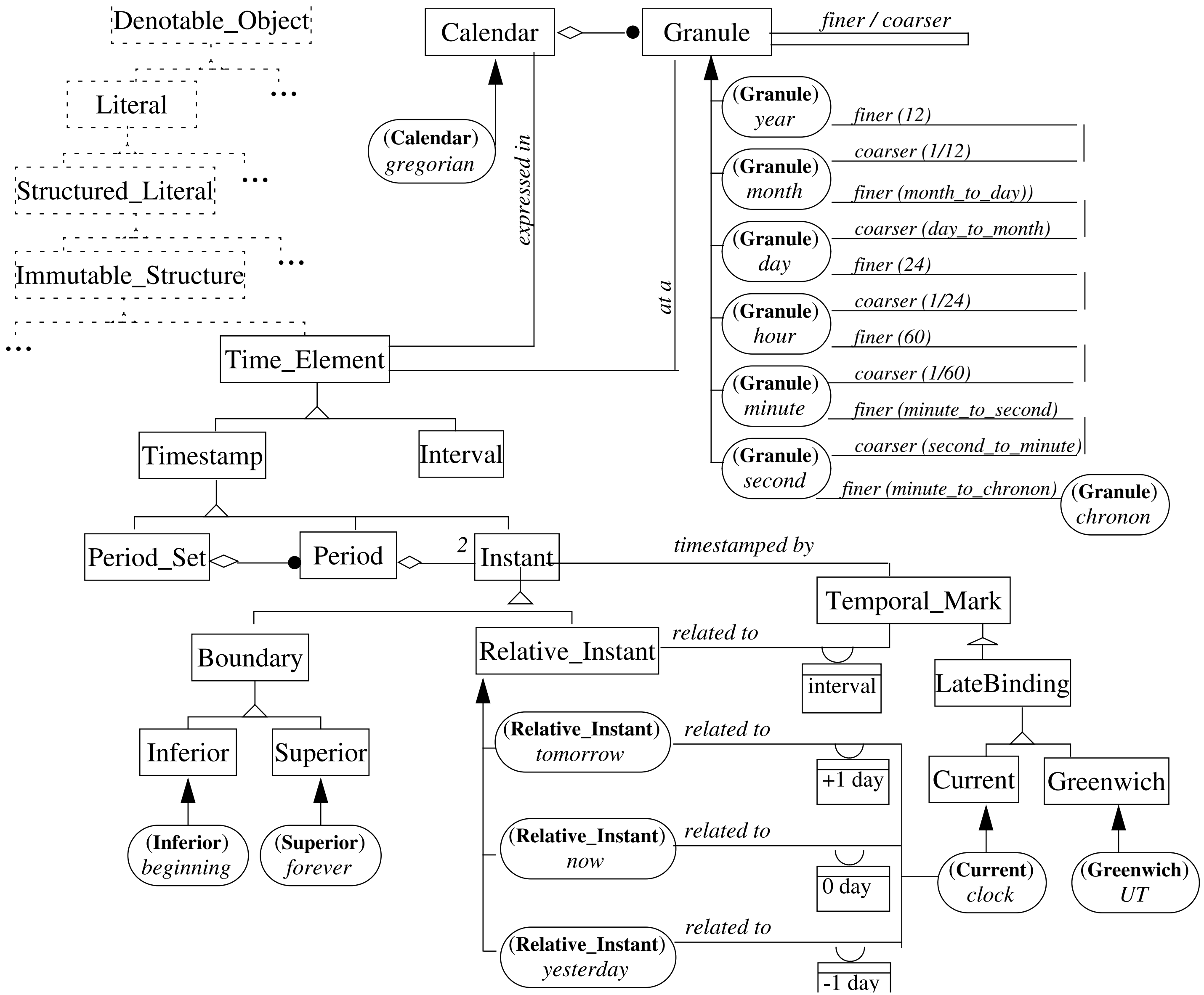
When an instant is expressed, in a given calendar, more information that the number of chronons since the calendar origin can be specified. In fact these pieces of information are useful for the input and output translations, but should not interfere in comparison and arithmetic operations which only know the unique internal timestamp format. They are only helpful for the user interface and timestamp expression. In TODM, they will appear as calendar properties in each temporal type.
For instance, with the Gregorian calendar, the time zone in which is expressed the instant can be specified, as well as the use or not of daylight saving time (DST).

