TOOBIS

Temporal Object Oriented Databases within Information Systems

TODM, Specification and Design

Deliverable T31TR.1

MATRA CAP SYSTEMES

O2 Technology

December 1996

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This document is the deliverable of the task T3.1 of the TOOBIS project.

The objective is the definition of a Temporal Object-Oriented Data Model to be upwards compatible to ODMG-93 standard and to support the concepts of temporal databases.

In the first part, the model of Time used within TODM is described. This includes the different quantities of time used to measure, to handle and to be located on the Time axis.

Then, the different temporal dimensions supported by TODM are presented, followed by a small state of the art of temporal extensions to relational and non-relational models which enables us to precisely define the scope of TODM.

The fifth chapter details the new types presented in the data model for the manipulation of the model of Time described previously. Quantities of time and calendar manipulations are set out, with a stress on the Gregorian calendar which is the default one used in TOOBIS.

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.

The following chapter is dedicated to the design and the implementation of temporal support. Some examples of mapping between TODL and TODM are presented.

In the end, a chapter is dedicated to a small study of compliance with the user-requirement catalogue from the previous task of TOOBIS project. This shows that TODM meets the needs of the user-requirement consolidation task.

To complete this document, an annex gives some additional information related to Time culture.

CHAPTER 2

Model of Time

In this chapter, we will describe a data model for Time.

The ideas and definitions that will be explained here, are the standards and the common notions of the temporal database community.

This data model for Time is only a formal model of time. It does not move towards the Temporal Object-Oriented Data Model of the TOOBIS project, but defines the basis and the theory about Time modelling that will be followed.

Basis Theory of the Time Model

Temporal Structure

In the temporal database community, we can find three time models which have been used as basis for all the researches about time: the continuous model, the dense model, and the discrete model.

- In the continuous model, time is isomorphic to real numbers: each real number is a point on the time line.
- In the dense model, time is isomorphic to rational numbers: each rational number is a point on the time line.
- In the discrete model, time is isomorphic to integer numbers: each integer number is a point on the time line.

Those three different models share a same hypothesis: the time line is an axis.

A temporal structure is a set of points with an asymmetric and transitive relation of precedence. The relation «less than» (´<´) in the integer set **Z** is a temporal structure; the same holds for this relation in the rational set **Q**, and in the real set **R**. Then the time space is projected on a time line, with the relation of precedence using the axis direction.

A temporal structure is called **linear** when the relation of precedence is satisfied for any two points chosen in the temporal structure. Therefore the model of Time is linear only if it supports one branch of time.
Z, **Q**, or **R** with ´<´ relation form a linear temporal structure (total order).

Versus linear temporal structure, there exists branching time. With branching time you could have several different pasts, called postlinear or right linear; several different futures, called antelinear or left linear. The ´<´ relation can only determine partial order, since it is only well defined over one branch of time.
In the temporal database community, only linear models of time have appeared. Branching time is used in reasoning systems, where hypothesis calculations are often needed.

In our model the branching time will not be supported. The three main reasons are the following ones:

- Branching time does not appear as a need for the TOOBIS applications, from the user requirements analysis.
- TSQL2, the consensus extension of SQL2 from the temporal database community, and SQL3/Temporal, drafts of the future SQL3 standard dealing with temporal extension, do not support branching time. Then in a compatibility way, we will not support branching time too.
- Branching time appears as a very specific domain of interest in which only a small percentage of users seems concerned.

Instant & Period

An isolated point on the time axis is called an **instant**. The time between two instants is called a **period**. A period is precisely and only described by a couple of instants: its beginning boundary, and its ending boundary, with the constraint that the former one has to precede the latter. All the instants contained between these two boundaries belong to the period. An instant can be seen as a period of zero duration, where its beginning and ending boundaries are equal.

The choice among the three basic models, namely continuous, dense and discrete, is not crucial at the level of temporal database ontology. The definitions of instants and periods are the same, on continuous, dense or discrete time axis. On the other hand, since the time is modelled on a discrete computing device, the view of time has to be discrete. So the user will manipulate a discrete view of a time axis that can be continuous, dense or discrete.

Interval

An unanchored quantity of time is called an **interval**. The difference between an interval and a period is that a period is well-located on the time axis, whereas an interval is only a time-window that can be moved along the time axis. An interval is not defined by the quantity of time between two boundaries but only by a quantity of time.

Boundaries of Time

On the conceptual level, the Time is infinite, in the past as well as in the future. On a more

practical hand, the Time will be stored into some data structures, which are limited in size. Therefore the infinity of Time would be impractical at the storage level.
Moreover, in the literature the «Big Bang», 14±4 billion years ago, and the «Big Crunch», sometime in the future, are well-known as the boundaries of Time.

Two special instants enable us to escape to these boundaries: **beginning** and **forever**. Those two instants are located at the exterior of the boundaries: beginning will be smaller than any instant, and forever will be greater than any instant.

A discrete view of Time

Now we will speak about the time segment instead of the time line, because of the boundaries. Let us call this time segment T. We can divide T into a finite number of smaller segments called **granules**. Each set of granules is well ordered, since each granule is associated to a segment of T and the ordered relation on T implies an ordered relation on the set of granules. This partitioning is called **scaling**. It gives a discrete view of the time segment. This view is dynamic since it only appears at the execution time of the requests. For instance, a day-sized scaling partitions the time segment into day-sized granules.
The smallest scale is composed of chronons. The **chronon** is the smallest granule. The size of the chronon is specific to the implementation.

When the user chooses an appropriate scaling, he/she can see the time segment as a discrete set of seconds, or of days, or of years …
The discrete view of the Time is created at the execution time of requests. But, if you look at an instant with different granularities, the view of the information you will get can be different. For instance, the quantity of time contained within the first day of 1970 can be seen as the instant ‹1970-01-01› through a day-sized scaling, whereas it can be seen as the period ‹1970-01-01 00:00:00 , 1970-01-01 23:59:59:› through a second-sized scaling.

Let us introduce two terms often used in time research: **temporal domains** and **temporal universes**.
A temporal universe, TU, is a finite succession of temporal domains, $<T_0, T_1, \dots T_n>$. Two temporal domains, T_i and T_j, where $i \neq j$, are disjunct. Each temporal domain T_i is a partitioning in segments of the temporal domain T_{i+1}. Therefore the segments of the temporal domain T_0 are the chronons.

For instance <seconds, minutes, hours, days, months, years> is a temporal universe. In this universe it is impossible to insert the weeks, since we can not represent a week which overlaps two months or two years.

The discrete view, manipulated by users, is a discrete linear temporal structure, in the same fashion with (**Z**,´<´) one.

Modelling Instants

The smallest quantity of time that can be represented in an instant timestamp is a chronon. But a chronon is a segment of time, whereas an instant is an isolated point on the time segment.

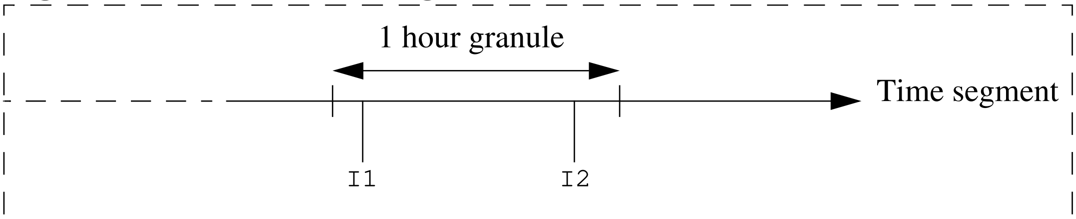
In fact, since we did not choose an explicit model of time, we can assume a continuous model. Then the chronons are bigger than the instants, since the continuous model implies that there is an infinite number of instants into each chronon.

NB: If we explicitly choose the discrete model, the size of an instant has to be equal to the size of a chronon.

Then an instant timestamp stores the information that this event has occurred sometime in a particular chronon or a particular granule. We can assume that the instant timestamp represents any instant in the granule where it is located. Then the exact instant modelled by an instant timestamp, is never well-known. At the best, only the granule where it is located, is known.

However, two timestamps which are represented by a same granule, may model two different instants. For instance, two instants located in the same hour-sized granule, can represent two different instants as far away as fifty nine minutes. The two instants are represented by the same instant timestamp, with hour-sized granularity. Cf. Figure 1, “Instants in the same granule”, page 7.

Figure 1. Instants in the same granule



The same timestamp, for I1 and I2 instants, with hour-sized granularity.

There exist some other hypotheses about the localization of the instants regarding to chronons and granules. Here are the two most common hypotheses, but they are far from the reality.

- We may assume that the instant timestamp represents the first instant of the granule where it is located. Unfortunately, when an instant is represented by ‹20h30min›, this only means that we do not know more than that is located sometime during the minute described by ‹20h30min›, and not precisely the first instant of the 30th minutes after 20h.
- We may also assume that the instant timestamp represents the whole chronon or the whole granule where it is located. But an instant has a zero duration, whereas a granule has a non-zero duration. For instance, the instant timestamp ‹20h30min› does not mean that it begins at the 30th minute after 20h, and that it ends just before the 31st minute.

So, an instant is modelled by an instant timestamp, with an associated time granularity. The instant represented by this timestamp is an instant from a granule, and this granule is determined by the scaling.
For example, the instant timestamp ‹January 1st 1996›, with a day-sized granularity, represents one instant during the first day of the January month of year 1996.

Modelling Periods

The timestamp of a period owns an associated granularity. This timestamp is composed of two instant timestamps, and a constraint. This constraint forces the instant timestamp of the beginning boundary of the period, to precede or to be equal to the instant timestamp of the ending boundary. The two instant timestamps have the same granularity.

For instance, ‹[January 1st 1996, April 1st 1996)› is a period timestamp, and represents a period from the first day of January of 1996 to the last day of March of 1996. The ending boundary is not included in the period.
Let us have a look at another example: the period timestamp ‹[January 1st 1996 , January 2nd 1996)› represents a period somewhere during the day of January 1st 1996. But with this particular example, we could not be more precise about the instant where it starts and where it ends, because the two instant timestamps are equal at the day-sized granularity, and the period has a zero duration length. In fact this peculiar period can be concidered as the January 1st 1996 instant.

Calendars & Granularities

Calendars

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

There exist two ways of defining a calendar:

- the descending method, where you define a smaller unit thanks to the division of the basic unit. For instance, one minute is the division of one hour in sixty segments.
- the ascending method, where a greater unit is defined by the aggregation of several smaller units. For instance, one day is the aggregation of twenty four hours. This is the way to build temporal domains and temporal universes.

This second method will be followed in the TODM support of calendars. The smallest granule of each calendar has to be expressed with regards to the chronon which is implementation specific.

Conversion to finer or coarser granularities

Each calendar has to provide the way to move from one of its granularities to another, i.e. a quantity of time expressed with a particular granule could be converted to a smaller, finer, or greater, coarser granularity. The mapping between the different granules is called the granularity lattice.

On one hand, converting an instant to a finer granularity results in an indeterminacy since that transforms the instant into an indeterminate timestamp where we do not know precisely when the instant occurred. For instance, «January 1st 1996» which represents the first day of 1996 at the day-sized granularity, is converted into the «January 1st 1996 00h ~ January 1st 1996 23h» indeterminate instant at the hour-sized granularity, 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. For instance, «January 1st 1996» which represents the first day of 1996 at the day-sized granularity, is converted into the «January 1996» instant at the month-sized granularity which represents the first month of 1996.

In TSQL2, two operations are provided to convert timestamps into finer or coarser granularity timestamps. First, `scale` allows to move a timestamp up or down the granularity lattice. Moving a timestamp up the lattice produces a determinate timestamp, but moving a timestamp down the lattice produces an indeterminate timestamp. Second, `cast` is similar to `scale` except that it only delivers determinate timestamps. When converting the granularity of a timestamp into a finer one, `cast` returns the first possibility of the indeterminate timestamp that `scale` would return. The following figure, Figure 2, “SCALE & CAST TSQL2 Operations”, page 10, gives examples of `scale` and `cast` operations from TSQL2 consensus.

Figure 2. SCALE & CAST TSQL2 Operations

```
SCALE(DATE ´1994-06-01´ AS YEAR) = ´1994´
SCALE(DATE ´1994-06-01´ AS MONTH) = ´1994-06´
SCALE(DATE ´1994-06-01´ AS DAY) = ´1994-06-01´
SCALE(DATE ´1994-06-01´ AS HOUR) =
                    ´1994-06-01 00 ~ 1994-06-01 23´
SCALE(DATE ´1994-06-01´ AS MINUTE) =
                    ´1994-06-01 00:00 ~ 1994-06-01 23:59´
SCALE(DATE ´1994-06-01´ AS SECOND) =
                    ´1994-06-01 00:00:00 ~ 1994-06-01 23:59:59´

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

An indeterminate instant is described by a lower support (boundary), an upper support, and a probability mass function (*p.m.f.*). The manipulation of such timestamps is described in a special chapter of TSQL2 as a possible extension. About SQL/Temporal, the actual drafts do not deal with temporal indeterminacy. And in the road map of additions to SQL/Temporal, temporal indeterminacy appear as described in TSQL2.

In TODM, we will not support temporal indeterminacy. TODM adopts the semantics of the cast operation: the conversion of a timestamp into a finer granularity results into the first value that appears in the theoretical indeterminate timestamp.

Regarding the casting of a period, the beginning boundary is always converted as described above. But for the ending boundary, the rule for casting depends on whether the destination granularity is finer or coarser than the original one.

- if the destination granularity is finer than the original one, then the ending boundary is converted as described above
- if the destination is coarser 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. For instance the period '[1996-01, 1996-07), casted at the year-sized granularity, will result into the period '[1996,1997)'.

Gregorian Calendar

The most familiar and worldwide used calendar is the Gregorian one. The Gregorian calendar is based on the rotation of the Earth on its axis and its revolution around the Sun. For more theoretical and historical details, see the "Calendars and Timestamping", page 214, in the Annex.

The origin used in SQL-92 is «January 1st 1 A.D., midnight».

The default granularities are second, minute, hour, day, month, year.

The relationships between the granularities are classified in regular mappings and irregular mappings.

- Regular mappings

  - one Gregorian year includes 12 Gregorian months.

  - one Gregorian day includes 24 Gregorian hours.

  - one Gregorian hour includes 60 Gregorian minutes.
- Irregular mappings

  - one Gregorian month includes 28 to 31 Gregorian days, depending on the month name and if the month is located in a leap year or not. Cf. Table 1, "Table of Gregorian months", page 12.

  - one Gregorian minute includes 58 to 62 Gregorian seconds, depending on weather -2, -1, 0, +1, +2 leap seconds have been added.

Leap seconds are due to the variation in the Earth´s rotation with regards to the International Atomic Time: the Earth runs slower at roughly two milliseconds per day. Then after about five hundred days, the accumulation reaches near one second. Leap seconds are added just before midnight on dates selected in advance by the International Earth Rotation Service (IERS), in order to keep the Coordinated Universal Time (UTC) within one second of the Greenwich Mean Time (GMT). For instance, the latest leap second occurred at the end of 1995 year: 1 leap second has been added on December 31st 1995 at 23h59min59sec, enabling the Earth to catch up with its delay. Then the last minute of the

1995 year contained 61 seconds and the run was: 1995-12-31 23:59:58, 1995-12-31 23:59:59, 1995-12-31 23:59:60, 1996-01-01 00:00:00.

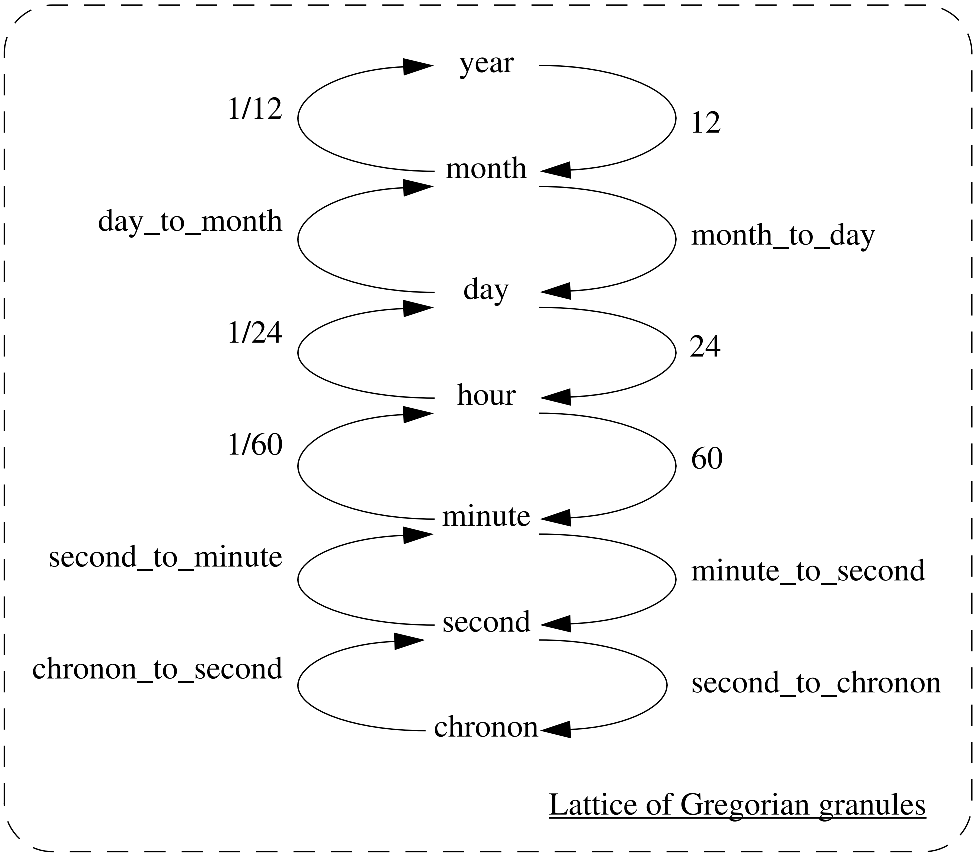
For more details regarding the different times and the leap seconds, cf. “International clocks and leap seconds”, page 221, in annex. In this chapter are also listed all the leap seconds which have occurred since 1972 when was adopted the U.T.C., along with a copy of the IERS Bulletin C announcing the leap second occurence described above.

Table 1. Table of Gregorian months

Month	Duration in days
January	31
February	• 29, if the year is a multiple of 400, or a multiple of 4 and not of 100 - leap year • 28, else
March	31
April	30
May	31
June	30
July	31
August	31
September	30
October	31
November	30
December	31

With the mapping between the different granules, we can define the lattice of Gregorian granules, as shown in the Figure 3, “Network of Gregorian granule mapping”, page 13. Irregular mappings are represented using functions with a name inferred from the source and target granularities, and regular mappings using the number of target granules within the source one. The mappings dealing with chronons are implementation specific.

Figure 3. Network of Gregorian granule mapping



Fixed and variable intervals

Another point to explore with calendars is the use of spans, or intervals. A fixed interval represents a fixed duration. On the other hand, a variable interval is a span whose duration varies in relation to an associated event. As for instance, within the Gregorian calendar, month-sized intervals can be variable.

The fixed span for one month is thirty one days, as used in TimeDB demonstrator of SQL/Temporal drafts.

But fixed size span is not always suitable for month operations because of the irregular mapping between months and days.
Let us assume that we get an instant, defined at a day-sized granularity or smaller. Then we can notice three semantics for the result of the addition of one month-long interval to this instant:

- the same day in the next month

- For instance: ‹1996-05-03› + ‹1› month = ‹1996-06-03›
- But what about: ‹1996-05-31› + ‹1› month = ?

• the same number of days from the beginning of the month
- For instance: ‹1996-05-30› + ‹1› month = ‹1996-06-30›
- But what about: ‹1996-05-31› + ‹1› month = ?

• the same number of days from the end of the month
- For instance: ‹1996-05-31› + ‹1› month = ‹1996-06-30›
- But what about: ‹1996-05-01› + ‹1› month = ?

We will follow the characteristics of the MultiCal project, which is mentioned in TSQL2 as a concrete realization of supporting calendars. The solution is a combination of the two last semantics. If the instant is located in the first half of a month then the operation will be performed using the number of days from the beginning of the month. Conversely, if the instant is located in the second half of a month, then the operation will be performed using the number of days from the end of the month. The middle of the month is chosen at midnight of the sixteenth day, except for February where the middle is at midnight of the fifteenth day.
Then the previous unresolved operations have now the following results:

- ‹1996-05-31› + ‹1› month = ‹1996-06-30›
- ‹1996-05-01› + ‹1› month = ‹1996-06-01›

Regarding fractional month span addition, the operation is performed in two steps. First the addition of the integer part is computed, following the previous rules. Second, the fractional part is calculated in the result month. For instance, «1996-05-10» + «1,5» month, gives as result «1996-06-10» + «0.5» month, and finally «1996-06-25» since June contains 30 days and 30*0.5=15. If we get more days than there are in the destination month, we convert the excess to a fraction of the first month and apply this fraction to the next month. For instance, «1996-06-21» + «0.5» month, gives as result «1996-07-01» + «1/6» month since (30*0.5=15, 15-10=5 and 5/30=1/6), and finally «1996-07-06 04h04min48sec».

Leap seconds are used in variable interval computations only after the destination month and day are determined. The final time is computed by multiplying the percentage of day left by the number of second in that day, including any leap seconds. With this policy the addition of an integer number of days within a month will result in the same time as the starting day.

Time Zones & Daylight Saving Times

Below a precise example of time rules for France country is described: definition of the Europe/Paris zone, and rules for daylight saving time.

The following tables are extracted from the ‘europe’ time zone information V1.7 file of Solaris Unix system.

The time zone definition table is composed of five columns:

- NAME: the name of the time zone.
- GMTOFF: the amount of time to add to GMT to get standard time in this zone.
- RULES: the name of the rule(s) that apply in this time zone for local standard time.
- FORMAT: the format for time zone abreviation in this time zone.
- UNTIL: the time at which the GMT offset or the rule(s) change.

Table 2. Time Rules for Europe/Paris Zone

Zone	NAME	GMTOFF	RULES	FORMAT	[UNTIL]
Zone	Europe/Paris	0:09:05	-	LMT[a]	1891 Mar 15 0:01
		0:09:05	-	PMT[b]	1911 Mar 11
		0:00	France	WET[c]%s	1940 Jun 14
		1:00	M-Eur	MET[d]%s	1944 Aug 25
		0:00	France	WET%s	1945 Sep 16 3:00
		1:00	France	MET%s	1977 Apr Sun>=1 2:00
		1:00	M-Eur	MET%s	

a. LMT: Local Mean Time

b. PMT: Paris Mean Time

c. WET: Western Europe Time

d. MET: Middle Europe Time

From this first table, and more precisely from its last two rows, we can retrieve the following information: since the first Sunday of April 1977 at 2:00, the Europe/Paris zone is +1 hour off from GMT, it follows the M-Eur rules and uses the MET format. In fact this table gives the rule of the local ‘standard’ time, and refers to the different rules for the local ‘wall clock’ time.

The M-Eur rules are listed in the following table, Table 3, “Daylight Saving Time Rules for Middle Europe”, page 16.

The local rules definition table is composed of nine columns:

- NAME: the arbitrary name of the set of rules.
- FROM: the first year in which the rule applies.
- TO: the final year in which the rule applies.
- TYPE: the type of year in which the rule applies ('-': all the years between FROM and TO, …).
- IN: the month in which the rule takes effect.
- ON: the day of the month on which the rule takes effect.
- AT: the time of the day at which the rule takes effect.
- SAVE: the amount of time to be added to local standard time when the rule takes effect.
- LETTER/S: the 'variable part' zone abbreviation to be used when this rule is in effect.

Table 3. Daylight Saving Time Rules for Middle Europe

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	M-Eur	1800	only	-	Jan	1	0:00	0	-
Rule	M-Eur	1916	only	-	Apr	30	23:00	1:00	" DST"
Rule	M-Eur	1916	only	-	Oct	1	1:00	0	-
Rule	M-Eur	1917	1918	-	Apr	Mon>=15	2:00s	1:00	" DST"
Rule	M-Eur	1917	1918	-	Sep	Mon>=15	2:00s	0	-
Rule	M-Eur	1940	only	-	Apr	1	2:00s	1:00	" DST"
Rule	M-Eur	1940	only	-	Dec	31	2:00s	0	-
Rule	M-Eur	1941	only	-	Feb	25	2:00s	1:00	" DST"
Rule	M-Eur	1941	only	-	Oct	5	2:00s	0	-
Rule	M-Eur	1942	only	-	Jan	1	2:00s	1:00	" DST"
Rule	M-Eur	1942	only	-	Nov	2	2:00s	0	-
Rule	M-Eur	1943	only	-	Mar	29	2:00s	1:00	" DST"
Rule	M-Eur	1943	only	-	Oct	4	2:00s	0	-
Rule	M-Eur	1944	only	-	Apr	3	2:00s	1:00	" DST"
Rule	M-Eur	1944	only	-	Oct	2	2:00s	0	-
Rule	M-Eur	1977	1980	-	Apr	Sun>=1	2:00s	1:00	" DST"
Rule	M-Eur	1977	only	-	Sep	lastSun	2:00s	0	-
Rule	M-Eur	1978	only	-	Oct	1	2:00s	0	-
Rule	M-Eur	1979	1995	-	Sep	lastSun	2:00s	0	-
Rule	M-Eur	1981	max	-	Mar	lastSun	2:00s	1:00	" DST"
Rule	M-Eur	1996	max	-	Oct	lastSun	2:00s	0	-

The two shaded rows enable us to know the precise daylight saving time rules for 1995 year for instance.
With the second one we know that: on the last Sunday of March 1995 (1995-03-26), at 2:00 local ‘standard’ time, i.e. at 2:00 local ‘wall clock’ time, an amount of one hour has to be added to the local “standard” time and “ DST” string has to be appended to the ‘standard’ output format. Therefore the local ‘wall clock’ time becomes “3:00 DST”.
The first shaded row allows to say that: on the last Sunday of September 1995 (1995-09-24), at 2:00 local ‘standard’ time, i.e. 3:00 local ‘wall clock’ time since DST is used, nothing has to be added to the local ‘standard’ time and the standard output format has to be used. Therefore the local ‘wall clock’ time becomes “2:00”.

TODM has to take into account the above specifications for the Gregorian calendar support. Then the user must be able to state precisely in which time zone his timestamp is expressed, and if he uses DST or not.

If the time zone is not specified, the current time zone of the system is used.

With DST, some problems have to be faced:

- A timestamp can be ambiguous:
  ex: “1995-09-24 1:30”
  We can not say if it is before or after the time change due to the end of the DST usage: …, 1:58 DST, 1h59 DST, 1:00, 1:01, …
- A timestamp can be non-existent:
  ex: “1995-03-26 2:30”
  This timestamp does not exist since the start of DST usage created an one hour-sized gap: …, 1:58, 1h59, 3:00 DST, 3:01 DST, …

To solve this problem, the following policy, drawn from the policy of the C library dedicated to time from Solaris, has been adopted in TODM:

- if the DST specification is absent, then an attempt of translation of the timestamp to a non-DST time is made; if this fails, then one hour is added and the timestamp will be converted to a DST time.

- if the DST specification is present, then the first attempt will be a translation to DST time; if this fails, then one hour is substracted and the timestamp will be converted to non-DST time.

- if the conversion leads to an instant located in the gap due to the start of DST usage, then an hour is substracted for the non-DST time conversion and an hour is added for the DST time conversion.

Several examples will illustrate this rule.

- “1995-09-24 1:30 MET”

  - the DST specification is absent, then a translation to non-DST time is attempted. It is successful, the timestamp is “1995-09-24 2:30 MET” or “1995-09-24 0:30:00 UTC”.

- “1995-09-24 1:30 MET DST”

  - the DST specification is present, then a translation to DST time is attempted. It is successful, the timestamp is “1995-09-24 1:30 MET DST” or “1995-09-23 23:30:00 UTC”.

- “1995-03-26 2:30 MET”

  - the DST specification is absent, then a translation to non-DST time is attempted. Unfortunately this timestamp is located in the gap due to the start of DST usage, then this fails. One hour is added and the timestamp is converted in DST time The timestamp is now “1995-03-26 3:30 MET DST” or “1995-03-26 1:30:00 UTC”.

- “1995-03-26 2:30 MET DST”

  - the DST specification is present, then a translation to DST time is attempted. Unfortunately this timestamp is located in the gap due to the start of DST usage, then this fails. One hour is substracted and the timestamp is converted in non-DST time. The timestamp is now “1995-03-26 1:30 MET” or “1995-03-26 0:30:00 UTC”.

- “1995-06-16 2:30 MET”

  - the DST specification is absent, then a translation to non-DST time is attempted. It fails since DST time is used in June 1995 in MET zone. One hour is added and the conversion to DST time is successfully performed. The timestamp is now “1995-06-16 3:30 MET DST” or “1995-06-16 1:30:00 UTC”.

- “1995-06-16 2:30 MET DST”

  - the DST specification is present, then a translation to DST time is attempted. It is successful, the timestamp is “1995-06-16 2:30 MET DST” or “1995-09-24 0:30:00 UTC”.

- “1995-12-25 2:30 MET”

  - the DST specification is absent, then a translation to non-DST time is attempted. It is successful, the timestamp is “1995-12-25 2:30 MET” or “1995-12-25 1:30:00 UTC”.

- “1995-12-25 2:30 MET DST”

  - the DST specification is present, then a translation to DST time is attempted. It fails since DST time is not used in December 1995 in MET zone. One hour is substracted and the conversion to non-DST time is successfully performed. The timestamp is now “1995-12-25 1:30 MET” or “1995-12-25 0:30:00 UTC”.

The list of the commonly used time zone abbreviations, and more details regarding DST rules, as Greece for instance, are described in the annex “Time Zones & Daylight Saving Time (DST)”, page 226. One can notice that the DST specification can be retrieved from the time zone specification, as for instance MEST (Middle Europe Summer Time).

Timestamp operations

Associating a granularity to the time segment creates a discrete view of this time segment. However the time segment could be continuous, dense or discrete.
The timestamp operations are defined with respect to this discrete view of the time segment. For example, a comparison operation carried out at the day-sized granularity will compare some days; a duration calculation operation carried out at the second-sized granularity will return a duration in seconds.

As we saw before, an instant-sized granularity can not exist. The smallest granularity is composed by chronons, which are segments of time. Therefore the timestamp operations can not be carried out at the level of the instants. For instance, we could not answer to a question such as «Does an instant precede another instant, in an only instant point of view?». It is impossible to reply precisely since an instant timestamp does not store the precise location of the event, but only the granule where is it located. Of course, we could answer precisely to the question «Does an instant precede another instant, in an instant timestamp point of view?», since here we compare two instant timestamps.

Granularities in operations

Inside a temporal domain, the timestamp operations are carried out at the granularity of this domain. On the other hand, in order to extend these operations to the whole temporal universe, i.e. to operate with timestamps from different temporal domains, it is necessary to be able to manipulate one of the two timestamps in the granularity of the other one. This is possible thanks to the lattice of the calendar granules in which the timestamps are expressed.
Then we have to consider three possible semantics, about the choice of the granularity at which the operation will be carried out.

- The operation can be performed at the granularity of the first or left operand. That is be done in SQL-92. The main disadvantage is that symmetric operations are no longer symmetric.

- The finer operand granularity can be chosen, i.e. the coarser granularity timestamp will be plunged into the finer one. The main problem is that converting a timestamp to a smaller granularity results in an indeterminacy, as described before in this chapter.
- The coarser operand granularity can be chosen, i.e. the finer granularity timestamp will be plunged into the coarser one. This solution avoids indeterminacy, but some information of the finer granularity is discarded.

TSQL2 adopted the semantics that the operations are performed at the granularity of the left operand, in order to remain consistent with SQL-92.
SQL/Temporal should require the granularities to be identical. This is a simpler way but also a more compelling one.

Finer operand granularity semantics owns also the inconveniency that explicit casts are needed when operations at a coarser level are required.

As default semantics, TODM will adopt the policy of TSQL2 which is more flexible with regards to the user. So operations will be performed at the granularity of the left operand. Of course, the user can overload this behaviour by casting the operands to his wanted granularities.

The only exception to this rule will occur when an unanchored quantity of time will be added to or substracted from an anchored quantity of time. In this case the operation will be directly performed without any casting. Then operations such as "'1996-01-31' + '1' Month", will return correct result, such as '1996-02-29', and not erroneous solutions such as '1996-03-02' if the month-sized interval was casted to 31 days.

Summary

Users handle a discrete view of the Time model which can be either continuous, dense or discrete. The time axis is bounded.

Instants are isolated points located on the time axis. Periods are anchored quantities of time, whereas intervals are unanchored.
Timestamps are used to represent these time entities. Each timestamp is associated to one calendar and to one granularity. A granularity is a partitioning of the time axis in granules. A calendar is a human abstraction of the time axis. It is composed of a set of granules with precise mapping functions between them. The Gregorian calendar uses the second, minute, hour, day, month and year granules.

Timestamp operations are defined with respect to the discrete view of time. They are carried out at the left operand granularity. The theoretical conversion of a timestamp expressed in a particular granularity into a finer granularity leads to an indeterminate timestamp. TODM will not support indeterminacy, and chooses to return the first possibility of the indeterminate timestamp.

CHAPTER 3

Temporal Dimensions

Temporal databases deal with two time dimensions: the **valid time** and the **transaction time**. This chapter describes these two time dimensions and the resulted typology in a temporal database.

Introduction

There exist three main concepts in temporal database: the **user-defined time**, the **valid time** and the **transaction time**. Here are the definitions of these concepts, given in the «Consensus Glossary of Temporal Database Concepts» of the temporal databases community.

- The user-defined time is an uninterpreted attribute domain of date and time.
- The valid time of a fact is the time when the fact is true in the modelled reality.
- The transaction time of a database fact is the time when the fact is current in the database and may be retrieved.

Then, with these three dimensions, four types of database appear:

- snapshot or classical database, where only the last state of the fact is stored in the database, i.e. only a snapshot of the real world is kept.
- historical or valid time database, where all the valid states of the fact are stored in the database, i.e. the entire history of the real world is kept.
- rollback or transaction time database, where all the modifications of a fact in the database are stored, the entire audit-trail of the database is kept.
- bitemporal or valid and transaction time database, where all the states and modifications of the state are stored in the database, i.e. both the entire real world history and the entire audit trail of the database are kept.

User-defined Time

The user-defined time enables the manipulation of temporal concepts described in the previous chapter, «Model of Time».

Time is an attribute of most real-world phenomena, and then data representing temporal information appeared in databases many years ago. But their only way to deal with these information was to use the user-defined time dimension. As for instance, the birthday of a person is usually modelled with a date attribute.

The manipulation of the user-defined time dimension is the same as any attribute.

In order to support user-defined time, the DBMS has to provide the temporal attribute domain, like instant, interval, period, calendar, and also to provide tools to deal with them.

Valid Time

Adding valid time support, comes to the timestamping of facts with the time when they were, are or will be true in the modelled reality. No restriction is applied on these times. The user can supply past, present or future timestamps. If the user does not supply any valid time timestamp, a default one is provided by the system from the time the operation is performed to the specific instant forever.

There exist two kinds of timestamping: timestamping with instants or timestamping with periods. A valid time point timestamp represents the instant when the fact has been measured or evaluated in the real-world. Valid time period timestamping gives a notion of duration, and is used when the system tracks precisely the evolutions of the real-world.
The precision of the valid time timestamps can be chosen by the user. If a month precision is sufficient or if he/she needs a more accurate timestamping expressed at the day precision, then the user has to declare it.

In TSQL2 consensus from the temporal databases community, event and state characteristics are introduced. In an event relation, each tuple is timestamped with an instant set, whereas in a state relation, each tuple is timestamped with a period set also called temporal element. TSQL2 uses sets of instants or sets of maximal non-contiguous periods to record the time during which a fact was, is or will be valid.
On the other hand, in the SQL3/Temporal drafts (from May 1996), only periods are used in valid time timestamping. And each timestamp contains only one period, and not a set. This stays in 1NF and it is easier to implement. However on the road map of additions to SQL3/Temporal, event tables appear in the list of next change proposals.

From the requirements analysis of the previous task, time points and periods are needed for timestamping.