All the operations dealing with variable intervals are calendar dependent operations. The operation described in the previous paragraph dedicated to intervals, are calendar independent with fixed size intervals. But when variable size spans are used, as for instance the addition of a one-month-sized interval to an instant, the operation is calendar dependent and can not be performed without calendar intervention.

Hierarchy of new types

The following figure shows the hierarchy of new types introduced by TODM.

For more details regarding all the new types introduced by TODM for User-Defined Time support, cf. the whole TODM document, "Data Model for Time Manipulation", page 41.


Denotable_Object
Literal
Structured_Literal
Immutable_Structure
...
Time_Element
Calendar
(Calendar) gregorian
expressed in
Granule
finer / coarser
at a
(Granule) year
(Granule) month
(Granule) day
(Granule) hour
(Granule) minute
(Granule) second
(Granule) chronon
finer (12)
coarser (1/12)
finer (month_to_day))
coarser (day_to_month)
finer (24)
coarser (1/24)
finer (60)
coarser (1/60)
finer (minute_to_second)
coarser (second_to_minute)
finer (minute_to_chronon)
Timestamp
Interval
Period_Set
Period
2
Instant
timestamped by
Temporal_Mark
Boundary
Relative_Instant
related to
interval
LateBinding
Inferior
Superior
(Inferior) beginning
(Superior) forever
(Relative_Instant) tomorrow
(Relative_Instant) now
(Relative_Instant) yesterday
+1 day
0 day
-1 day
Current
Greenwich
(Current) clock
(Greenwich) UT

Temporal Support

In ODMG the main concept is the object. An object is defined by a set of instance properties and a set of operations. The value of an object, also called state of the object, is defined by the values it carries for the set of its instance properties.
To introduce the concepts of temporal databases, TODM proposes to define temporal support at the instance properties level and at the object level. Then temporal attributes, temporal relationships and temporal objects will be defined: **historical** types over valid time dimension, **rollback** types over transaction time dimension and **bitemporal** types over both valid time and transaction time dimensions. Temporal objects will maintain histories of the modifications of the states of the objects.

Evolution tracking flag

In the modelled world represented by the database, the change of a value is due to one of the two following reasons:

- The world evolves. Then the value stored in the database which models this true-world, has to evolve too.
- A mistake has been made in the observation of the true-world. Then the stored value has to be corrected to reflect the exact value.

With the use of transaction time, all the database modifications are kept within the database, including the deletion of information. Then in order to be able to distinguish the two above cases of modification of the database, TODM introduces an **evolution tracking flag**. The value of this flag will be *CURRENT/EVOLVED/DELETED*. This flag will be stored with all the variants of temporal types supporting transaction time, i.e. rollback and bitemporal. The user will not have to specify which flag he/she wants to store. The flag will be implicitly stored using operations with different semantics. On the other hand, the user will be able to make a selection regarding the flag value when he/she want to retrieve the information stored in the database.

Temporal attributes

In ODMG-93, two operations are defined on attributes: `set_value` and `get_value`.
TODM override these two operations to take into account the temporal features.

The main operations of the **Historical_Attribute** type, with no overlapping, are listed below.

- `set_value(value:Literal,validtime:Timestamp)`

  The `set_value` operation gives a new value to the attribute. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `set_value` operation is performed to *forever*, is allocated to the attribute by the system. The attribute keeps its old value in its history.
- `delete(validtime:Timestamp)`

  The `delete` operation deletes the information which was, is, or will be valid in the attribute history at the valid time timestamp argument. If the valid time timestamp is not given by the user, then the operation will delete the currently valid information by cutting its timestamp validity.
- `get_value(validtime:Timestamp(*))`
  `->variant:{value:Literal,validtime:Timestamp}`

  The `get_value` operation returns the value of the attribute which was, is or will be valid at the valid time timestamp, and the valid timestamp. If no valid time timestamp is given then the operation returns the value which is valid at present, at the instant the operation is performed.