Therefore our TODM will support valid time instant timestamping and valid time period timestamping. Sets of instants and sets of maximal non-contiguous periods may be used to represent valid timestamps. May be in a more implementation care, this use of set of timestamps will allow space saving, avoiding the duplication of identical values with different valid time timestamps. The choice between periods or set of periods will have to be precise at the implementation level. In this model, both are supported and can be used.

Points and periods will be expressed at a valid time precision which is user-defined.

Transaction Time

Adding transaction time support comes to the timestamping of the database facts with the time when they were current and retrievable. This time is bounded by the lifetime of the database. It can not be earlier than the creation of the database, and can not be later than the present time. The transaction time is supplied by the system, without any intervention from the user, neither on the value, nor on the precision of transaction time. This transaction time timestamp states that the value is current since the time the operation was performed, and will be current until a new modification appears, but with no information regarding the exact future time of the modification which does not exist over the transaction time dimension.

In TSQL2, as well as in SQL3/Temporal drafts, only periods are used for transaction time timestamping. A transaction time period represents the lifespan of the database fact, from its insertion in the database to its logical deletion.
In TSQL2, temporal elements, i.e. sets of non-contigous periods, are used as transaction time timestamps whereas in SQL3/Temporal each timestamp is one period.

Therefore our TODM will support transaction time period timestamping. As with valid time timestamping, the model will deal with both periods and sets of periods transaction time timestamping. Each timestamp will be expressed at the transaction time precision which is implementation defined.

Transaction time timestamping and DBMS transactions

Adding transaction time dimension support should not challenge the properties and integrity of DBMS transactions. DBMS transactions obey to the ACID rules. These rules are recalled below.
ACID: Atomic Coherency Isolation Durability

- Atomic: a transaction succeeds in its whole or not at all; there can not exist piece of transactions.
- Consistency: if a DBMS satisfies a set of coherency constraints before a transaction, then the transaction should release the DBMS in a state where all the same coherency constraints are satisfied.

- Isolation: When different transactions run concurrently, a transaction could not see the effects of another transaction which has not yet been committed.
- Durability: when a transaction has been validated, its effects are permanent in the database.

For the timestamping of database information with transaction time dimension, several times can appear as possible candidates: the beginning time of the transaction, the modification time of each object, the locking object time of each object, the freeing lock object time, the ending time of the transaction … But an important feature has to be taken into account: all the objects modified within a same database transaction should own the same transaction time timestamp. In this case times that are different for each object are not a good solution. The two other solutions, beginning time and ending time of the transaction, seem to meet this requirement. But in the following lines, the usage of beginning time will be invalidated since it can lead to some incoherency. Ending time of transaction will be used in TODM for transaction time timestamping.

First let us take a small example (#1). A list actually contains three elements: `L={a,b,c}`. Two operations are defined using `L`: `count(L)` which returns the number of elements in `L`, and `add(L,d)` which adds another element into `L`. The first operation is only a read operation on L, whereas the second one is a write operation. Two transactions are also defined. `t1` can be modelled as the following run (`begin(t1); count(L); end(t1)`), and `t2` as (`begin(t2);add(L,d);end(t2)`).

Taking into account the two different locks, reading lock and writing lock which are used in the two phases locks mechanism[1], the different possible cases of concurrently runs for t1 and t2 are listed in the Table 4, “Example #1 of concurrent transactions”, page 28. The first column represents the execution of the transaction `t1`, and the other six columns are the different cases of execution of transaction `t2` with regards to `t1`. The locks are freed only at the end of transactions, which might be either commit or rollback end. That’s why the operation of `t2` can not be inserted between the counting operation and the end of transaction `t1`.

If we choose the beginning time of the transaction as for transaction time timestamping:

- when the two transactions are not really concurrent but `t2` runs before or after `t1`, as the cases (1) and (2) of our example: then there is no problem using the beginning time of transaction.
- when the two transactions are really concurrent and the order of the access operations, i.e. the order of lock acquisitions, is the same as the order of the begin operations, as the cases (4) and (5) of our example: then there is no problem to use the beginning time of transaction since the transactions end in the same order as they began.

1. first phase: acquisition of locks
   second phase: as soon as one lock is freed, acquisition of new locks is impossible.

- when the order of the access operations, i.e. the order of lock acquisitions, is different than the order of the begin operations, as the cases (3) and 6) of our example: the beginning time of transaction can not be used since the transactions end in a different order from their begin operations. That may lead to some incoherency such as `L={a,b,c,d}` at $t2_{begin}$ and `#count=3` at $t1_{begin}$ with $t1_{begin} > t2_{begin}$, which is the sixth case of our example.

Table 4. Example #1 of concurrent transactions

(0)	(1)	(2)	(3)	(4)	(5)	(6)
	begin(t2)			begin(t2)		begin(t2)
	add(L,d)					
	end(t2)					
begin(t1)						
			begin(t2)		begin(t2)	
			add(L,d)			
			end(t2)	add(L,d)		
count(L)				end(t2)		
end(t1)						
		begin(t2)				
		add(L,d)			add(L,d)	add(L,d)
		end(t2)			end(t2)	end(t2)

When choosing the ending time of transaction, we do not face such problems as described above. The ending operation of transactions corresponds to the freeing of the locks. Then using this ending time we are sure that there does not exist another transaction which has also worked with the same objects since they were locked. All the cases of our example are valid if we use the ending time of the transactions for the transaction time timestamping.
The only problem is that the ending time of a transaction is not known until the transaction ends.

Since in its body a transaction is isolated, i.e. other transactions can not see what it is doing, then we can use a value which is not coherent regarding the whole database but which allows this transaction to be coherent in its body.

The beginning time of the transaction is the instinctive and simple timestamp to use. But we can face some other problems with long database transactions. Let us take a second example (#2). For instance, two transactions `t3` and `t4` run: in `t4` an object `o2` is modified

and is given `t2(end)` transaction time timestamp; `t3` modifies an object `o1`, compares the transaction time timestamps of `o1` and `o2`, and modifies an object `o3` if `o2` transaction time timestamp is greater than `o1` one (`TT(o2)>TT(o1)`). If `t4` takes place after the beginning of `t3`, and before the comparison test, as in Table 5, "Example #2 of concurrent transactions", page 29, then there will be a problem if we use beginning time of transaction as transaction time timestamp. In `t3` the test will be `(TT(o2)=t4(end))>((TT(o1)=t3(begin))` and then `o3` will be modified. But at the end of `t3`, `o1` and `o3` transaction time timestamps will be changed to `t3(end)`. And now one has `(TT(o2)=t4(end))<((TT(o1)=t3(end))`, therefore `o3` should not be modified.
A best solution is to use a specific value which in the transaction always returns the current time when queried, and at the end of the transaction will be replaced by the ending time of the transaction. Then the above problem will not exist since the test within `t3` will be `(TT(o2)=t4(end))<((TT(o1)='NOW')` since the current time is greater than the transaction time of `o2` otherwise it could not be accessible because `t4` has not yet finished.

Table 5. Example #2 of concurrent transactions

transaction t3	transaction t4
begin(t3)	
	begin(t4)
	modification of o2
	end(t4)
modification of o1	
if TT(o2)>TT(o1) then modification of o3 else {}	
end(t3)	

Therefore a special value will be used for transaction time timestamping within the body of the transaction, which will always return the current time when queried by the transaction. At the end of the transaction the ending time of the transaction will be used to replace all the transaction time timestamps previously modified by the transaction. The objects can be handled in the body of the transaction and when it will be finished the consistency is extended to the whole database by using the ending time of the transaction.

Bitemporal

Adding bitemporal support is equivalent to timestamp with both valid and transaction times.

Since valid and transaction times are orthogonal, all the characteristics of both valid and transaction time features are supported by the bitemporal data.

Then our TODM will deal with bitemporal data as the cohabitation of valid and transaction time dimensions. A bitemporal timestamp is a couple composed of a valid time and a transaction time timestamps, and to maintain maximum non contiguous timestamps, bitemporal timestamps could be sets of couples, each couple composed of a valid time timestamp set and a transaction time timestamp set.

Summary

The three temporal dimensions, namely **user-defined time**, **valid time** and **transaction time**, will be supported by our TODBMS.

With the user-defined time dimension, the user can deal with all the common concepts of time, like instant, interval, period, calendar and so on.

On one hand, the valid time dimension is supported thanks to instant or period timestamping, which can be supplied by the user.

On the other hand, the transaction time dimension is supported thanks to period timestamping, which is supplied by the system.

And finally, the bitemporal dimension is a combination of both valid time and transaction time dimensions.

The model can support timestamping thanks to sets of instants or set of periods in order to maintain maximal non contiguous quantities of time. The other possibility is to deal only with periods or instants, which is simpler. The choice will have to be decided at the implementation level, between a space saving orientation or a simpler one.

CHAPTER 4

Object & Time

This chapter presents a small state of the art regarding some temporal extensions of the relational model and of some other models. It precises the scope of the TODM specifications which will be an extension of the ODMG-93 model.

Introduction

There are two main strategies in the Object Models:

- whole object models, like Smalltalk.

  In Smalltalk, the object is the only entity, and message invocation is the only way to monitor and communicate with objects.
- object oriented models, like C++.

  In C++, objects can be composed of literals, as for instance *float* or *char*.

The model of ODMG-93 differentiates objects from attributes and relationships which make up the instance properties. Attributes are literals, and relationships are links between objects. Attributes and relationships own operations to handle them. For instance a *set_value* and a *get_value* operations are defined for attribute.
Then the model of ODMG can be interfaced easily with object oriented models as well as whole object models. In the ODMG-93 book, the C++ interface and the Smalltalk interface are described to support this feature.

The two following parts are dedicated to brief descriptions of some temporal extensions of relational models and of non-relational ones. Attribute vs tuple, and attribute vs object temporal supports are explored.

Then, some recommendations will be explained so that the TODM of TOOBIS should act as an extension of the ODMG-93 model.

Temporal semantics & Relational model

The literature about temporal extensions to the relational model proposes adding temporal semantics through tuple or attribute versioning. Some of these relational extensions deal only with valid time dimension or transaction time dimension, and others deal with both dimensions.

In the two-parts table, "Temporal extensions to the relational model", page 34 and page 35, several temporal extensions to the relational model are summarized. This is not an exhaustive list, but it includes the work done by several members of the TSQL2 Language Design Committee. In this table the name of the model, its author, the time dimensions over which it is defined, the type of timestamping, either attribute or tuple, the temporal domains of the timestamps, and few information about the implementation way are described.

As we can see, a vast majority of these relational extensions defined the timestamping at the tuple level which enables to keep 1NF relations. But there exists one extension, the "Historical Relational Data Model", which proposes to associate timestamps with both tuples and attributes.

Table 6. Temporal extensions to the relational model

Data Model	Author	Time Dimensions	Attribute / Tuple timestamping
Temporally Oriented Data Model	Ariav	valid time	tuple
Time Relational Model	Ben-Zvi	valid time transaction time	tuple
Historical Relational Data Model	Clifford	valid time	attribute tuple
TempSQL	Gadia	valid time transaction time	attribute
DM/T	Jensen	transaction	tuple
LEGOL 2.0	Jones	valid time	tuple
Interval Extended Relational Model (IXRM)	Lorentzos	valid time	attribute (set of attributes)
Temporal Relational Model (TRM)	Navathe	valid time (transaction time)	tuple
HSQL	Sarda	valid time	tuple
TQuel	Snodgrass	valid time transaction time	tuple
Postgres	Stonebraker	transaction time	tuple
HQuel	Tansel	valid time	attribute
TSQL2		valid time transaction time	tuple
SQL3 (SQL/Temporal Drafts)		valid time transaction time	tuple

Table 7. Temporal extensions to the relational model (continued)

Data Model	Temporal Domains	Implementation Way
Temporally Oriented Data Model		
Time Relational Model	instant, period, period set	5 supp. attributes: Tes/Tee (effective), Trs/Tre (registration), Td (deletion)
Historical Relational Data Model	*lifespan* (period)	attribute = value + lifespan 1 supp. att: Lifespan
TempSQL	temporal element (set of periods)	attribute = value + temporal element
DM/T	instant	2 supp. attributes: Time, Operation (Ins., Del., Mod.)
LEGOL 2.0	chronon	2 supp. attributes: Start, Stop
Interval Extended Relational Model (IXRM)	*interval* (period)	4 supp. attributes to represent validity of other(s) attribute(s)
Temporal Relational Model (TRM)	instant	2 supp. attributes: Ts, Te
HSQL	interval (period), instant	1 supp. attribute
TQuel	instant	4 supp. attributes: Vb, Ve, Tb, Te
Postgres	instant	2 supp. attributes
HQuel	period	attribute = value(s) + timestamp
TSQL2	instant, period, temporal element	addition of attributes
SQL3 (SQL/Temporal Drafts)	period	addition of attributes

Temporal semantics & Object-Oriented models

As for the temporal extensions of the relational model, we can find several temporal extensions of non-relational models, like entity-relationship and object-oriented models. They are more recent than the relational extension because of the relative youth of the object-oriented models.

In the following lines some of the most known models are described.

Temporally Extended Entity-Relationship (TEER)

Elmasri, Wuu & Kouramajian [Elm]

The Temporal data model for EER (TEER) extends the EER model to include temporal information on entities, relationships and attributes. EER is not really an object-oriented model, but it was one of the first non-relational model which has been extended.

Lifespan of an entity or a relationship instance is represented by a temporal element, which could be a period or a set of disjoint periods.

Temporal attribute values are defined to be functions from the time domain to the attribute domain of an entity. To an entity e of type E, composed of a list of attributes (A_i) is associated its lifespan with a temporal element $T(e)$. The temporal value of each attribute is given by a function $A_i(e)\colon T(e) \rightarrow dom(A_i)$.

Each relationship instance, $r = \langle e_i \rangle$, is associated with a temporal element $T(r)$ which represents its lifespan. This temporal element has to be a subset of the intersection of the temporal elements of all the entities that participate in the relation: $T(r) \subseteq (T(e_1) \cap T(e_2) \ldots)$. And the TEER deals with the relationship attributes as it does with entity attributes: $A_i(r)\colon T(r) \rightarrow dom(A_i)$.

Temporal extension of OODAPLEX

Wuu & Dayal [Wuu]

OODAPLEX is an object-oriented data model based on the DAPLEX functional data model. Real-world entities are modelled thanks to objects, and functions are used to model properties, relationships and operations of objects.

Time points, and time point sets are treated as abstract objects.
The time-varying properties or behaviours of the objects are modelled by functions that relate time objects to snapshot values. This flexibility allows to version temporal information at both attribute level and object level.
This results into a great advantage of the query language: no special time-oriented construct are needed since the model deals with time-varying information thanks to functions as it does with non-temporal information. Therefore non-temporal and temporal complex objects are handled in a uniform way.

Temporal Data Model based on Time Sequences

Segev & Shoshani [Seg.a]

The Time Sequence (TS) concept is basically the sequence of values in the time domain for a single entity instance, where all values are ordered in time.

A temporal datum is defined as $<s,t,a>$, where s is the invariant identifier of the object, t is the time, and a is the attribute value. Then a TS is defined as $<s,(t,a)^*>$, i.e. an indentifier and a sequence of pairs *(t,a)*.
The collection of TSs for objects of a same type are called Time Sequence Collection (TSC).
With TSCs, the model deals with temporal data, and also with temporal relationships.

Temporal Object-Oriented Data Model (TOODM), Algebra (TOOA) and Query Language (TOOSQL)

Ellen Rose & Arie Segev [Ros93a]

A type encapsulates the structure (attributes), behaviours (operations) and constraints of an object. Each object has a unique object identifier (OID). And the state of an object is described by the values of its attributes.

The time-sequence collection type is used to model histories of attribute values of objects and histories of the set of properties which define an object type. Each time sequence object has the history and the correction history properties each of which contains a sequence of *(A,TL)* pairs. *A* represents attribute values and *TL* represents temporal elements which are finite unions of time-lines.

Therefore the temporal semantic which is located at the attribute level, can also be seen at the object level where in fact each attribute carries its own history and the history of the object is built by slicing them.

However this work started at the end of eighties could not base itself on object and object database standards like ODMG. Still it remains as a research result contrary to the more practical and implementation-oriented goals of the TOOBIS project.

TOOBIS TODM

In order to be presented as an extension to the ODMG-93 model, the TODM of TOOBIS needs to be coherent and compliant to the ODMG concepts.

TODM should also deal with the concepts of the temporal databases, and introduce them in the object databases world.

TODM will provide temporal features at both the object and attribute level. The interfaces of these two supports will be as similar as possible. This similarity will enable an implementation in object oriented as well as in whole object languages.

Schema evolution/versioning

In order to remind the distinction between schema modification, evolution and versioning, the following definitions are given, extracted from the «Consensus Glossary of Temporal Database Concepts» of the temporal database community:

- **schema modification**: schema modification is accommodated when a database system allows changes to the schema definition of a populated database.
- **schema evolution**: schema evolution is accommodated when a database system permits the modification of the database schema without loss of existing data.
- **schema versioning**: schema versioning is accommodated when a database system allows the access of all data, both retrospectively and prospectively, through user definable version interfaces.

Today, nothing is normalized about schema evolution and versioning.
The following lines describe the present position of the standards and consensus of interested for the TOOBIS project.

Regarding ODMG-93, versioning is introduced in the paragraph dedicated to the next revisions of ODMG standard. Versioning of objects and versioning of types are listed in object candidate functionality for the next release.

In the TSQL2 document, a specific chapter is dedicated to schema evolution and versioning. A number of aspects of schema evolution and versioning are indicated as candidate for inclusion in query languages and particularly in TSQL2. But only few of them have been proposed for inclusion in TSQL2, others were considered too complex for the aim of the TSQL2 language.

About SQL3, the present SQL/Temporal drafts plan a later progression dedicated to schema evolution/versioning. But this will be proposed as discussed in TSQL2. Today the current transaction time change proposal states that schema modification on tables with transaction time support requires versioning of the schema base, and refers to the later progression.

If we think to the next task of the project, i.e. the implementation of TODM as extension of O2 DBMS, the characteristics of O2 regarding schema evolution/versioning have to be taken into account. O2 DMBS allows schema evolution but not schema versioning. The schema manager maintains the history of the evolution of class definitions, but this history is not open to the user and is strictly O2 DBMS engine information for the conversion of objects. This conversion of objects to the current schema can be either deferred when the application accesses them or immediate for all the base if required. O2 DBMS also supplies a complete interface for the versioning of objects, but this is different from schema versioning.

Then, the policy adopted by TODM is to not provide schema evolution/versioning facilities since today nothing is normalized and precisely described in standards. Moreover, schema evolution/versioning does not appear as a need in the User Requirement analysis of the previous TOOBIS task (T2.2), used as input information for the current specification task. TODM will restrict to data evolution and not schema. However, since TODM will be implemented as a library to O2 DBMS, the schema evolution mechanism of O2 will be present. The point we have to take care is that schema evolution is not allowed on types with transaction time support. The characteristics of transaction time support are to maintain the audit of all the modifications occurred in the database. Then the schema evolution will need to be also historised and this is schema versioning which is not supported.

A full schema versioning in temporal databases appears as an interesting study which needs a sustained dedicated work. Unfortunately, this work can not be done within the TOOBIS project. Moreover, the implementation of such a concept could not be done as a library extension of a DBMS but needs at least to modify the DBMS engine; and this is not one of the targets of TOOBIS.

CHAPTER 5

Data Model for Time Manipulation

In this chapter, we will describe a data model for Time manipulation.

We will introduce new abstract data types to represent and manage the different notions of time that we presented in the previous chapter "Model of Time".

These new time types will be the basis for the manipulation of time for the Temporal Object-Oriented Data Model of the TOOBIS project.

Introduction

In the Object Model of the ODMG-93, there is no part dealing with temporal information. Only few types are described in the C++ interface to ODMG-93. Namely, these types are Interval, Date, Time and TimeStamp. Unfortunately, these are not sufficient and they do not provide all the desired functionalities to express and manipulate the time features we need to represent and handle Time in a temporal database.

NB: In the drafts of ODMG 2.0, Date, Interval, Time and Timestamp types are directly introduced with the model (ODM) of ODMG. These types are defined as in the ANSI SQL specifications, but their interfaces are not sufficient for the needs of a temporal database.

Then, TODM will define some new abstract data types that support the temporal model of the previous chapter"Model of Time", page 3. These types will be introduced as new structured literals.

Unanchored quantity of time: 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**.

NB: As seen before, this type is already defined in ODMG, but not in a satisfying way to temporal databases. For example, it is not complete in regards to arithmetic operations on unanchored quantities of time. In order to not complicate the model, we will keep this name, «Interval», for our model. More over, at the implementation level in O2 system, there will be no problem since the type supplied by O2 is *d_Interval*.

An interval defines an unanchored duration of time, that is an amount of time with known length but no specified starting neither ending point.

We can divide up the operations on intervals in two main categories: comparison operations and arithmetic operations.

About the comparison of intervals, there are only three possibilities. Let us assume that I and J are two intervals, then we can face:

- I is smaller than J (smaller than or equal to): I<J (I<=J)
- I is equal to J (different from): I==I (I!=J)
- I is greater than J (greater than or equal to): I>J (I>=J)

Then these six operations have to be defined for the Interval type.

The following table, Table 8, “Legal Arithmetic Operations on Intervals”, page 43, shows the arithmetic operations we can apply to intervals. In this table, only the legal operations are described. Then there is no orthogonality since some operations, like the multiplication of two intervals, have no meaning.

Table 8. Legal Arithmetic Operations on Intervals

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.

Anchored quantity of time: Instant & Period

Points on the time line are called instants. Thus TODM will define a new data type to represent a time point: **Instant**.

An anchored quantity of time, well-defined by its beginning and the ending instant, is called period. Thus TODM will define a new data type to represent these quantities of time: **Period**.

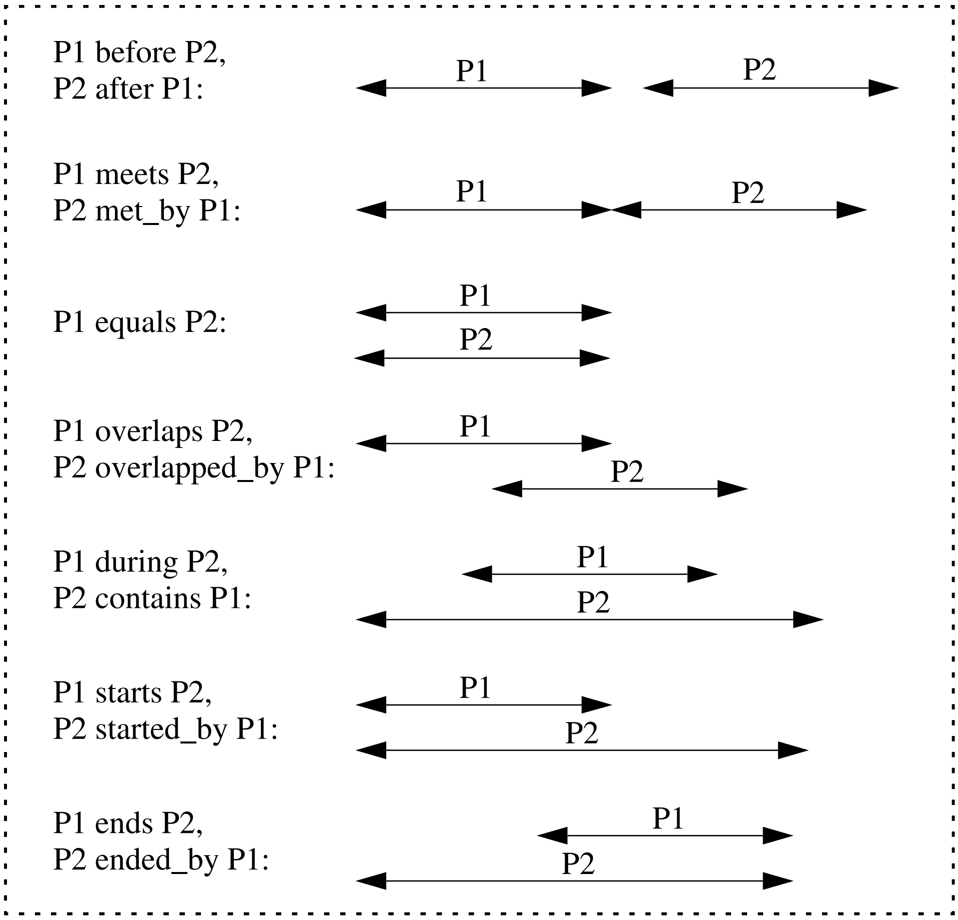
These two new types will own a collection of operations to manipulate and compare their values.

Allen´s Relations

J.F. Allen defined a set of thirteen binary operators that cover all the relations between two periods.

In the following figure, Figure 4, “Thirteen Allen´s relations on Periods”, page 44, these thirteen operators are described.

Figure 4. Thirteen Allen´s relations on Periods



In fact, six of those thirteen operators are inverse operators. For instance «P2 after P1» relation is the inverse of «P1 before P2». Only the «equal» operator has no inverse operator because of the symmetry of the equality.

Comparison operations between instants and periods

The classical comparison operators are extended to compare instants and periods. We will also introduce few new operators to deal with the Allen´s relations.
Before describing them, let us introduce two functionalities to the Period type: we can access the boundaries of a Period object thanks to the `Begin()` and `End()` operations which return Instant objects. They will be very useful to manage the interactions between Instant and Period objects.

First the six classical operators of comparison, namely «==, !=, <, >, <=, >=», are extended to comparisons of instants and periods. Let us asssume that P1 and P2 are two periods, and D1 and D2 are two instants:

- D1 == (!=) D2, iff D1 is equal to (different from) D2.
- D1 < (<=) D2, iff D1 is earlier than (or equal to) D2.
- D1 > (>=) D2, iff D1 is later than (or equal to) D2.
- P1 == P2, iff the beginning boundary of P1 is equal to the beginning boundary of P2, and the ending boundary of P1 is equal to the ending boundary of P2.
- P1 != P2, iff at least one of the two boundaries of P1 is different from the equivalent boundary of P2.
- P1 < P2, iff the beginning boundary of P1 is earlier than the beginning boundary of P2, or the beginning boundary of P1 is equal to the beginning boundary of P2 and the ending boundary of P1 is earlier than the ending boundary of P2.
- P1 <= P2, iff P1 < P2 or P1 == P2
- P1 > P2, iff the beginning boundary of P1 is later than the beginning boundary of P2, or the beginning boundary of P1 is equal to the beginning boundary of P2 and the ending boundary of P1 is later than the ending boundary of P2.
- P1 >= P2, iff P1 > P2 or P1 == P2

Second, in addition to the six classical operators, four new comparison operators will be used. We defined them in regards to periods comparisons. Let us assume that P1 and P2 are two periods:

- P1 ***PRECEDES*** P2, iff the ending boundary of P1 is inferior to the beginning boundary of P2.
- P1 ***MEETS*** P2, iff the ending boundary of P1 + one granule, is equal to the beginning boundary of P2.
- P1 ***OVERLAPS*** P2, iff there exists at least one chronon c, so that c belongs to P1 and c belongs to P2.
- P1 ***CONTAINS*** P2, iff the beginning boundary of P1 is earlier than the beginning boundary of P2, and the ending boundary of P1 is later than the ending boundary of P2.

The set of these ten operators enables us to cover all the comparison relations between instants and periods.
The different cases are described below.

If we take two instants, let us assume D1 and D2, they can interact in three ways:

- D1 is earlier than D2: D1 ***PRECEDES*** D2
- D1 is equal to D2 (different from): D1==D2 (D1!=D2)
- D1 is later than D2: D2 ***PRECEDES*** D1

NB: We use the «***PRECEDES***» operator instead of the «<» because it is more explicit and meaning full in regards to the order of the time line. The previous definitions of the operators can be applied to instants since an instant can be seen as a period of zero duration.

If we take one instant and one period, let us assume D and P, they can interact in three ways:

- D is earlier than P: D ***PRECEDES*** P
- D is contained in P: D ***OVERLAPS*** P
- D is later than P: P ***PRECEDES*** D

If we take one period and one instant, let us assume P and D, they can interact in three ways:

- P is earlier than D: P ***PRECEDES*** D
- P contains D: P ***CONTAINS*** D
- P is later than D: D ***PRECEDES*** P

NB: If we want to know more details about the position of the instant inside the period, for instance «D begins P», then we can use the `Begin()` and `End()` operations of the Period type to directly manipulate and compare the boundaries of the periods with instants.

If we take two periods, let us assume P1 and P2, they can interact as the thirteen Allen´s relations show:

- P1 is equal to P2: P1==P2
- P1 is before P2 (P2 is after P1): P1 ***PRECEDES*** P2
- P1 meets P2 (P2 is met by P1): P1 ***MEETS*** P2
- P1 overlaps P2 (P2 is overlapped by P1):
  P1.***Begin()***<P2.***Begin()*** and P1.***End()***>P2.***Begin()*** and P1.***End()***<P2.***End()***
- P1 starts P2 (P2 is started by P1):

P1.***Begin()***==P2.***Begin()*** and P1.***End()***<P2.***End()***

- P1 contains P2 (P2 during by P1): P1 ***CONTAINS*** P2
- P1 ends P2 (P2 is ended by P1):
  P1.***Begin()***>P2.***Begin()*** and P1.***End()***==P2.***End()***

NB: The ***OVERLAPS*** operator that we defined is wider than the Allen´s one. «P1 ***OVERLAPS*** P2» means, in the Allen´s relations, that P1 overlaps P2 or P2 overlaps P1.

We reduce the number of new operators to «***PRECEDES***», «***MEETS***», «***CONTAINS***» and «***OVERLAPS***». Some other ones like «STARTS» or «ENDS» could also be defined, but we do not define them for several reasons. First our four operators seem to be the most useful, and most used in the temporal requests. Second, they represent the set of operators defined in TSQL2. Third, with `Begin()` and `End()` functions, we can simulate other operators like «STARTS».

Then those four operators, and the classical ones, have to be defined in the Instant and Period types, in order to implement all the relations shown above.

Arithmetic operations dealing with instants and periods

The legal arithmetic operations dealing with instants or periods are not numerous. Adding or substracting an anchored quantity of time to an unanchored quantity of time are two of these valid operations. For instance, the «01/01/1996 10h30min» instant can be translated by an interval of «+1h15min», and to give the «01/01/1996 11h45min» instant.
In the following table, Table 9, “Legal Arithmetic Operations on Instants and Periods”, page 47, operations between instants, periods and intervals are described.

Table 9. Legal Arithmetic Operations on Instants and Periods

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

Other functionalities

Regarding to Period and Instant types, few functionalities can be added in order to make easier their use.

It would be interesting to compare two instants, but instead of getting a boolean result, we would like to obtain the earliest or the latest instant. So the two operators, ***FIRST*** and ***LAST***, will be added to the Instant type, and will return the earliest instant, and the latest instant, of the two instant operands.

About a period, we may want to calculate the duration of time between its two boundaries. Then we add the ***INTERVAL*** operation to the Period type. It will return an interval object, resulting from the calculation of the interval of time contained in the period.
An alternative to the addition of an interval to a period is to only move the ending boundary of the period by the interval quantity of time: this will be available thanks to the << (>>) shift operator.

Two other useful comparison operations, which are combinations of some previous ones can also be defined over periods: ***ADJACENT*** and ***MERGES***.

- A ***ADJACENT*** B is equivalent to A ***MEETS*** B or B ***MEETS*** A
- A ***MERGES*** B is equivalent to A ***OVERLAPS*** B or A ***ADJACENT*** B

Set operations

Periods can also be seen as sets of instants. Then, we may want to apply set operations to periods: $\cup$ (union), $\cap$ (intersection), \ (difference).
First, we will examine the behaviour of these operations on periods. Let us assume that P1 and P2 are two periods.

- P1 $\cup$ P2:

  - if P1 ***MERGES*** P2, or P1 == P2, or P1 ***CONTAINS*** P2, or P2 ***CONTAINS*** P1, then the result will be a period with the beginning boundary defined as the earliest of the beginning boundaries of P1 and P2, and the ending boundary defined as the latest of the ending boundaries of P1 and P2.

  - else, i.e. if P1 ***PRECEDES*** P2, or P2 ***PRECEDES*** P1, then the result will be a set of periods with the two periods P1 and P2 as elements.

- P1 $\cap$ P2:

  - if P1 and P2 are disjoint, then the result will be $\emptyset$.

- else, i.e. if P1 ***OVERLAPS*** P2, or P1 **==** P2, or P1 ***CONTAINS*** P2, or P2 ***CONTAINS*** P1, then the result will be a period with the beginning boundary defined as the latest of the beginning boundaries of P1 and P2, and the ending boundary defined as the earliest of the ending boundaries of P1 and P2.

• P1 \ P2:

- if P1 and P2 are disjoint, then the result will be the period P1.

- if P1 == P2, then the result will be ∅.

- if P2 ***CONTAINS*** P1, or P1 starts or ends P2, then the result will be ∅.

- if P2 starts P1, or P2 overlaps[1] P1, then the result will be a period with the beginning boundary defined as the ending boundary of P2 + one granule, and the ending boundary defined as the ending boundary of P1.

- if P2 ends P1, or P1 overlaps[2] P2, then the result will be a period with the beginning boundary defined as the beginning boundary of P1, and the ending boundary defined as the beginning boundary of P2 - one granule.

- if P1 ***CONTAINS*** P2, then the result will be a set of periods with two periods: the first period is defined from the beginning boundary of P1 to the beginning boundary of P2 - one granule, and the second one is defined from the ending boundary of P2 + one granule to the ending boundary of P1.

We face a dilemma since these operations introduce a new data type, the set of periods. There exist several names for this data type: for instance, Yu Kandrashina called them ‹*periods strings*›, Ladkin called them ‹*no convex periods*›, and in TSQL2 they are called ‹*temporal elements*›. In our model, we will call them ‹*period_set*›, and they are discussed in detail in the following paragraph.

The tables of the next page summarize the theoretical results of the set operations performed on periods, Table 10, “Set operations performed on periods”, page 50.

TODM will support set operations on periods where the result will be a period if the theoretical result is a period, or a nil value if the theoretical result is a set of periods.
But TODM will also deal with the set of non contiguous periods which are described in the following part. Of course a period can be seen as a set of period with only contains this period.

1. This is the Allen’s operator, and not the one introduced by TODM.
2. idem

Table 10. Set operations performed on periods

	P1 ∪ P2	P1 ∩ P2
P1 == P2 (P2 == P1)	P1	P1
P1 ***MEETS*** P2	[P1.Begin - P2.End)	∅
P2 ***MEETS*** P1	[P2.Begin - P1.End)	∅
P1 ***CONTAINS*** P2	P1	P2
P2 ***CONTAINS*** P1	P2	P1
P1 ***PRECEDES*** P2	{ P1, P2 }	∅
P2 ***PRECEDES*** P1	{ P2, P1 }	∅
P1 ***OVERLAPS*** P2 (P2 ***OVERLAPS*** P1)	[First(P1.Begin,P2.Begin) - Last(P1.End,P2.End))	[Last(P1.Begin,P2.Begin) - First(P1.End,P2.End))

	P1 \ P2	P2 \ P1
P1 == P2 (P2 == P1)	∅	∅
P1 ***MEETS*** P2	P1	P2
P2 ***MEETS*** P1	P1	P2
P1 ***CONTAINS*** P2	{ [P1.Begin - P2.Begin), [P2.End - P1.End) }	∅
P2 ***CONTAINS*** P1	∅	{ [P2.Begin - P1.Begin), [P1.End - P2.End) }
P1 ***PRECEDES*** P2	P1	P2
P2 ***PRECEDES*** P1	P1	P2
P1 ***OVERLAPS*** P2 (P2 ***OVERLAPS*** P1)	if P1.Begin < P2.Begin then [P1.Begin - P2.Begin) else [P2.End - P1.End)	if P2.Begin < P1.Begin then [P2.Begin - P1.Begin) else [P1.End - P2.End)

Set of periods

The previous table, Table 10, “Set operations performed on periods”, page 50, summarizes the results of the set operations performed on periods. Some of these results are sets of periods, as for instance the union of two disjoint periods. These sets of periods can also be used to timestamp facts with one or both of the valid time and transaction time dimensions. Then TODM introduces the **Period_Set** type to model those sets of periods containing non-contiguous periods.

In order to deal with periods and period_sets in a similar way, TODM redefines the set operations applied to periods and period_sets, and also generalizes some of the period comparison operators.

Set operations.

The set operations are defined to take into account the fact that period_sets are sets of periods, and each period can be considered as a set of instants.
In the following formula, P represents a period, and PS represents a period_set which contains the n non contiguous periods. Operations between two periods are performed as described before. The resulting period_sets are consistent with the definition of a period_set, i.e they only contain non contiguous periods.

$$PS = \bigcup_{i=1}^{n} P_i \text{ with } (\forall(i,j)/i \neq j),\ P_i \cap P_j = \varnothing$$

- Union between one period and one period_set

$$PS \cup P = \bigcup_{i=1}^{n} (P_i \cup P)$$

- Union between two period_sets

$$PS1 \cup PS2 = (\ldots((PS1 \cup PS2_1) \cup PS2_2)\ldots \cup PS2_n)$$

- Intersection between one period and one period_set

$$PS \cap P = \bigcup_{i=1}^{n} (P_i \cap P)$$

- Intersection between two period_sets

$$PS1 \cap PS2 = \bigcup_{i=1}^{n} (PS1 \cap P2_i) = \bigcup_{i=1}^{n} \left(\bigcup_{k=1}^{m} (P1_k \cap P2_i) \right)$$

- Difference between one period_set and one period

$$PS \backslash P = \bigcup_{i=1}^{n} (P_i \backslash P)$$

- Difference between one period and one period_set

$$P \backslash PS = \bigcup_{i=1}^{n} (P \backslash P_i)$$

- Difference between two period_sets

$$PS1 \backslash PS2 = (\ldots((PS1 \backslash PS2_1) \backslash PS2_2) \ldots \backslash PS2_n)$$

Some examples are given below:

- {[1996/01, 1996/03), [1996/06, 1996/09)} ∪
  {[1996/02, 1996/05), [1996/07, 1996/08)} =
  {[1996/01, 1996/05), [1996/06, 1996/09)}
- {[1996/01, 1996/03), [1996/06, 1996/09)} ∩
  {[1996/02, 1996/05), [1996/07, 1996/08)} =
  {[1996/02, 1996/03), [1996/07, 1996/08)}
- {[1996/01, 1996/03), [1996/06, 1996/09)} \
  {[1996/02, 1996/05), [1996/07, 1996/08)} =
  {[1996/01, 1996/02), [1996/06, 1996/07), [1996/08, 1996/09)}

Period_set comparisons.

First, *Begin*, *End*, *First* and *Last* utilities are generalized to period_sets. In the following formula, PS represents a period_set, P a period and I an instant.

- *Begin* applied to a period_set will return the earliest instant contained in the period_set.

$$\mathrm{Begin}(PS) = I;(I \in P, P \in PS)$$
$$/\forall(i, j)(P_i \in PS) \wedge (I_{ij} \in P_i) \wedge (I_{ij} \neq I);I \text{ PRECEDES } I_{ij}$$

- *End* applied to a period_set will return the latest instant contained in the period_set.

$$\mathrm{End}(PS) = I;(I \in P, P \in PS)$$
$$/\forall(i, j)(P_i \in PS) \wedge (I_{ij} \in P_i) \wedge (I_{ij} \neq I);I_{ij} \text{ PRECEDES } I$$

- *First* applied to a period_set will return the earliest period of the period_set, i.e. the period which contains the earliest instant.

$$\mathrm{First}(PS) = P;(P \in PS)/\mathrm{Begin}(P) = \mathrm{Begin}(PS)$$

- *Last* applied to a period_set will return the latest period of the period_set, i.e. the period which contains the latest instant.

$$\mathrm{Last}(PS) = P;(P \in PS)/\mathrm{End}(P) = \mathrm{End}(PS)$$

Second, *PRECEDES*, *OVERLAPS*, *CONTAINS* and == comparison operators are extended to period_sets. In the following formula, PS1 and PS2 represent two period_sets.

- $PS1 \text{ PRECEDES } PS2 \Leftrightarrow \mathrm{End}(PS1) \text{ PRECEDES } \mathrm{Begin}(PS2)$
- $PS1 \text{ OVERLAPS } PS2 \Leftrightarrow PS1 \cap PS2 \neq \varnothing$
- $PS1 \text{ CONTAINS } PS2 \Leftrightarrow PS2 \backslash PS1 = \varnothing$

Period_set functionalities.

- Calculation of the duration of a period_set

$$\mathrm{INTERVAL}(PS) = \sum_{i=1}^{n} \mathrm{INTERVAL}(PS_i)$$

All those operations enable us to deal with period_sets in a similar way as periods.

Of course, the same operations can be defined for sets of instants. An instant can be seen as a period of zero duration, and then can me handled as a period..

Forever & beginning

forever and *beginning* are two special instants. *beginning* is smaller than any other instant (-∞), and *forever* is greater than any other instant (+∞). To model these two instants, TODM introduces a sub-type of the Instant type: **Boundary**. And two specialisations, **Superior** and **Inferior** types. One instance of each these two types will exist, and they will be named respectively ´forever´ and ´beginning´. All the comparison and arithmetic operations of the Instant type will be overridden in the Superior and Inferior types in order to deal with the infinite characteristics of *forever* and *beginning*. When the user want to use one of these two special instants, he/she just has to use the forever object or the beginning object.
cf. Figure 9, “Time Manipulation types”, page 68.

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**».

In the previous paragraph, we described the instants and periods used for absolute dating. In this one, we will introduce into the model the facility for relative dating. The Instant type introduced previously is specialised in **Absolute_Instant** and **Relative_Instant** types.

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 temporal 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.

In the Relative_Instant type, the get and set operations are overridden. The get operation has to get the instant value of the temporal mark, and to compute its result with respect to the delay. There is no restriction about the dating of the temporal mark. Then the temporal mark may contain a relative instant, and the get operation will be performed recursively. The set operation has to provide the way to handle the temporal mark and the delay associated to a relative instant.

In the Relative_Instant type, the comparison operators are overridden in order to deal with relative dating. A comparison between two relative instants is only possible if the temporal mark of each relative instant is timestamped by an absolute instant or timestamped by a relative instant whose temporal mark is timestamped by an absolute instant, and recursively. An exception will be raised if it is not. For instance, the PRECEDES operation can be defined as following, with RI1 and RI2 two instances of the Relative_Instant type: RI1 PRECEDES RI2 if RI1 and RI2 are related to a same temporal mark and the delay of RI1 is smaller than the delay of RI2, or if the timestamp of the RI1's temporal mark plus RI1's delay is inferior to the timestamp of the RI2's temporal mark plus RI2's 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 relative instants

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.
Cf. Figure 9, “Time Manipulation types”, page 68.

Then when the user wants to handle the current date, he/she has just to perform the get operation of the *clock* object. And to indicate that a fact is valid until the current date, the user just has to put the *now* object as the ending boundary of the valid time period. Some restrictions exist about the use of now relative instants as boundaries of periods:

- if both the beginning and ending boundaries are now relative instants, then the period has to be valid at the current date of instantiation. For instance [now,now+5) will always be valid, whereas [now+5,now) will never be.

- if only the beginning boundary is now relative, then the ending boundary has to be *forever* since this is the only instant which will always be greater than the beginning one.

- if only the ending boundary is now relative, then the beginning boundary has to be an instant which precedes the current time. For instance [1996/12/01,now) is not valid if entered before December 1st 1996.

In fact these rules can be extended to any period using relative instants as boundaries: a period will be correct only if the system is sure that the beginning boundary precedes the ending one.

Any other now-relative instant can be instantiated by the user by simply defining a relative instant related to *clock*, just as *yesterday* and *tomorrow* are defined.

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.

Timestamps for time varying data

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 is stroring 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.

May be in an improper way, but in a simpleness wish, the "UC" sentence will be used in the different examples both over valid time and transaction time dimensions given in this document. However its meaning is different: for valid time timestamp it says that the data is valid until a time which is not know in the future corresponding to the next modification; whereas for transaction time timestamp it says that the data is current at the present time, but nothing is sayd about the future which does not exist over transaction time dimension.

Calendars & Granules

Until now, we have not spoken about the calendars, assuming that we are working with the default calendar, namely the Gregorian one. We have also assumed that we are working with the default granularities of the Gregorian calendar, namely year, month, day, hour, minute and second.

However, any time representation, anchored and 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. 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.

A super-type of Interval, Period and Instant types is introduced. This type is **Time_Element** and it will be linked to one calendar and one granule in order to model the fact that each quantity of time is expressed with regards to a precise granule of a precise calendar. The granule and calendar have to be supplied by the user at the creation of a new instance of Time_Element object.

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 the Time_Element type:

```
cast(into_granule:Granule)->Time_Element
```

The semantics of the `cast` operation are described in a previous chapter,“Conversion to finer or coarser granularities”, page 9. This operation has to be overridden in each subtype:

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

Regarding the instants, TODM introduces another functionality: the creation of period which represents the boundaries of the theoretical indeterminate instant resulting from the conversion of an instant into a finer granularity. Then the `create_finer_period` operation is defined in the Instant type:

```
create_finer_period(finer_granule:Granule)->p:Period
```

For instance, an instant expressed at the year-sized granularity can create a period representing the boundaries of the indeterminate instant at the month-sized granularity: ‹1996› can give [1996/01,1997/01).

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` operations defined in the Granule type.
Each of these operations will call a mapping function particular to each instance and known only at the instantiation.

With the Gregorian calendar, six granularities have to be defined, namely year, month, day, hour, minute and second. There will be six particular instances of the Granule type, respectively named *´year´*, *´month´*, *´day´*, *´hour´*, *´minute´* and *´second´*. And the `map_to_finer` and `map_to_coarser` operations will call the mapping functions provided. For instance, as the *year* granule is not linked to a coarser granule, the `map_to_coarser` operation has no behaviour; on the other hand *year* is linked to the *month* finer granule, and the regular mapping function «each year contains 12 months» is provided at the instantiation of the *year* granule. Regarding the *month* granule, the regular mapping function to the *year* granule «12 months are contained within a year», and the irregular mapping function to the *day* granule «a month contains from 28 to 31 days» are provided at the *month* granule instantiation.

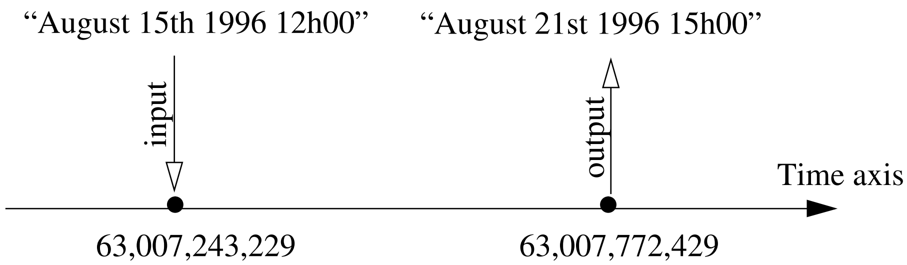
Cf. Figure 9, “Time Manipulation types”, page 68.

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 part of this chapter, comparison, arithmetic and manipulation operations of instants, periods and interval are performed without any intervention of the calendar characteristics.
However, a timestamp represents a quantity of time expressed in a given calendar and at a given granularity from this calendar. To model that, each Time_Element object will be related to a Calendar object and to a Granule object belonging to 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, Figure 5, “Gregorian Input/Output operations to/from the Time axis”, page 60, illustrates the input and output operations for the Gregorian calendar.

Figure 5. Gregorian Input/Output operations to/from the Time axis



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.
For instance, with the Gregorian calendar, the timezone in which is expressed the instant can be specified, as well as the use or not of daylight saving time (DST).

The input function has to take into account all these data to translate the sting timestamp into the right internal representation. And of course the output operation, which is the inverse function, has to translate the internal timestamp to the original timestamp format. So all the information specific to the calendar and owned by timestamp when performing the input, have to be kept within the database timestamp in order to be retrievable when performing the output. To implement this feature, a list of calendar properties will be specific to each calendar, each timestamp expressed in this calendar will own a list of couples (property_name, property_value), to store the values for the properties belonging to this calendar. This list of property values will be instantiated during the input translation, and its values will be used at the output translation.
For instance, in the Gregorian calendar, the unique internal timestamp format will be the use of chronons since the Gregorian origin, expressed in UTC. Then the information concerning the timezone in which the external timestamp is expressed and the use or not of DST will appear as Gregorian properties. The values used by the external timestamp will be stored in the list of property values of the internal timestamp. For more details regarding DST and time zones, cf. “Time Zones & Daylight Saving Times”, page 15.

In the same way as the time zone and DST data are defined for the Gregorian calendar, a week[1] mask can also be defined as a property of the Gregorian calendar. In fact weeks will only be used for input operations, and not for the storage of the timestamp. If required by the user, he can also use weeks for outputing timestamps, since weeks are only a mask

to see the timestamp in another external format.
For instance, the input procedure of the Gregorian calendar should be able to parse an external timestamp such as “Monday 6th week 1996 13:00 MET”, and store the “1996-02-05 12:00:00 UTC” timestamp. And the ouput operation should be able to translate the “1996-06-16 15:20 UTC” timestamp in a format such as “1996 Sunday 24th week 17:20 MET DST”.

Of course in the same way other units for user interface can be defined, as for instance trimester or semester.

In order to illustrates the functionalities provided by the input/output operations of the Gregorian calendar, the two formats adopted by TODM are given below.

- **YYYY**-**MM**-**DD** **hh**:**mm**:**ss** [**TZ**] [*DST*] where

  - YYYY represents the year

  - MM represents the month number in the year, from 01 to 12; default is 01

  - DD represents the day in the month, from 01 to 31; default is 01

  - hh represents the hours in the day, from 00 to 23; default is 00

  - mm represents the minute in the hour, from 00 to 59; default is 00

  - ss represents the seconds in the minute, from 00 to 60; default is 00

  - TZ represents the time zone name, cf. Table 16, “Time Zone abbreviations”, page 227; the default time zone is system’s one

  - if DST is present, then a first attempt will try to translate the timestamp to DST time; if it absent, then a first attempt will try to translate the timestamp to non-DST time

  ex: 1996/06/16 17:20 MET DST

- **YYYY** **day_name** **#week** “week” **hh**:**mm**:**ss** [**TZ**] [*DST*] where

  - YYYY represents the year

  - day_name represents the name of the week, from Monday to Sunday; default Monday

  - #week represents the number of the week in the year

  - hh represents the hours in the day, from 00 to 23

  - mm represents the minute in the hour, from 00 to 59

  - ss represents the seconds in the minute, from 00 to 60

1. The first day of the week is on Monday. The first week of the year contains the first Thursday of January. The weeks are numbered from 01 to 52. A 53th week appears when the year ends on a Thursday, or on a Thursday or Friday when this is a leap year.

- TZ represents the time zone name, cf. Table 16, “Time Zone abbreviations”, page 227

- if DST is present, then a first attempt will try to convert the timestamp to DST time; if it absent, then a first attempt will try to convert the timestamp to non-DST time

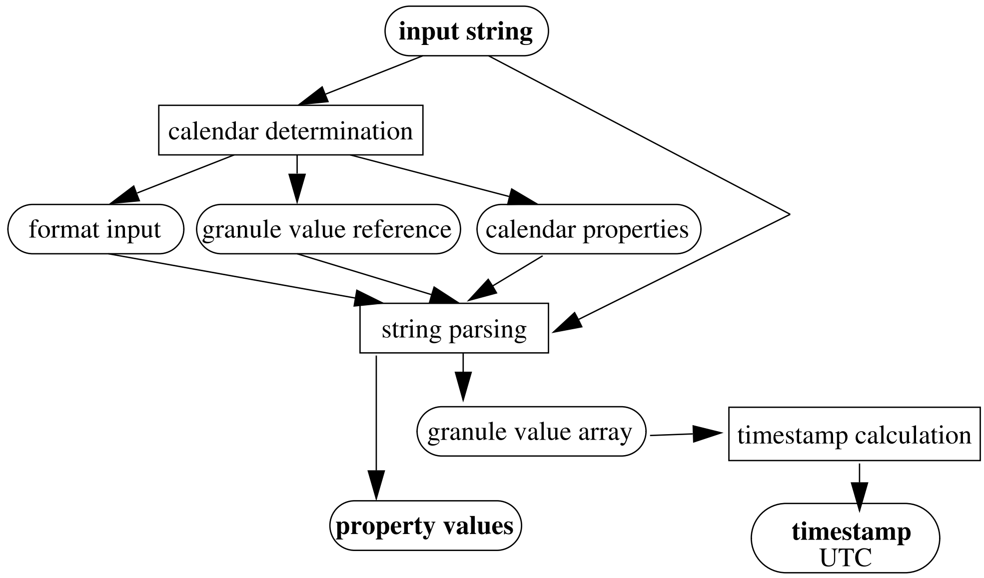
ex: 1996 Sunday 24 week 17:20 MEST

NB: Leap seconds can appear or be removed from selected dates, as for instance on 1995-12-31 where one leap second has been added just before the new year. Until now, only positive leap seconds have appeared, but negative ones are also allowed. Then if such leap seconds appear, the input procedure will allow non-existent instants which will be converted to the closest valid one. That will be more convenient for the user who would not expect this time to be invalid. For instance if the UTC run is 23:59:58, 00:00:00 where one negative leap second occurs, a timestamp such as 23:59:59 will be allowed and processed by the input operation to 23:59:58.

In the following pages simple procedures for Input/Output operations are shown with the “1996-06-16 17:20:00” instant from Gregorian calendar example.

- Figure 6, “Calendar Input procedure”, page 63
- Figure 7, “Calendar Output procedure”, page 64

Figure 6. Calendar Input procedure



Example: in Paris, France, “1996-06-16 17:20:00”

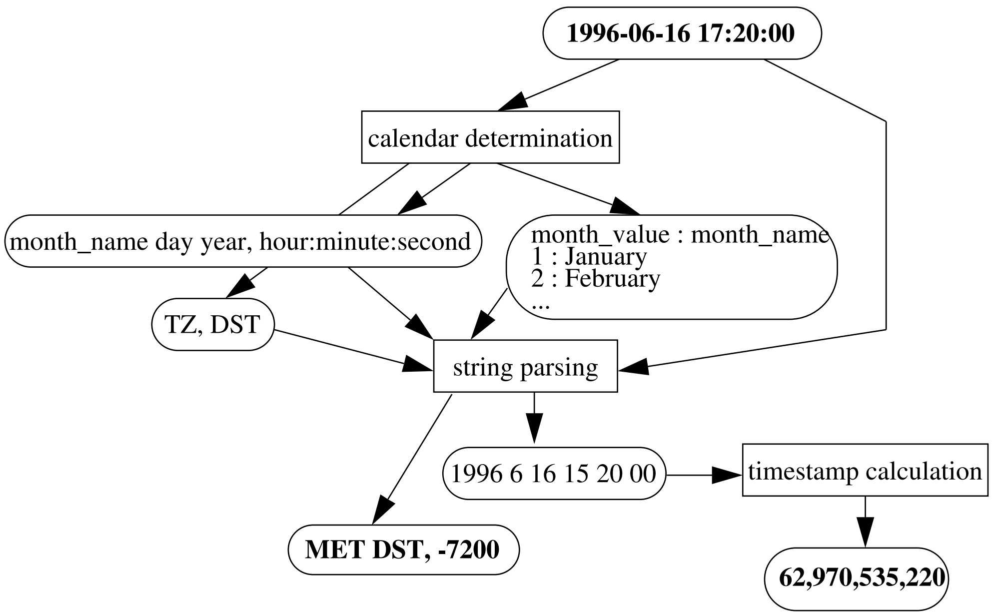
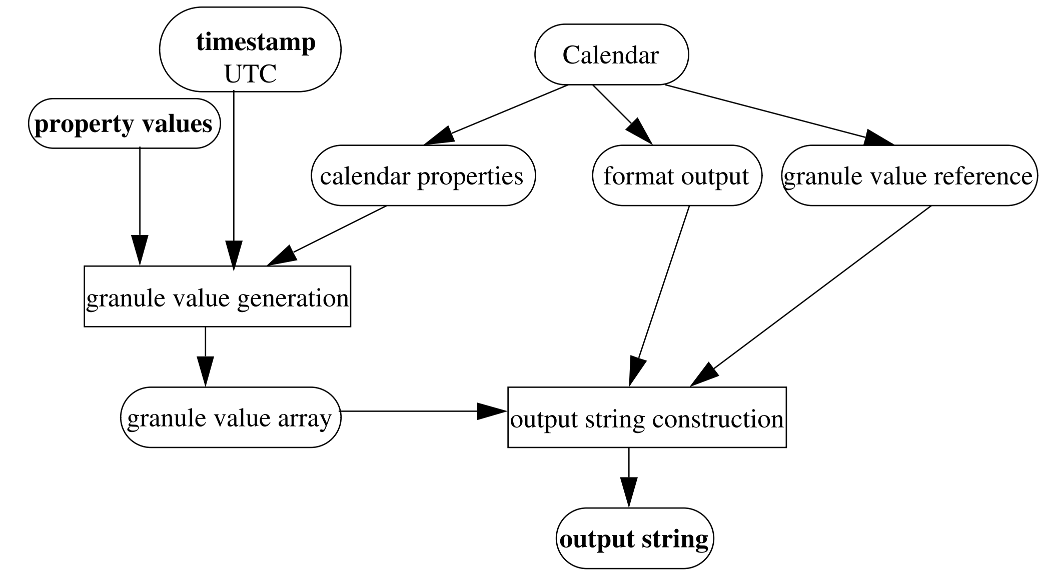
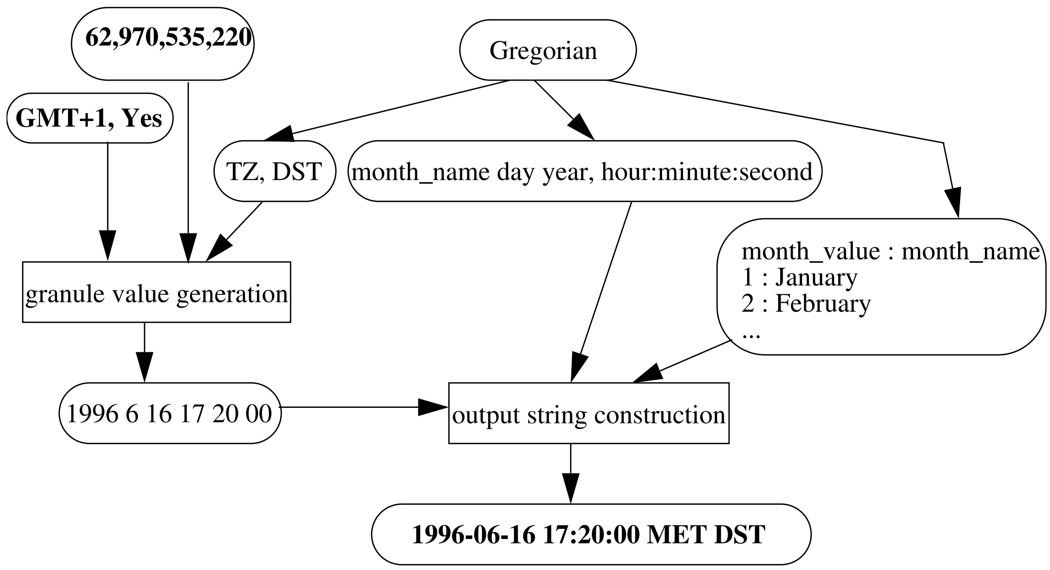


Figure 7. Calendar Output procedure



Example: the same as previous one



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. (cf. “Fixed and variable intervals”, page 13).

It would also be convenient to convert a timestamp from a calendar to another. Such operations are also calendar dependent, they should not be supported directly by the DBMS, rather the calendar system would decide what to do.
For instance, let us take two calendars: the Gregorian calendar and the TimeCard calendar - 8 hours days and 5 days weeks. Any instant of the Timecard calendar can be converted to an instant of the Gregorian calendar. But the conversion from Gregorian to TimeCard faces a problem since some instants of the Gregorian calendar do not exist in the TimeCard one. In this case, it is the TimeCard calendar which has to deal with such instants. Several solutions can be chosen: a mapping to the closest instant, a mapping to the latest or earliest instant, or an “undefined” exception can be raised. This choice has to be made at the implemention level of the TimeCard calendar.
The following figure, Figure 8, “Gregorian and TimeCard Calendars”, page 66, describes Gregorian and Timecard calendars with respect to the Time axis.

At the creation of a new calendar, input and output operations have to be supplied and need to be well defined in order to deal with such problems as described above.

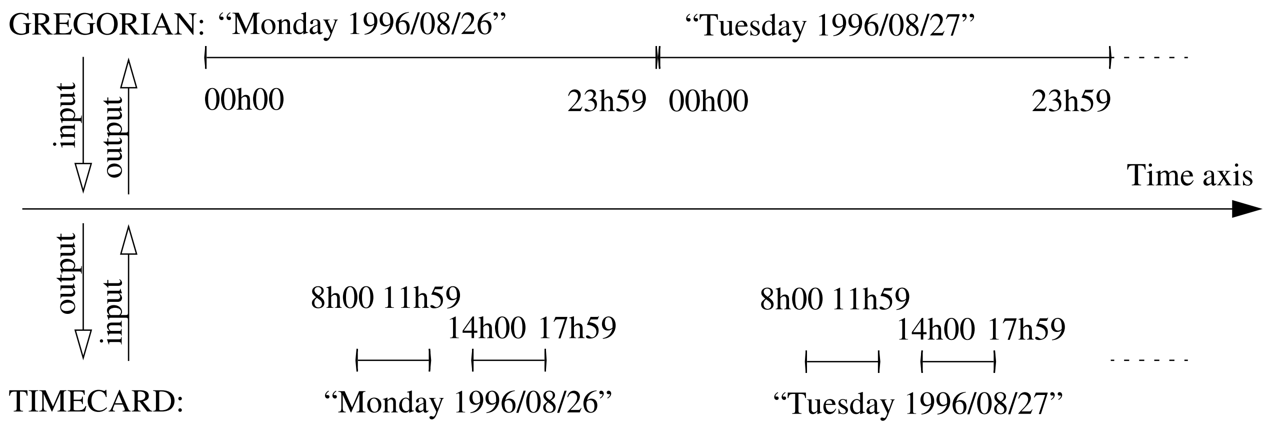


Figure 8. Gregorian and TimeCard Calendars

Time Manipulation Types

To summarize, on one hand there are the unanchored quantities of time, intervals, which are modelled by the **Interval** type. On the other hand there are the anchored quantities of time, instants and periods, which are modelled by the **Instant**, **Period** and **Period_set** types. Instant is then specialised into **Boundary** and **Relative_Instant** types.

Regarding relative time, a Relative_Instant object is linked to a **Temporal_Mark**, with a delay either in the past or in the future. Some specific instances are defined: ***clock*** which is a temporal mark enabling to get the current time; ***UT*** which is temporal mark modelling the Universal Time of the Greenwich meridian; ***tomorrow***, ***now*** and ***yesterday*** which are three relative instants delayed from *clock* by +1, 0 and -1 day-sized interval.

We also saw that each time quantity is expressed in a particular calendar, with a particular granularity of this calendar. Then each temporal object is linked to a calendar and to a granule of this calendar. Therefore we introduce a new type, **Time_Element**, which will be a supertype of the Interval, Instant, Period and Relative_Instant types. The Time_Element type will be related with the **Calendar** type and with the **Granule** type. The Gregorian calendar and its granule are provided as default.

The following figure, Figure 9, “Time Manipulation types”, page 68, summarizes the new temporal types we are introducing.

NB: In this schema, time quantities as instant for example, are introduced as new structured literals. The links towards Calendar and Granule types show that an instant is expressed in a calendar and at a granule. But of course they could not be relationships since relationships are only defined between objects. This problem is solved within the drafts of ODMG 2.0 where an attribute’s value is either a literal or an object identifier. An attribute which is object-valued enables to refer to an object, but without expectation of inverse link or referential integrity which are only maintain with relationships.

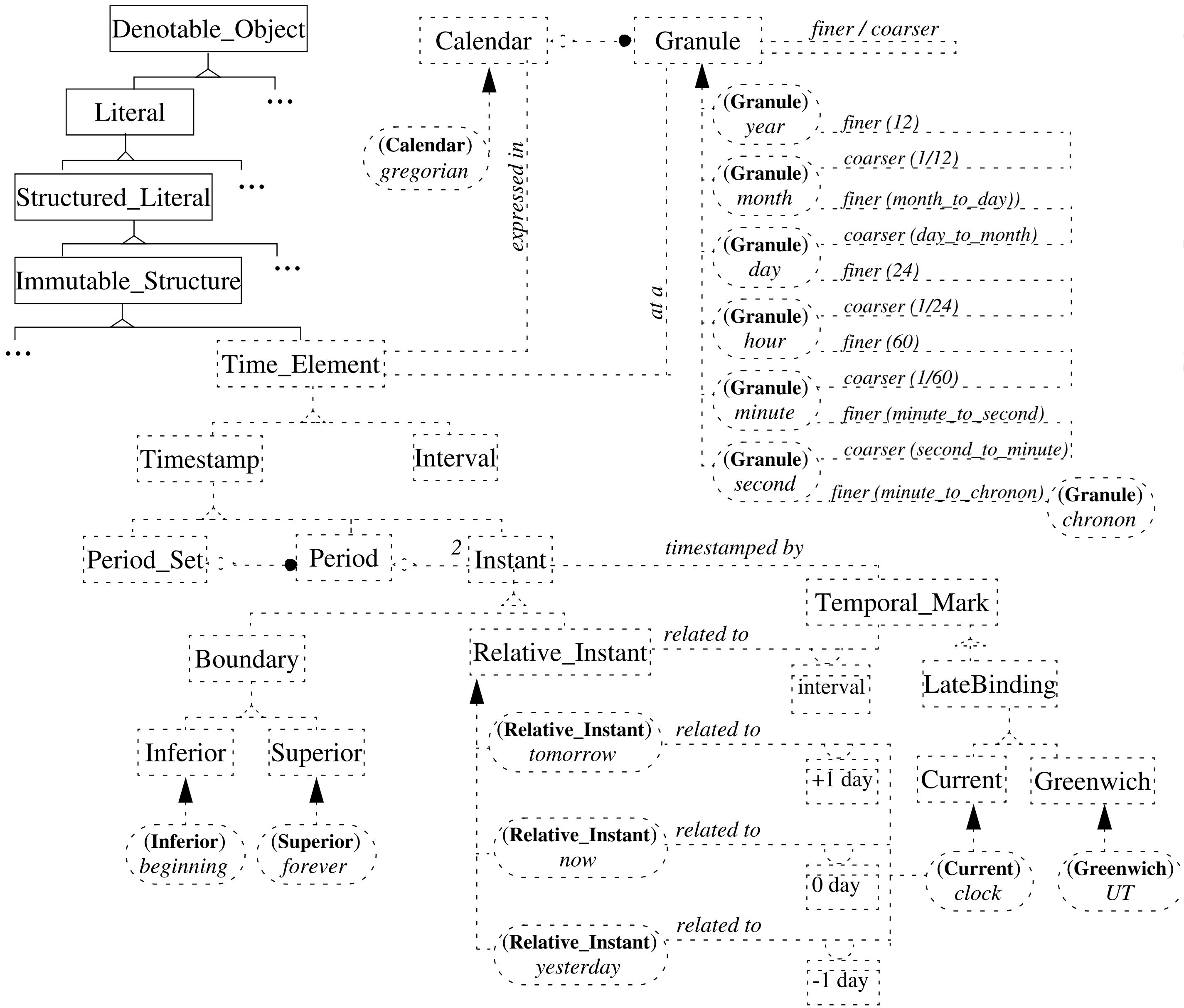


Figure 9. Time Manipulation types

Summary

We have defined new abstract data types to represent and manage time entities. These types are:

- ***Interval***, to deal with unanchored quantities of time.
- ***Instant***, to deal with time point.
- ***Period***, to deal with anchored quantities of time.
- ***Period_set***, to deal with set of non contiguous periods.
- ***Relative_Instant*** to deal with relative dating.
- ***Calendar*** and ***Granule***, to deal with the different calendars and granularities used to represent the time axis; the Gregorian calendar and its standard granule (year, month, day, hour, minute, second) are default ones.