  (*): if the timestamp is not an instant, several values can be related to the valid time argument, and the result will be a set of `(value,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.
- `get_history()->history:Set<(value:Literal,validtime:Timestamp)>`

  The `get_history` operation returns the whole history of the attribute. The history is represented as a set of couples, where each couple is composed of a value and of its valid time timestamping.

Example

Let us take a small example: a 'Patient' type is created as snapshot object, with a 'weight' attribute as historical attribute using periods expressed in the Gregorian calendar at the month-sized granule. In the following lines, a small scenario of `set_value` and `delete` operations applied to the 'weight' attribute of a `P1` instance of 'Patient' and their consequences on the internal data are described.

- `P1.weight = {}`
- `set_value(60,[96-01,96-04))`
  `P1.weight = { 1:(60,[96-01,96-04)}`
- `set_value(55,[96-03,96-06))`
  ```
  P1.weight = { 1:(60,[96-01,96-03));
                2:(55,[96-03,96-06))}
  ```
- `delete([96-02-96-04))`

```
P1.weight = { 1:(60,[96-01,96-02));
              2:(55,[96-04,96-06))}
```

For more details regarding the historical and other temporal attributes, cf. the whole TODM document, "Temporal Attributes", page 76.

Temporal Relationships

In ODMG-93, relationships are defined as binary relationships between two types. These relationships can be either one to one, one to many or many to many relationships.
TODM introduces temporal support by defining temporal one to one relationships, temporal one to many relationships and temporal many to many relationships. The operations defined in ODMG are overridden to take into account the time dimensions features.

The main operations of the **Rollback_one_to_one** type are listed below.

- `create(o1:Denotable_Object,o2:Denotable_Object)`

  This operation creates a new relationship between the two argument objects. A default transaction time timestamp which is a specific period from the instant when the `create` operation is performed until the next modification, is allocated to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And the evolution tracking flag of the old variant will be set to *EVOLVED*.
- `delete()`

  `correct(o1:Denotable_Object,o2:Denotable_Object)`

  The `delete` operation is equivalent to `correct(o1,NULL)`. The second operation corrects the current relationship with this new one. The new relationship transaction time timestamp is a period from the instant the operation is performed until the next modification. The new variant will own the *CURRENT* flag, and the evolution tracking flag will be set to *DELETED* for the old one.
- `traverse(from:Denotable_Object,transactiontime:Timestamp(*))`

  `->{to:Denotable_Object,transactiontime:Timestamp}`

  `traverse` operation enables to reach the object which was in relationship with the argument object at the transaction time timestamp, with the transaction time timestamp. If no transaction time timestamp is given then the operation traverses the relationship which is currently present, at the instant the operation is performed.

  Only the variants with *CURRENT/EVOLVED* variants flags are retrieved.

(*): if the timestamp is not an instant, several relationships can be related to the transaction time argument, and the result will be a set of `(to,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the second argument.

- ```
  traverse_rollback(from:Denotable_Object)
          ->rollback:Set<(to:Denotable_Object,
                                   transactiontime:Timestamp)>
  ```

  The `traverse_rollback` operation returns the whole audit of the relationship. The audit is represented as a set of couples, where each couple is composed of the object which was in relationship with the argument object and of the transaction time timestamp.