We have also defined a set of operations for these types, in order to compare and to handle these different time representations.

All these types and operations define a wide range of tools for the user-defined time dimension, and also the basis for the storage and the manipulation of time for temporal varying data of valid and transaction time dimensions.

CHAPTER 6

Specification of Temporal Support

In this chapter, the model supporting the temporal features is described. An ODMG-93 like hierarchy of type is used to introduce the temporal characteristics at the attribute and object levels, as an extension of the model of ODMG, and more precisely an extension of ODM of ODMG-93.

The interfaces of the new types introduced by TODM are specified in details.

Overview

In summary, the ODMG data model can be described with the following notions:

- the primitive concept is the object
- objects are classified into types
- the behaviour of an object is defined by a set of operations
- the state of an object is defined by the values carried by the set of instance properties, either attributes or relationships

TODM will extend this model in order to be able to deal with temporal data. TODM will support data varying over time, and recorded with regards to the valid time dimension, or the transaction time dimension or both the valid time and transaction time dimensions.

Therefore, temporal instance properties and temporal objects are introduced.

NB: In the following lines, the ‹timestamp› word will be used to represent either instant timestamp, or period timestamp, or sets of timestamps. Differences are precised when necessary.

Temporal Instance Properties

The basic type hierarchy of the model of ODMG-93 describes Property as a sub-type of Characteristic, and as the super-type of Attribute and Relationship types.
TODM presents specialisations of both Attribute and Relationship types in order to support temporal features.

Temporal Attributes.

In ODMG, an attribute takes as its value a literal (or a set of literals)[1] - but we only keep literal in a more generic sense. With TODM, Attribute is extended with the **Temporal_Attribute** sub-type, and four specialisations appear: **Snapshot_Attribute**, **Historical_Attribute**, **Rollback_Attribute** and **Bitemporal_Attribute**.

1. Within the drafts of ODMG 2.0, attributes are extended and an attribute's value is now either a literal or an object identifier. An object-valued attribute enables to refer to an object but without expectation of inverse link or referential integrity which are only maintain within relationships.

Historical_Attribute allows the attribute to keep its whole history with regards to the valid time dimension. Rollback_Attribute allows the attribute to keep its whole audit with regards to the transaction time dimension. And Bitemporal_Attribute allows the attribute to keep its whole history with regards to both the valid time and transaction time dimensions. TODM also defines a Snapshot_Attribute type which acts as a classical attribute of ODMG, without any temporal support; its use will be explained later.
ODMG defines two operations on attributes, namely `set_value` and `get_value`. TODM overrides these operations in the three temporal attribute types, in order to deal with the timestamping characteristics.

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.

The `set_value` operation is not sufficient to express these two possibilities. Then a delete operation is also provided to allow the deletion of information in the already stored values. To retrieve and distinguish between the different reasons why a value has been modified, a flag will be implicitly stored with the different variants of the attributes. This flag will be *CURRENT/EVOLVED/DELETED*. Of course this evolution tracking flag will only be present within bitemporal and rollback attributes where the deletion of historical value is kept using the transaction time dimension. This flag could also be used in the operations allowing to retrieve the different histories and audits of the concerned temporal attributes.

More details and examples are given in the following part, “Temporal Attributes”, page 76.

Temporal Relationships.

ODMG-93 supplies three kinds of relationship, namely one_to_one, one_to_many and many_to_many.
In a similar way to temporal attributes, TODM defines **Temporal_Relationship** and four specialisations of each relationship. For instance, **Snapshot_one_to_one**, **Historical_one_to_one**, **Rollback_one_to_one** and **Bitemporal_one_to_one** are the four specialisations of one_to_one relationship.
ODMG describes the different operations for each kind of relationship. In TODM, these operations are overridden in order to deal with the timestamping characteristics. And as with temporal attributes, temporal relationships own evolution tracking flag management, operations to delete information and to retrieve whole histories and audits of the temporal relationships.

More details and examples are given in the following part, “Temporal Relationships”, page 85.

Summary.

TODM defines temporal attributes and temporal relationships, as specialisations of the attributes and relationships of ODMG-93. The operations of Attribute and Relationship types are overridden in the temporal types in order to deal with the temporal characteristics of these new types. An operation of deletion and different flags associated with the variants allow to manage the different situations for which a value need to be changed.

Temporal Objects

With temporal properties, TODM presents the temporal features at the attribute level. Whereas, with temporal objects, TODM introduces the temporal features at the object level. The interface, and above all the functionalities, are equivalent.

In the ODMG types hierarchy, Denotable_Object is split in Object and Literal, both decomposed in atomic and structured sub-types.
TODM proposes a temporal extension of both Atomic_Object type and Structured_Object type. A specialisation is introduced for each of the two last types: **Atomic_Temporal_Object** and **Structured_Temporal_Object**, and both are decomposed in ***Snapshot_Object***, ***Historical_Object***, ***Rollback_Object*** and ***Bitemporal_Object***. An historical object allows the object to keep its whole history with regards to the valid time dimension. A rollback object does the same for the transaction time dimension, and a bitemporal object for both valid time and transaction time dimensions. The histories and audits are composed of the different states of the object with the different timestamps associated. Then, TODM provides operations to get and set these object states with respect to the timestamping features. As with the temporal instance properties, evolution tracking flag management is provided, and other operations allowing to delete information and to retrieve histories and audits of the temporal objects are introduced.

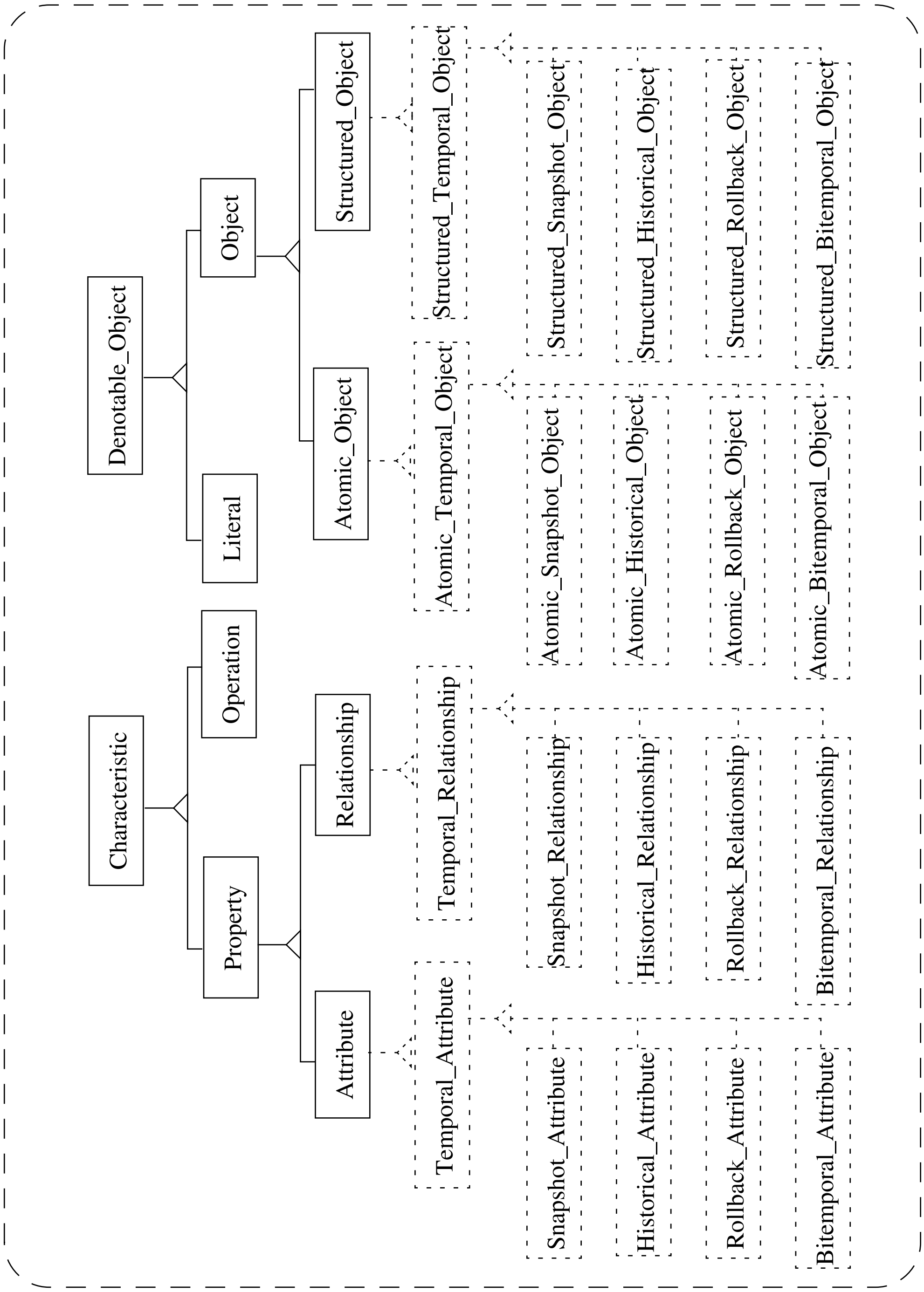
A snapshot object does not contain temporal characteristics at the object level. But as a standard object, it is composed of instance properties - attributes and relationships - and then it can own temporal instance properties as well as no time varying instance properties.

More details and examples are given in the following part, “Temporal Objects”, page 108.

Summary

The Figure 10, “Temporal Support Types”, page 75, shows the different type hierarchies that are introduced by TODM.

Figure 10. Temporal Support Types



Temporal Attributes

We define four particular attribute types, one snapshot and three temporal attributes: **Snapshot_Attribute**, **Historical_Attribute**, **Rollback_Attribute** and **Bitemporal_Attribute**.

These different types are described in the following lines and summarized in the Figure 11, “Temporal Attributes”, page 84.

Snapshot_Attribute

Snapshot_Attribute is a non temporal attribute, it acts as a normal attribute of the ODM of ODGM-93. Then in Snapshot_Attribute, we do not override neither define any operation for the manipulation of the values. Snapshot_Attribute inherits the `set_value` and `get_value` operations of Attribute, which are described below.

- `set_value(value:Literal)`

  The `set_value` operation gives a new value to the attribute. The old value is forgotten.
- `get_value()->value:Literal`

  The `get_value` operation returns the present value of the attribute. In fact it returns the argument of the last successful `set_value` operation call. The distinguished `nil` value is returned if the attribute has no value.

Historical_Attribute

Historical_Attribute enables us to support valid time attributes, and more precisely to deal with attributes of which the whole history regarding the valid time is maintained. Then we need to override the `set_value` and `get_value` operations inherited from Attribute. Information can be deleted in the history using the new `delete` operation. And a fourth operation allows to return the history of the attribute.
Since historical attributes have no way to keep track of a deleted information using transaction time timestamps, evolution tracking flags are not managed because there would be only one value *CURRENT*.

- `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 assigned 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.

Let us take a small example: a ‘Patient’ type is created as snapshot object, with a ‘temperature’ attribute as historical attribute. In the following lines, a small scenario of `set_value` and `delete` operations applied to the ‘temperature’ attribute of a `P1` instance of ‘Patient’ and their consequences on the internal data are described.

- `P1.temperature = {}`
- `set_value(37,[96-01,96-04))`
  ```
  P1.temperature = {
          1:(37,[96-01,96-04)}
  ```
- `set_value(38,[96-03,96-06))`
  ```
  P1.temperature = {
          1:(37,[96-01,96-03));
          2:(38,[96-03,96-06))}
  ```
- `delete([96-02-96-04))`
  ```
  P1.temperature = {
          1:(37,[96-01,96-02));
          2:(38,[96-04,96-06))}
  ```

Rollback_Attribute

Rollback_Attribute enables us to support transaction time attributes, and more precisely to deal with attributes of which the whole audit regarding the transaction time is maintained. Then we need to override the `set_value` and `get_value` operation inherited from Attribute. Information can not be deleted since no gap are allowed on the transaction time: in a database, there is always a value for an entity (literals for attributes, complex values for objects, sets for relationships ...) until this entity is deleted. But information can be corrected using the `correct_value` operation. And operations allow to return the audit of the attribute.
Evolution tracking flags are managed to distinguish the different types of data modification.

- `set_value(value:Literal)`

  The `set_value` operation gives a new value to the attribute. A default transaction time timestamp which is a particular period from the instant when the `set_value` operation is performed until the next modification, is assigned to the attribute by the system. The attribute keeps its old value in its audit shortening its transaction time to the current instant. The new information variant will own the *CURRENT* flag, and the old one will own an *EVOLVED* flag.
- `correct_value(value:Literal)`

  The `correct_value` operation corrects the value of the attribute. A default transaction time timestamp which is a particular period from the instant when the `corect_value` operation is performed until the next modification, is assigned to the attribute by the system. The attribute keeps its old value in its audit shortening its transaction time to the current instant. The new information variant will own the *CURRENT* flag, and the old one will own a *DELETED* flag.
- `get_value(transactiontime:Timestamp`$^{(*)}$`)`
  `->variant:{value:Literal,transactiontime:Timestamp}`

  The `get_value` operation returns the value of the attribute which was current at the transaction time timestamp and the timestamp. If no transaction time timestamp is given then the operation returns the value which is currently present, at the instant the operation is performed.

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

  $^{(*)}$: 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 `(value,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the argument one.
- `get_rollback()->rollback:Set<(value:Literal,`