  Only the variants with *CURRENT/EVOLVED* variants flags are retrieved.

- ```
  traverse_flagged_rollback(from:Denotable_Object,
              flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
          ->rollback:Set<(to:Denotable_Object,transactiontime:Timestamp,
                                   flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

  The `traverse_flagged_rollback` operation returns the variants which satisfy the selection on the evolution tracking flag. `NOT_DELETED` represents `CURRENT` or `EVOLVED`; whereas with `ALL` no selection is made on the flag value. The result is a set of triplets, where each triplet is composed of the object which was in relationship with the argument object, of the transaction time timestamp of this relationship and of its *CURRENT/ EVOLVED*/DELETED flag.

Example

Let us take a small example: a 'Patient' type is created as snapshot object, with a 'doctor' relationship towards an 'Investigator' type as a rollback one_to_one relationship. In order to make this example more readable, the transaction time timestamps will be expressed at the day-sized granularity. In the following lines, a small scenario of operations applied to the 'doctor' relationship of a `P1` instance of Patient and their consequences on the internal data are described.

- `P1.doctor = {}`
- ```
  create(P1,I1) on 96-03-01
  P1.doctor = {  1:(OID-P1,OID-I1,[96-03-01,UC),CURRENT)}
  ```
- ```
  create(P1,I2) on 96-04-01
  P1.doctor = {  1:(OID-P1,OID-I1,[96-03-01,96-04-01),EVOLVED);
                 2:(OID-P1,OID-I2,[96-04-01,UC),CURRENT}}
  ```
- ```
  delete() on 96-06-01
  P1.doctor = {  1:(OID-P1,OID-I1,[96-03-01,96-04-01),EVOLVED);
                 2:(OID-P1,OID-I2,[96-04-01,96-06-01),EVOLVED};
                 3:(OID-P1, Ø, [96-06-01,UC),CURRENT)}
  ```

For more details regarding the rollback and other temporal relationships, cf. the whole TODM document, "Temporal Relationships", page 85.
  ```

Temporal Objects

Analogously to the temporal extension of the instance properties, TODM suggests temporal objects. The only difference is that temporal objects handle states of objects instead of literals. In order to make a similarity with the `set_value` and `get_value` operations of the attributes, TODM introduces `set_state` and `get_state` operations.

The main operations of the **Bitemporal_Object** type, with no overlapping on valid time dimension, are listed below.

- `set_state(state:Object_State,validtime:Timestamp)`

  The `set_state` operation gives a new state to the object. The default valid time timestamp is a period from the instant when the operation is performed to *forever*. The transaction time timestamp is a period from the instant when the operation is performed until the next modification. The object keeps its old state in its bitemporal history. The variant with this new information will own the *CURRENT* flag. If some already present variants need to be modified due to this new information arrival, then the evolution tracking flag will be put to *EVOLVED*.

- `delete(validtime:Timestamp)`

  The `delete` operation deletes the information which was, is, or will be valid in the object history at the valid time timestamp argument, and current in the database. If the valid time timestamp is not given by the user, then the operation will delete the currently valid information by cutting its timestamp validity. The *DELETED* flag will appear within the logically deleted variants.

- `get_state(validtime:Timestamp(*),transactiontime:Timestamp(*))`
  `->{state:Object_State,`
  `validtime:Timestamp,transactiontime:Timestamp}`

  The `get_state` operation returns the state of the object which was, is or will be valid at the valid time and was current at the transaction time timestamps, with both the valid and transaction time timestamps. If no valid time timestamp is given then the operation returns the state which is valid at the instant the operation is performed. If no transaction time timestamp is given then the operation returns the state which is current at the instant the operation is performed.

  Only the variants with *CURRENT/EVOLVED* flags can be retrieved.

  (*): if the timestamps are not instants, several states can be related to the valid time and transaction time arguments, and the result will be a set of `(state,validtime,transactiontime)` triplets, where each `validtime` timestamp is contained in the first argument and each `transactiontime` timestamp is contained in the second argument.

- `get_history(transactiontime:Timestamp(*))`

`->history:Set<(state:Object_State,validtime:Timestamp)>`

The `get_history` operation returns the whole history of the object which was current in the database at the transaction time timestamp. The history is represented as a set of couples, where each couple is composed of a state and of its valid time timestamp. If no transaction time timestamp is given then the operation returns the history which is currently present, at the instant the operation is performed.

Only the history containing variants with *CURRENT/EVOLVED* flags can be retrieved.

$^{(*)}$: if the timestamp is not an instant, several states can be related to the transaction time argument, and the result will be a set of `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the argument one.

- `get_flagged_history(transactiontime:Timestamp`$^{(*)}$`,`

```
        flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
    ->history:Set<(state:Object_State,validtime:Timestamp,
        flag:Enum(CURRENT,EVOLVED,DELETED))>
```

The `get_flagged_history` operation returns the variants which was present in the database at the transaction time timestamp, and which satisfy the selection on the evolution tracking flag. `NOT_DELETED` represents `CURRENT` or `EVOLVED`; whereas with `ALL` no selection is made on the flag value. The result is a set of triplets, where each triplet is composed of a state, of its valid time timestamp and of its flag *CURRENT/EVOLVED/DELETED*. If no transaction time timestamp is given then the operation returns the variants which are currently present, at the operation performing date.

$^{(*)}$: if the timestamp is not an instant, several states can be related to the transaction time argument, and the result will be a set of `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the first argument.

- `get_rollback(validtime:Timestamp`$^{(*)}$`)`

```
    ->rollback:Set<(state:Object_State,transactiontime:Timestamp)>
```

The `get_rollback` operation returns the audit of the object regarding the states which was, is or will be valid at the valid time timestamp. The audit is represented as a set of couples, where each couple is composed of a state and of its transaction time timestamp. If no valid time timestamp is given then the operation returns the audit of the values which are valid at present, at the instant the operation is performed.

Only the audit containing variants with *CURRENT/EVOLVED* flags can be retrieved.

$^{(*)}$: if the timestamp is not an instant, several states can be related to the valid time argument, and the result will be a set of `(rollback,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.

- `get_flagged_rollback(validtime:Timestamp`$^{(*)}$`,`

```
        flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
    ->rollback:Set<(state:Object_State,transactiontime:Timestamp,
        flag:Enum(CURRENT,EVOLVED,DELETED))>
```

The `get_flagged_rollback` operation returns the variants regarding the values which was, is or will be valid at the valid time timestamp, and which satisfy the selection on the evolution tracking flag. `NOT_DELETED` represents *CURRENT* or *EVOLVED*; whereas with `ALL` no selection is made on the flag value. The result is a set of triplets, where each triplet is composed of a state, of its transaction time timestamp and of its flag *CURRENT/EVOLVED/DELETED*. If no valid time timestamp is given then the operation returns the audit of the states which are valid at present, at the instant the operation is performed.

(*): if the timestamp is not an instant, several values can be related to the transaction time argument, and the result will be a set of `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the first argument.

- ```
  get_full_history()->rollback:Set<(state:Object_State,
                  validtime:Timestamp,transactiontime:Timestamp)>
  ```

  The `get_full_history` operation returns the whole audit of the object, with regards to both the transaction and the valid time dimensions. But only the variants with the CURRENT/EVOLVED flags will be returned. The whole audit is a set of variants, each variant is composed of a value, of its valid time timestamp and of its transaction time timestamp.

- ```
  get_full_flagged_history(
              flag:Enum:(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
          ->rollback:Set<(state:Object_State,
              validtime:Timestamp,transactiontime:Timestamp,
              flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

  To retrieve the history of the deleted information, or of all the information stored, the `get_full_flagged_history` operation has to be used. The audit is represented a set of variants, each variant is composed of a state, of its valid time timestamp, of its transaction time timestamp and of its flag *CURRENT/EVOLVED/DELETED*.

Example

Let us take a small example: a ‘Patient’ type is created as bitemporal object, using valid time periods expressed in the Gregorian calendar at the month-sized granularity, with a ‘weight’ attribute. In the following lines, a small scenario of `set_state` and `delete` operations applied to a ‘patient’ object and their consequences on the internal data are described.

- ```
  Patient = {}
  ```
- ```
  set_state({(weight,60)},[96-01,96-04)) on 96-03-01
  Patient = {
          1.1:({(weight,60)},[96-01,96-04),[96-03,01-UC),CURRENT)}
  ```
- ```
  set_state({(weight,55)},[96-03,96-06)) on 96-04-01
  Patient = {
          1.1:{(weight,60)},[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
          1.2:{(weight,60)},[96-01,96-03),[96-03-01,UC),CURRENT);
          2.1:({(weight,55)},[96-03,96-06),[96-04-01,UC),CURRENT)}
  ```
  ```

```
• delete([96-02,96-04)) on 96-05-01
  Patient = {
          1.1:{(weight,60)},[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
          1.2:{(weight,60)},[96-01,96-03),[96-03-01,96-05-01),DELETED);
          1.3:{(weight,60)},[96-01,96-02),[96-03-01,UC),CURRENT);
          2.1:({(weight,55)},[96-03,96-06),[96-04-01,96-05-01),DELETED);
          2.2:({(weight,55)},[96-04,96-06),[96-04-01,UC),CURRENT)}
```