```
                    transactiontime:Timestamp)>
```

The `get_rollback` operation returns the audit of the attribute made with the *CURRENT* and *EVOLVED* variants. The audit is represented as a set of couples, where each couple is composed of a value and of its transaction time timestamp.

- ```
  get_flagged_rollback(
              flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
         ->rollback:Set<(value:Literal,
              transactiontime:Timestamp),
              flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

  The `get_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 a value. of its transaction time timestamp and of its flag *CURRENT/EVOLVED/DELETED*.

Let us take a small example: a 'Patient' type is created as snapshot object, with a 'temperature' attribute as rollback attribute. In the following lines, a small scenario of `set_value` operations applied to the 'temperature' attribute of a `P1` instance of 'Patient' and their consequences on the internal data are described.

- `P1.temperature = {}`
- ```
  set_value(37) on 96-03-01
  P1.temperature = {
         1:(37,[96-03-01,UC[1]),CURRENT)}
  ```
- ```
  set_value(38) on 96-04-01
  P1.temperature = {
         1:(37,[96-03-01,96-04-01),EVOLVED);
         2:(38,[96-04-01,UC),CURRENT)}
  ```
- ```
  correct_value(39) on 96-05-01
  P1.temperature = {
         1:(37,[96-03-01,96-04-01),EVOLVED);
         2:(38,[96-04-01,96-05-01),DELETED);
         3:(39,[96-05-01,UC),CURRENT)}
  ```

1. One only knows that the ending boundary is greater than the current time, but not in the future since future time does not exist over transaction time dimension. This is different from the UC specific timestamp available for valid time timestamp which means that the ending boundary is located somewhere in the future, at the next modification of the information.

Bitemporal_Attribute

Bitemporal_Attribute enables us to support bitemporal attributes, and more precisely to deal with attributes of which the whole history regarding both the valid and the transaction time is maintained. Then we need to override the `set_value` and `get_value` operations inherited from Attribute. The current information can be logically deleted using the new `delete` operation. And operations allow to retrieve the audits and/or histories of the attribute.
Evolution tracking flags are managed to distinguish the different types of data modification.

- `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 assigned to the attribute by the system. A default transaction time timestamp which is a particular period from the instant when the `set_value` operation is performed until the next modification, is assigned to the attribute by the system. The attribute keeps its old value 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 these variants will own an *EVOLVED* flag.

- `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, 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_value(validtime:Timestamp`$^{(*)}$`,transactiontime:Timestamp`$^{(*)}$`)`
  `->variant{value:Literal,validtime:Timestamp,`
  `transactiontime:Timestamp}`

  The `get_value` operation returns the value of the attribute which was, is or will be valid at the valid time and was present at the transaction time timestamps, with both the valid and transaction time timestamps. If no valid time timestamp is given[1] then the operation returns the value which is valid at present, at the instant the operation is performed. If no transaction time timestamp is given then the operation returns the value which is currently present, at the instant the operation is performed. If both

1. A value nil is equivalent to no value.

valid time and transaction time are not specified, the two operations are equivalent, and return the value which is valid at present and currently present, at the instant the operation is performed.

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

(*): if the timestamps are not instants, several values can be related to the valid time and transaction time arguments, and the result will be a set of `(value,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<(value:Literal,validtime:Timestamp)>
  ```

  The `get_history` operation returns the history of the attribute which was present in the database at the transaction time timestamp. The history is represented as a set of couples, where each couple is composed of a value 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 operation performing date.

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

  (*): 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 argument one.

- ```
  get_flagged_history(transactiontime:Timestamp(*),
           flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
       ->history:Set<(value:Literal,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 value, 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 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_rollback(validtime:Timestamp(*))
       ->rollback:Set<(value:Literal,transactiontime:Timestamp)>
  ```
  ```

The `get_rollback` operation returns the audit of the attribute regarding the values 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 value 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 operation performing date.

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

(*): 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 `(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<(value:Literal,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 value, 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 values which are valid at present, at 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()->full_history:Set<(value:Literal,
                  validtime:Timestamp,transactiontime:Timestamp)>
  ```

  The `get_full_history` operation returns the whole audit of the attribute, with regards to both the transaction time and the valid time dimensions. But only the variants with the *CURRENT/EVOLVED* flags will be returned. The audit is represented 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))
         ->full_history:Set<(value:Literal,
             validtime:Timestamp,transactiontime:Timestamp,
             flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

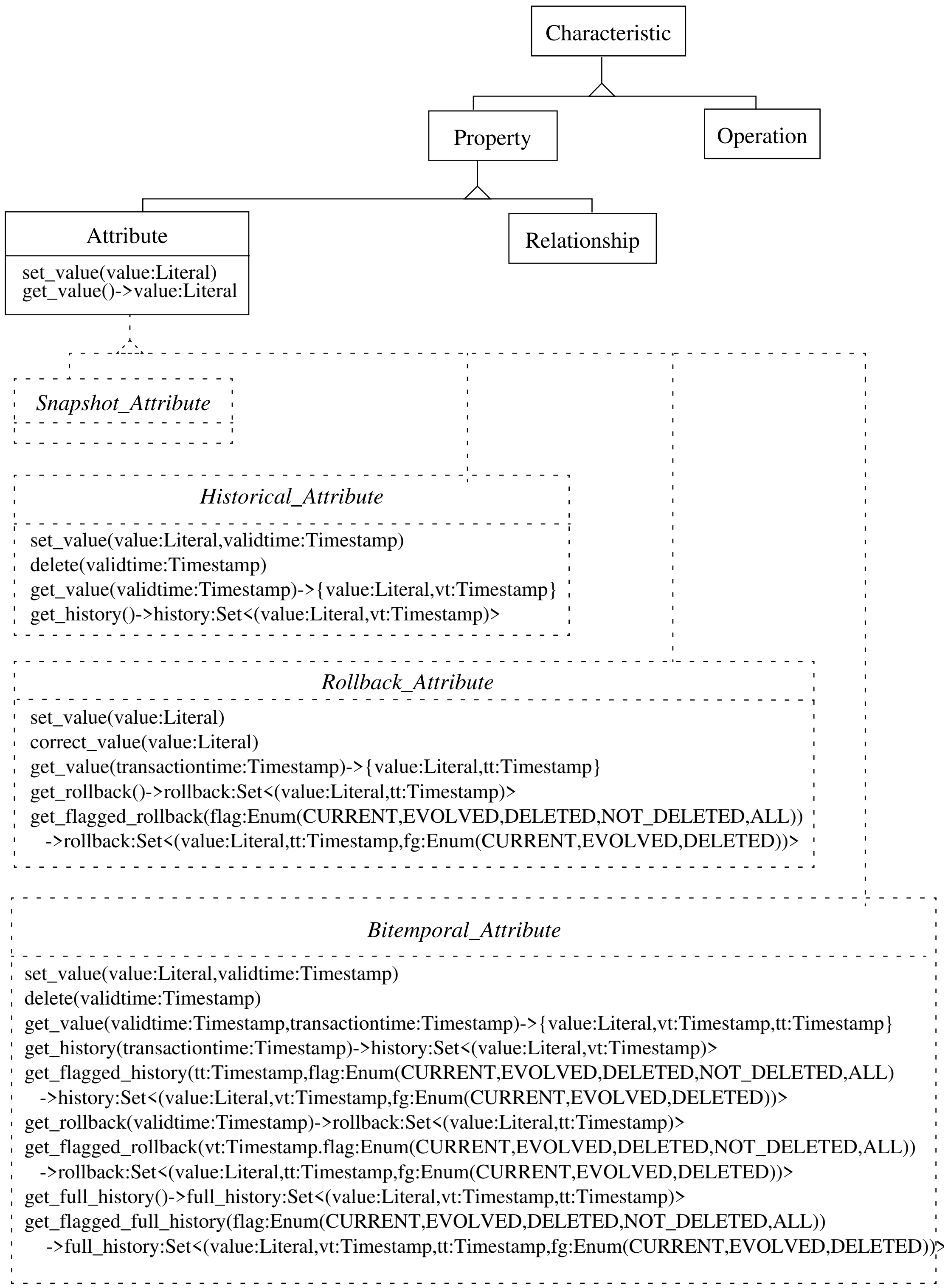
  To retrieve the history of the deleted information, or of all the information stored, the `get_full_flagged_history` operation has to be used. The audit is represented a set of variants, each variant is composed of a value, of its valid time timestamp, of its transaction time timestamp and of its flag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

Let us take a small example: a 'Patient' type is created as snapshot object, with a 'temperature' attribute as bitemporal attribute. In the following lines, a small scenario of `set_value` and `delete` operations applied to the 'temperature' attribute of a `P1` instance of 'Patient' and their consequences on the internal data are described.

- `P1.temperature = {}`
- `set_value(37,[96-01,96-04)) on 96-03-01`
```
P1.temperature = {
        1.1:(37,[96-01,96-04),[96-03,01-UC),CURRENT)}
```
- `set_value(38,[96-03,96-06)) on 96-04-01`
```
P1.temperature = {
        1.1:(37,[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
        1.2:(37,[96-01,96-03),[96-03-01,UC),CURRENT);
        2.1:(38,[96-03,96-06),[96-04-01,UC),CURRENT)}
```
- `delete([96-02,96-04)) on 96-05-01`
```
P1.temperature = {
        1.1:(37,[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
        1.2:(37,[96-01,96-03),[96-03-01,96-05-01),DELETED);
        1.3:(37,[96-01,96-02),[96-03-01,UC),CURRENT);
        2.1:(38,[96-03,96-06),[96-04-01,96-05-01),DELETED);
        2.2:(38,[96-04,96-06),[96-05-01,UC),CURRENT)}
```

The main operations described in this part are defined in details in "Overlapping vs non-overlapping timestamps", page 148.

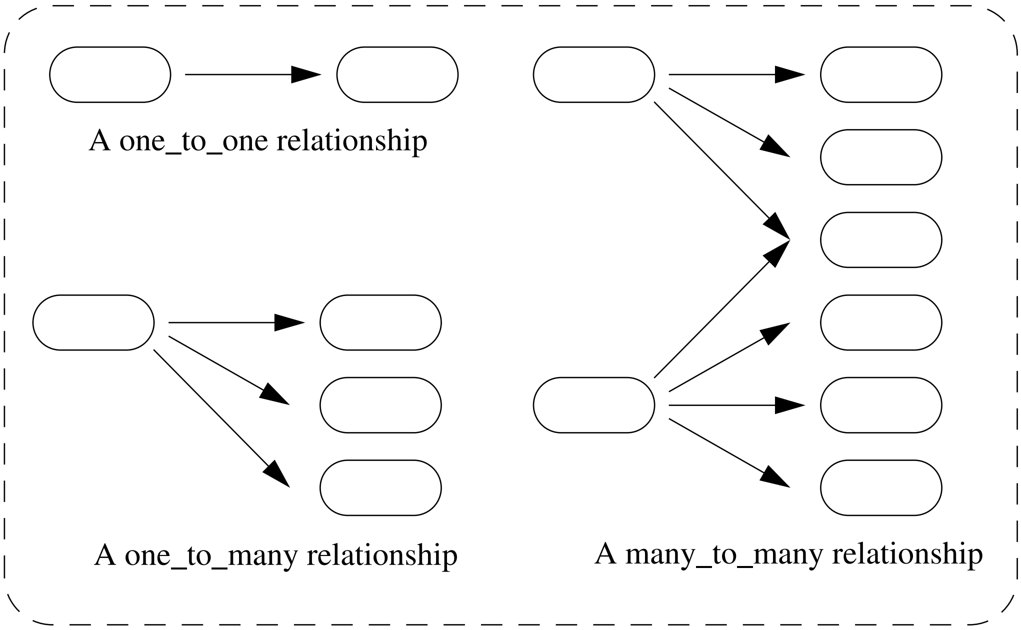
Figure 11. Temporal Attributes



Temporal Relationships

In ODMG-93, three types of binary relationships are defined, namely one-to-one, one-to-many and many-to-many. A one-to-one relationship is a relationship between two objects. A one-to-many relationships is a relationship between an object and a set of objects, and can be represented as a set of one-to-one relationships from the same object. A many-to-many relationship can be represented by two sets of one-to-many relationships. The next figure summarizes these relationship types, Figure 12, “ODMG-93 Relationships”, page 85.

Figure 12. ODMG-93 Relationships



Analogously to attributes, TODM will define four specifications for each relationship of ODMG, one snapshot and three temporal ones, to handle the evolution of relationships over one or both of the valid time and transaction time dimensions.
The only restriction is that relationships and inverse links, when an inverse relationship can be defined, have to evolve over the same time dimension. More details are given in the part “Temporal relationships and inverse links”, page 132.

A synchronisation constraint appears with temporal relationships involving objects which vary over the same time dimension. In this case, the destination object should own timestamp which includes the timestamp of the origin object.

These new relationships are described in the following paragraphs and summarized in several figures:

- Figure 13, “Temporal one-to-one Relationships”, page 105
- Figure 14, “Temporal one-to-many Relationships”, page 106
- Figure 15, “Temporal many-to-many Relationships”, page 107

More details regarding the inverse relationships are given in “Temporal relationships and inverse links”, page 132.

Snapshot relationships

Snapshot relationships are non temporal relationships, they act as normal relationships of the ODM of ODGM-93. Then in snapshot relationships, we do not override neither define any additional operation to manipulate the links between objects. Snapshot relationships inherit the operations of ODMG-93 relationships.

Snapshot_one_to_one.

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

  This operation creates a relationship between the two argument objects.
- `delete()`

  It destroys the relationship.
- `traverse(from:Denotable_Object)->to:Denotable_Object`

  `traverse` operation enables to reach the object which is in relationship with the argument object.

Snapshot_one_to_many.

- `create(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation creates a relationship between the first argument object and the second argument set of objects.
- `delete()`

  It destroys the relationship.
- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object)`

This operation adds a relationship between the two argument objects.

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

  This operation deletes the relationship between the two argument objects.

- `traverse(from:Denotable_Object)->s:Set<Denotable_Object>`

  `traverse` operation enables to reach the set of objects which is in relationship with the argument object.

Snapshot_many_to_many.

- `delete()`

  It destroys the relationship.

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

  This operation adds a relationship between the two argument objects.

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

  This operation deletes the relationship between the two argument objects.

- `add_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation adds a relationship between the first argument object and the second argument set of objects.

- `remove_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation deletes the relationship between the first argument object and the second argument set of objects.

- `remove_all_from(o1:Denotable_Object)`

  This operation deletes all the relationships from the argument object.

Historical relationships

Historical relationships are temporal relationships, they allow to deal with the history of the relationship regarding the valid time dimension. Then in historical relationships, we override operations of classical ODMG relationships, and add new operations to retrieve whole histories. As well as with the historical attributes, the evolution tracking flag is not managed in historical relationships, since they do not keep history of the database modifications over the transaction time dimension.

Historical_one_to_one.

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

This operation creates a new relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `create` operation is performed to *forever*, is assigned to the relationship by the system.

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

  It deletes the relationship which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `traverse(from:Denotable_Object,validtime:Timestamp`$^{(*)}$`)`
  `->{to:Denotable_Object,validtime:Timestamp}`

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

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

- `traverse_history(from:Denotable_Object)`
  `->history:Set<(to:Denotable_Object,validtime:Timestamp)>`

  The `traverse_history` operation returns the whole history of the relationship. The history is represented as a set of couples, where each couple is composed of the object which was, is or will be in relationship with the argument object and of the valid time timestamp.

Historical_one_to_many.

- `create(o1:Denotable_Object,s:Set<Denotable_Object>,`
  `validtime:Timestamp)`

  This operation creates a relationship between the first argument object and the second argument set of objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `create` operation is performed to *forever*, is assigned to the relationship by the system.

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

  It deletes the relationship which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`

```
validtime:Timestamp)
```

This operation adds a relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_one` operation is performed to *forever*, is assigned to the relationship by the system.

- `remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object, validtime:Timestamp(*))`

  This operation removes the relationship between the two argument objects which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  (*): if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `traverse(from:Denotable_Object,validtime:Timestamp(*))`
  `->{s:Set<Denotable_Object>,validtime:Timestamp}`

  `traverse` operation enables to reach the set of objects which were, are or will be in relationship with the argument object at the valid time timestamp, with the valid timestamp. If no valid time timestamp is given then the operation traverses the relationship which is valid at present, at the instant the operation is performed.

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

- `traverse_history(from:Denotable_Object)`
  `->history:Set<(s:Set<Denotable_Object>,validtime:Timestamp)>`

  The `traverse_history` operation returns the whole history of the relationship. The history is represented as a set of couples, where each couple is composed of the set of objects which were, are or will be in relationship with the argument object and of the valid time timestamp of this relationship.

Historical_many_to_many.

- `delete(validtime:Timestamp(*))`

  It destroys the relationship which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  (*): if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object, validtime:Timestamp)`

This operation adds a relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_one` operation is performed to *forever*, is assigned to the relationship by the system.

- `remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`
  `validtime:Timestamp`$^{(*)}$`)`

  This operation removes the relationship between the two argument objects which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `add_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>,`
  `validtime:Timestamp)`

  This operation adds a relationship between the first argument object and the second argument set of objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_many` operation is performed to *forever*, is assigned to the relationship by the system.

- `remove_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>,`
  `validtime:Timestamp`$^{(*)}$`)`

  This operation removes the relationship between the first argument object and the second argument set of objects which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time argument.

- `remove_all_from(o1:Denotable_Object,validtime:Timestamp`$^{(*)}$`)`

  This operation deletes all the relationships from the argument object which was, is or will be valid at the valid time timestamp. If no valid time timestamp is given then it defaults to [NOW,forever).

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time argument.

Let us take a small example: a ‘Patient’ type is created as snapshot object, with a ‘doctor’ relationship towards an ‘Investigator’ type as an historical one_to_one relationship. 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,[96-01,96-06))`