Snapshot_Object

Snapshot objects are nearly classical objects from ODMG-93. The state of a snapshot object is defined by the values of a set of instance properties, and its behaviour thanks to a set of operations.
The only difference from the Object type from ODMG-93 is that Snapshot_Object, introduced by TODM, can contain temporal instance properties, i.e. temporal attributes or/and temporal relationships, as well as classical instance properties. It also owns few operations to allow interaction with temporal objects.

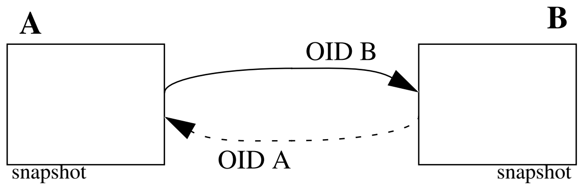
For more details regarding the bitemporal and other temporal objects, cf. the whole TODM document, “Temporal Objects”, page 108.

State Relationships

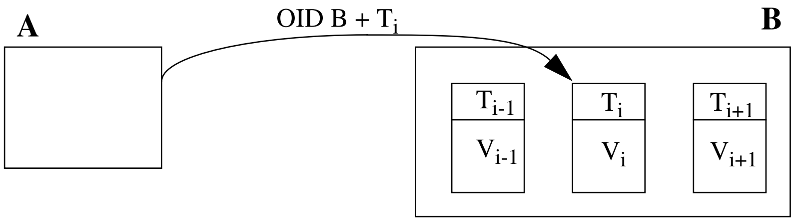
In ODMG, the relationships are defined between two types, and for instance a one to one relationship from an object A to an object B can be represented using a pointer from A towards the OID of B.
By introducing temporal objects which can be seen as collections of states of objects, TODM also introduces a new kind of relationships: **state relationships**. A state relationship does not point towards an object but towards a precise state of a temporal object. Then the OID of the target temporal object is not sufficient to model the state relationship, and it needs to be associated with a kind of Temporal-OID (TOID) which allows to precisely select the state of the temporal object involved in the state relationship.

The following schema illustrates classical relationships versus state relationships.

classical relationship



state relationship



Then TODM defines state relationships in a similar way as are defined classical relationships in ODMG-93. For instance the interface of the one to one state relationship is shown below.

- `create(o1:Denotable_Object,o2:Denotable_Object,TOID:Denotable_Object)`

  It will create a state-relationship from the `o1` object to the state of the `o2` object, precisely described by the temporal key argument.
- `delete()`

  The state-relationship will be deleted.
- `traverse(from:Denotable_Object)->state:Object_State`

  This operation will return the state of the temporal object pointed out by the state-relationship.

And as temporal classical relationships have been defined previously, temporal state relationships are also introduced.

For more details regarding the state relationships, cf. the whole TODM document, “Relationships towards states of temporal objects”, page 136.

Temporal Literals & Temporal Collections

In ODMG-93 are defined a collection of literals and a set of collection types.
TODM proposes a temporal extension of these types, as for instance the **Historical_Integer** type which allows to keep track of the value of an integer over the valid time dimension, or **Rollback_Set<T>** which allows to keep track of the value of a set of T over the transaction

time dimension.

For more details regarding the temporal collections, cf. the whole TODM document, “Temporal structured types”, page 119.

Other Functionalities

Some other functionalities are listed below, linked to the temporal support.

Vacuuming

Temporal data varying over the transaction time dimension face the problem of vacuuming. Deletion of transaction time timestamped information is only logical deletion, and not physical. The transaction time timestamp is cut at the instant of the deletion, and then the information is no more logically currently present, but remains physically present, and is logically present in an earlier state of the database. The vacuuming enables to delete data stored in transaction time audits.

TODM offers vacuuming tools at the object level. Operations, that the user could launch, will perform vacuuming on the object from where they have been launched. Two operations will be available: first to perform a vacuuming thanks to a cut-off point on the time axis; second to perform a vacuuming based on a maximal number of variants in a temporal object. These two operations which has to be added to the rollback and bitemporal types introduced previously, are described above.
The vacuuming at the database level has to be maintained by the application dynamism.

For more details regarding the vacuuming rules, cf. the whole TODM document, “Vacuuming”, page 147.

Logical vs Physical Destruction

For temporal objects, TODM introduces two kinds of destruction or deletion: **logical destruction** and **physical destruction**.

- With physical destruction, objects are physically deleted from the database, i.e. they do not exist any more.
- Regarding the logical destruction, objects are only logically deleted: they are still present in the database but they are not currently retrievable and modifiable.

An object participating in a relationship which keeps track of the evolution of the database, i.e. which deals with transaction time dimension, should not be physically destroyed until this relationship is deleted too. Then it will be logically deleted.

So the destruction of an object can not be always physical. When the object is only logically deleted, i.e. it is involved in some relationships dealing with transaction time, its physical deletion would be effective only at the destruction of these relationships where it participated.

For more details regarding the creation and destruction of objects, cf. the whole TODM document, "Creation & Destruction", page 117.

Conversion from snapshot to temporal

Snapshot objects can contain no time varying instance properties or/and the new temporal instance properties introduced by TODM. Then, it would be convenient, and it appears as a need in the previous task of the project, that the model allows the conversion of a snapshot object into temporal object. These conversions could be useful in some requests of the query language and available using operations defined in Snapshot_Object type.

For more details regarding the conversion of snapshot objects into temporal objects, cf. the whole TODM document, "Conversion from Snapshot into Temporal objects", page 143.

Temporal keys

As well as ODMG-93 defines keys in the type characteristics, TODM introduces temporal keys. These keys represent constraints of uniqueness either at the instance properties or at the object level. They are defined intra or inter time, and act intra or inter object. For instance, with 'inter-object point key' defined on an attribute, two distinct objects can not have the same value in this attribute at any time point.

For more details regarding the constraints of uniqueness, cf. the whole TODM document, "Temporal constraints of uniqueness", page 173.

Example of mapping TODL/TODM

In the following lines, an example is shown to illustrate the kind of mapping between TODL and TODM that should exist.

This example is extracted from the DELTA application described in TODL document.

The declaration in TODL language:

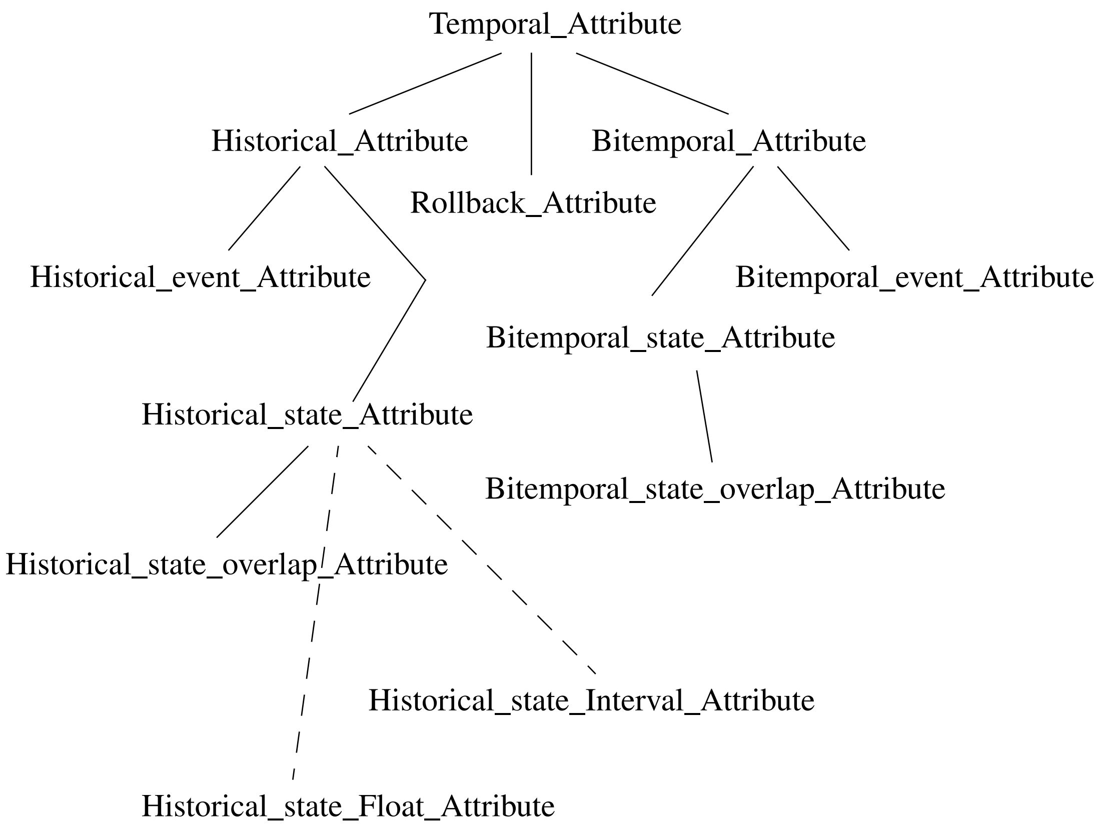
```
interface Product
{
     attribute String productName;
     attribute Float amount valid state granularity day;
     attribute Interval duration valid state granularity day;
...
}
```

will be mapped into TODM data types:

```
class Product
{
     String productName;
     Historical_state_Float_Attribute amount;
     Historical_state_Interval_Attribute duration;
...
     Product()
     { // initialise with precise calendars and granules
          amount = \
               new Historical_state_Flaot_Attribute(‘gregorian’,’day’);
          duration = \
               new Historical_state_Interval_Attribute(‘gregorian’,’day’);
     };
...
}
```

The calendar and granule specification are initialised for each temporal instance property in the constructor of the object. The type of timestamp and the type of value to store are check within each temporal attribute type. For instance Historical_state_Interval_Attribute type will control that the user provide periods for valid timestamping and that he/she make histories of Interval values.

The next schema presents a part of the type hierarchy that will be implemented by TODM. The entire hierarchy will be quite complex, with a lot of temporal types, but with this kind of hierarchy the usage of template will be reduced to a minimum, and the meta-schema will be stored within the database thanks to the type names.


Temporal_Attribute
Historical_Attribute
Rollback_Attribute
Bitemporal_Attribute
Historical_event_Attribute
Bitemporal_event_Attribute
Bitemporal_state_Attribute
Historical_state_Attribute
Bitemporal_state_overlap_Attribute
Historical_state_overlap_Attribute
Historical_state_Interval_Attribute
Historical_state_Float_Attribute

Conclusion

This document is an overview of the deliverable of the task T3.1 of the TOOBIS project, “Specification and Design of Temporal Object Data Model”.

TODM has been conceived in order to be presented to the ODMG board committee, and therefore appears as an extension of the ODM of ODMG-93. The type hierarchy of ODMG-93 model is the base for the TODM new type hierarchy. Temporal features have been introduced in a similar way at the object level and at the attribute level. So implementations within a whole object language or within an object-oriented language are feasible, as the ODMG-93 interfaces with Smalltalk and C++.

The basis theory of time used in TODM gathered the common and standard concepts of Time.
TODM has introduced new abstract data types to handle the instants, periods, intervals, calendars and granularities time notions. These quantities of time can be used within the user-defined time, and aid for the storage of time varying information.

To support temporal databases concepts, TODM has provided temporal instance properties and temporal objects. Historical types vary over the valid time dimension. Rollback types vary over the transaction time dimension. Bitemporal types vary over both the valid time and transaction time dimensions. Snapshot types which do not vary over time, has been modified for a better interaction with temporal types.

Regarding the catalogue issued of the user-requirement task (T2.2), TODM meets the needs and necessary tools required for manipulation of time varying information.

The work accomplished in this task has been done with exchanging collaboration with the tasks T3.2 and T3.3, “Specification and Design of Temporal Object Definition Language & Temporal Object Query Language”, which strengthens the validity and usability of the structures introduced in TODM.