  ```
  P1.doctor = {
          1:(OID-P1,OID-I1,[96-01,96-06))}
  ```

- `create(P1,I2,[96-04,96-07))`
  ```
  P1.doctor = {
          1:(OID-P1,OID-I1,[96-01,96-04));
          2:(OID-P1,OID-I2,[96-04,96-07)}}
  ```
- `delete([96-03,96-05))`
  ```
  P1.doctor = {
          1:(OID-P1,OID-I1,[96-01,96-03));
          2:(OID-P1,OID-I2,[96-05,96-07)}}
  ```

Rollback relationships

Rollback relationships are temporal relationships, they allow to deal with the audit of the relationship regarding the transaction time dimension. Then in rollback relationships, we override operations of classical ODMG relationships, and add new operations to retrieve whole audits. But on transaction time dimension information can not be physically deleted, then a `correct` operation is introduced.
Analogously to the rollback attributes, the evolution tracking flag will be stored with the transaction time timestamps.

Rollback_one_to_one.

- `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 assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag 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.

Rollback_one_to_many.

- `create(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation creates a relationship between the first argument object and the second argument set of objects. A default transaction time timestamp which is a period from the instant when the `create` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.

- `delete()`

  `correct(o1:Denotable_Object,s:Set<Denotable_Object>)`

The `delete` operation is equivalent to `correct(o1,{})`. 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 ot *DELETED* for the old one.

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

  This operation adds a relationship between the two argument objects. A default transaction time timestamp which is a specific period from the instant when the `add_one_to_one` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.

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

  This operation removes the current relationship between the two argument objects which was present at the instant the operation is performed. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `o2` will be set to *CURRENT.*

- `traverse(from:Denotable_Object,transactiontime:Timestamp`(*)`)`
  `->{s:Set<Denotable_Object>,transactiontime:Timestamp}`

  `traverse` operation enables to reach the set of objects which were 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 `(s,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the argument one.

- `traverse_rollback(from:Denotable_Object)`
  `->rollback:Set<(s:Set<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 set of objects which were in relationship with the argument object and of the transaction time timestamp of this relationship.

  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<(s:Set<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 set of objects which were in relationship with the argument object, of the transaction time timestamp of this relationship and of its *CURRENT/EVOLVED*/DELETED flag.

Rollback_many_to_many.

- `delete()`

  The `delete` operation corrects the current relationship to an empty relationship. 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 ot *DELETED* for the old one.
- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object)`

  This operation adds a relationship between the two argument objects. A default transaction time timestamp which is a particular period from the instant when the `add_one_to_one` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.
- `remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object)`

  This operation removes the current relationship between the two argument objects which was present at the instant the operation is performed. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `o2` will be set to *CURRENT.*
- `add_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation adds a relationship between the first argument object and the second argument set of objects. A default transaction time timestamp which is a specific period from the instant when the `add_one_to_many` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.
- `remove_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>)`

  This operation removes the current relationship between the first argument object and the second argument set of objects which was present at the operation performing date. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `s` will be set to *CURRENT.*
- `remove_all_from(o1:Denotable_Object)`

This operation removes all the current relationships from the argument object which was present at the instant the operation is performed, replaces them with a NULL relationship. The new variant will own the *CURRENT* flag, and the evolution tracking flag will be set to *EVOLVED* for the old one.

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 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),DELETED};
        3:(OID-P1, Ø, [96-06-01,UC),CURRENT)}
```

Bitemporal relationships

Bitemporal relationships are temporal relationships, that allow to deal with the history of the relationship regarding the valid time dimension and also the audit regarding the transaction time dimension. Then in bitemporal relationships, we override operations of classical ODMG relationships, and add new operation to retrieve whole audits and histories. As well as with the bitemporal attributes, the *CURRENT/EVOLUTION/DELETED* flags will be stored with the valid time and transaction time timestamps.

Bitemporal_one_to_one.

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

  This operation creates a new relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `create` operation is performed to *forever*, is assigned to the relationship by the system. A default transaction time timestamp which is a specific period from the instant when the `create` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.

- `delete(validtime:Timestamp(*))`

  It deletes the relationship which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation deletes the relationship which is valid at present, at the instant the operation is performed. If valid time is not specifiedit deletes the relationship which is valid at present and currently present, at the instant the operation is performed. The *DELETED* flag will appear within the logically deleted variants.

  (*): if the timestamp is not instant, several relationships can be related to the valid time argument.

- ```
  traverse(from:Denotable_Object,validtime:Timestamp(*),
                           transactiontime:Timestamp(*))
         ->{to:Denotable_Object,validtime:Timestamp,
                           transactiontime:Timestamp}
  ```

  `traverse` operation enables to reach the object which was, is or will be in relationship with the argument object at the valid time timestamp and was current at the transaction time timestamp, with both the valid and transaction time timestamps. If no valid time timestamp is given then the operation traverses the relationship which is valid at present, at the instant the operation is performed. If no transaction time timestamp is given then the operation traverses the relationship which is current, at the instant the operation iss performed. If both valid time and transaction time are not specified, it traverses the relationship which is valid at present and currently present, at the instant the operation is performed.

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

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

- ```
  traverse_history(from:Denotable_Object,
                            transactiontime:Timestamp(*))
         ->history:Set<(to:Denotable_Object,validtime:Timestamp)>
  ```

  The `traverse_history` operation returns the whole history of the relationship which was current at the transaction time timestamp. The history is represented as a set of couples, where each couple is composed of the object which was, is or will be in relationship with the argument object and of the valid time timestamp of this relationship. 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 is 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 `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the second argument.

- ```
  traverse_flagged_history(from:Denotable_Object,
               transactiontime:Timestamp(*),
               flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
         ->history:Set<(to:Denotable_Object,validtime:Timestamp,
               flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

  The `traverse_flagged_history` operation returns the variants which satisfy the selection on the evolution tracking flag. The result is a set of triplets, where each triplet is composed of the object which was, is or will be in relationship with the argument object, of the valid time timestamp of this relationship and of the *CURRENT/EVOLVED/DELETED* flag. If no transaction time timestamp is given then the operation returns the history which is currently present, at the instant the operation is performed.

  (*): 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 `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the second argument.

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

  The `traverse_rollback` operation returns the whole audit of the relationship regarding the values 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 the object which was in relationship with the argument object and of the transaction time timestamp of this relationship. If no valid time timestamp is given then the operation returns the audit of the relationship which is valid at present, at the instant the operation is performed.

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

  (*): if the timestamp is not an instant, several relationships 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 second argument.

- ```
  traverse_flagged_rollback(from:Denotable_Object,
               validtime:Timestamp(*),
               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. 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 the *CURRENT/EVOLVED/DELETED* flag. If no valid time timestamp is given then the operation returns the audit which is present, at the instant the operation is performed.

(*): if the timestamp is not an instant, several relationships 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 second argument.

- ```
  traverse_full_history(from:Denotable_Object)
     ->full_history:Set<(to:Denotable_Object,validtime:Timestamp,
               transactiontime:Timestamp)>
  ```
  The `traverse_full_history` operation returns the whole audit of the relationship, in transaction time dimension as well as in valid time dimension. But only the variants with *CURRENT/EVOLVED* flags will be returned. The whole audit is a set of variants, each variant is composed of the object which was, is or will be in relationship with the argument object, of its valid time timestamping and of its transaction time timestamp.
- ```
  traverse_flagged_full_history(from:Denotable_Object,
                flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
     ->full_history:Set<(to:Denotable_Object,validtime:Timestamp,
               transactiontime:Timestamp,
               flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```
  The `traverse_flagged_full_history` operation returns the variants which satisfy the selection on the evolution tracking flag. The result is a set of variants, each variant is composed of the object which was, is or will be in relationship with the argument object, of its valid time timestamping, of its transaction time timestamping and of its *CURRENT/EVOLVED/DELETED* flag.

Bitemporal_one_to_many.

- ```
  create(o1:Denotable_Object,s:Set<Denotable_Object>,
                             validtime:Timestamp)
  ```
  This operation creates a relationship between the first argument object and the second argument set of objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `create` operation is performed to *forever*, is assigned to the relationship by the system. A default transaction time timestamp which is a specific period from the instant when the `create` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.
  ```

- `delete(validtime:Timestamp(*))`

  It deletes the relationship which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation deletes the relationship which is valid at present, at the instant the operation is performed. If valid time is not specified, it deletes the relationship which is valid at present and currently present, at the instant the operation is performed. The *DELETED* flag will appear within the logically deleted variants.

  (*): if the timestamp is not instant, several relationships can be related to the valid time argument.

- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`
  `validtime:Timestamp(*))`

  This operation adds a relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_one` operation is performed to *forever*, is assigned to the relationship by the system. A default transaction time timestamp which is a period from the instant when the `add_one_to_one` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the evolution tracking flag will be set to *EVOLVED*.

- `remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`
  `validtime:Timestamp(*))`

  This operation removes the relationship between the two argument objects which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation removes the relationship which is valid at present, at the instant the operation is performed. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `o2` will be set to *CURRENT.*

  (*): if the timestamp is not an instant, several relationships can be related to the valid time arguments.

- `traverse(from:Denotable_Object,validtime:Timestamp(*),`
  `transactiontime:Timestamp(*))`
  `->{s:Set<Denotable_Object>,validtime:Timestamp,`
  `transactiontime:Timestamp}`

  `traverse` operation enables to reach the set of objects which were, are or will be in relationship with the argument object at the valid time timestamp and was current at the transaction time timestamp, with both the valid and transaction time timestamps. If no valid time timestamp is given then the operation traverses the relationship which is valid at present, at the instant the operation is performed. If no transaction time timestamp is given then the operation traverses the relationship which is currently present, at the instant the operation is performed. If both valid time and transaction time are not specified, the two operations are equivalent, and traverse the relationship which is valid at present and currently present, at the instant the operation is performed.

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

- ```
  traverse_history(from:Denotable_Object,
                   transactiontime:Timestamp(*))
         ->history:Set<(s:Set<Denotable_Object>,
                        validtime:Timestamp)>
  ```

  The `traverse_history` operation returns the whole history of the relationship which was current at the transaction time timestamp. The history is represented as a set of couples, where each couple is composed of the set of objects which were, are or will be in relationship with the argument object and of the valid time timestamp of this relationship. If no transaction time timestamp is given then the operation returns the history which is currently present, at the instant the operation is performed.

  (*): 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 `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the second argument.

- ```
  traverse_flagged_history(from:Denotable_Object,
           transactiontime:Timestamp(*),
           flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
         ->history:Set<(s:Set<Denotable_Object>,validtime:Timestamp,
           flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```

  The `traverse_flagged_history` operation returns the variants which satisfy the selection on the evolution tracking flag. The result is a set of triplets, where each triplet is composed of the set of objects which were, are or will be in relationship with the argument object, of the valid time timestamp of this relationship and of the *CURRENT/EVOLVED/DELETED* flag. If no transaction time timestamp is given then the operation returns the history which is currently present, at the instant the operation is performed.

  (*): 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 `(history,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the second argument.

- ```
  traverse_rollback(from:Denotable_Object,validtime:Timestamp(*))
         ->rollback:Set<(s:Set<Denotable_Object>,
                         transactiontime:Timestamp)>
  ```
  ```

The `traverse_rollback` operation returns the whole audit of the relationship regarding the values 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 the set of objects which were in relationship with the argument object and of the transaction time timestamp of this relationship. If no valid time timestamp is given then the operation returns the audit of the relationship which are valid at present, at the instant the operation is performed.

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

- ```
  traverse_flagged_rollback(from:Denotable_Object,
              validtime:Timestamp(*),
              flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
        ->rollback:Set<(s:Set<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. The result is a set of triplets, where each triplet is composed of the set of objects which were in relationship with the argument object, of the transaction time timestamp of this relationship and of the *CURRENT/EVOLVED/DELETED* flag. If no valid time timestamp is given then the operation returns the audit which is present, at the instant the operation is performed.

  $^{(*)}$: if the timestamp is not an instant, several relationships 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 second argument.

- ```
  traverse_full_history(from:Denotable_Object)
        ->full_history:Set<(s:Set<Denotable_Object>,
             validtime:Timestamp,transactiontime:Timestamp)>
  ```

  The `traverse_full_history` operation returns the whole audit of the relationship, in transaction time dimension as well as in valid time dimension. But only the variants with *CURRENT/EVOLVED* flags will be returned. The whole audit is a set of variants, each variant is composed of the set of objects which were, is or will be in relationship with the argument object, of its valid time timestamping and of its transaction time timestamp.

- ```
  traverse_flagged_full_history(from:Denotable_Object,
              flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
     ->full_history:Set<(s:Set<Denotable_Object>,validtime:Timestamp,
             transactiontime:Timestamp,
             flag:Enum(CURRENT,EVOLVED,DELETED))>
  ```
  ```

The `traverse_flagged_full_history` operation returns the variants which satisfy the selection on the evolution tracking flag. The result is a set of variants, each variant is composed of the set of objects which were, is or will be in relationship with the argument object, of its valid time timestamping, of its transaction time timestamping and of its *CURRENT/EVOLVED/DELETED* flag.

Bitemporal_many_to_many.

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

  It deletes the relationship which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation deletes the relationship which is valid at present, at the instant the operation is performed. If valid time is not specified, it deletes the relationship which is valid at present and currently present, at the instant the operation is performed. The *DELETED* flag will appear within the logically deleted variants.

  $^{(*)}$: if the timestamp is not instant, several relationships can be related to the valid time argument.

- `add_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`
  `validtime:Timestamp`$^{(*)}$`)`

  This operation adds a relationship between the two argument objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_one` operation is performed to *forever*, is assigned to the relationship by the system. A default transaction time timestamp which is a particular period from the instant when the `add_one_to_one` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the *EVOLVED* flag will be assigned to them.

- `remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object,`
  `validtime:Timestamp`$^{(*)}$`)`

  This operation removes the relationship between the two argument objects which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation removes the relationship which is valid at present, at the instant the operation is performed. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `o2` will be set to *CURRENT.*

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time arguments.

- `add_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>,`
  `validtime:Timestamp`$^{(*)}$`)`

This operation adds a relationship between the first argument object and the second argument set of objects. If the valid time timestamp is not given by the user, then a default one which is a period from the instant when the `add_one_to_many` operation is performed to *forever*, is assigned to the relationship by the system. A default transaction time timestamp which is a particular period from the instant when the `add_one_to_one` operation is performed until the next modification, is assigned to the relationship by the system. The variant representing this new information will own the *CURRENT* flag. And if some already present variants need to be modified, then the *EVOLVED* flag will be assigned to them.

- `remove_one_to_many(o1:Denotable_Object,s:Set<Denotable_Object>,`

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

  This operation removes the relationship between the two arguments which was, is or will be valid at the valid time timestamp and was current at the instant the operation is performed. If no valid time timestamp is given then the operation removes the relationship which is valid at present, at the instant the operation is performed. The *EVOLVED* flag will appear within the old variant, whereas the evolution tracking flag of the new one without the relationship between `o1` and `s` will be set to *CURRENT.*

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time arguments.

- `remove_all_from(o1:Denotable_Object,validtime:Timestamp`$^{(*)}$`)`

  This operation removes all the current relationships from the argument object which was present at the instant the operation is performed, replaces them with a NULL relationship. The new variant will own the *CURRENT* flag, and the evolution tracking flag will be set to *EVOLVED* for the old one.

  $^{(*)}$: if the timestamp is not an instant, several relationships can be related to the valid time and transaction time arguments.

Let us take a small example: a 'Patient' type is created as snapshot object, with a 'doctor' relationship towards an 'Investigator' type as a bitemporal one_to_one relationship. In the following lines, a small scenario of create and delete 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,[96-01,96-06)) on 96-03-01`

  ```
  P1.doctor = {
          1.1:(OID-P1,OID-I1,[96-01,96-06),[96-03-01,UC),CURRENT)}
  ```
- `create(P1,I2,[96-04,96-07)) on 96-04-01`

  ```
  P1.doctor = {
          1.1:(OID-P1,OID-I1,[96-01,96-06),
                  [96-03-01,96-04-01),EVOLVED);
  ```

```
        1.2:(OID-P1,OID-I1,[96-01,96-04),
                 [96-03-01,UC),CURRENT);
        2.1:(OID-P1,OID-I2,[96-04,96-07),
                 [96-04-01,UC),CURRENT}}
```

- `delete([96-03,96-05)) on 96-06-01`

```
P1.doctor = {
        1.1:(OID-P1,OID-I1,[96-01,96-06),
                 [96-03-01,96-04-01),EVOLVED);
        1.2:(OID-P1,OID-I1,[96-01,96-04),
                 [96-03-01,96-06-01),DELETED);
        1.2:(OID-P1,OID-I1,[96-01,96-03),
                 [96-03-01,UC),CURRENT);
        2.1:(OID-P1,OID-I2,[96-04,96-07),
                 [96-04-01,96-06-01),DELETED}}
        2.1:(OID-P1,OID-I2,[96-05,96-07),
                 [96-04-01,UC),CURRENT}}
```

Figure 13. Temporal one-to-one Relationships

Figure 14. Temporal one-to-many Relationships

```
one_to_many
----------------------------------------------------------------
create(o1:Denotable_Object,s:Set<Denotable_Object>)
delete()
add_one_to_one(o1:Denotable_Object,o2:Denotable_Object)
remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object)
traverse(from:Denotable_Object)->s:Set<Denotable_Object>
```

```
Historical_one_to_many
----------------------------------------------------------------
create(o1:Denotable_Object,s:Set<Denotable_Object>,validtime:Timestamp)
delete(validtime:Timestamp)
add_one_to_one(o1:Denotable_Object,o2:Denotable_Object,validtime:Timestamp)
remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object,validtime:Timestamp)
traverse(from:Denotable_Object,validtime:Timestamp)->{s:Set<Denotable_Object>,vt:Timestamp}
traverse_history(from:Denotable_Object)->history:Seet<(s:Set<Denotable_Object>,vt:Timestamp)>
```

```
Rollback_one_to_many
----------------------------------------------------------------
create(o1:Denotable_Object,s:Set<Denotable_Object>)
delete() - correct(o1:Denotable_Object,s:Set<Denotable_Object>)
add_one_to_one(o1:Denotable_Object,o2:Denotable_Object)
remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object)
traverse(from:Denotable_Object,transactiontime:Timestamp)->{s:Set<Denotable_Object>,tt:Timestamp}
traverse_rollback(from:Denotable_Object)->rollback:Set<(s:Set<Denotable_Object>,tt:Timestamp)>
traverse_flagged_rollback(from:Denotable_Object,
                flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
   ->rollback:Set<(s:Set<Denotable_Object>,tt:Timestamp,fg:Enum(CURRENT,EVOLVED,DELETED))>
```

```
Bitemporal_one_to_many
----------------------------------------------------------------
create(o1:Denotable_Object,s:Set<Denotable_Object>,validtime:Timestamp)
delete(validtime:Timestamp)
add_one_to_one(o1:Denotable_Object,o2:Denotable_Object,validtime:Timestamp)
remove_one_to_one(o1:Denotable_Object,o2:Denotable_Object,validtime:Timestamp)
traverse(from:Denotable_Object,validtime:Timestamp,transactiontime:Timestamp)
    ->{s:Set<Denotable_Object>,vt:Timestamp,tt:Timestamp}
traverse_history(from:Denotable_Object,transactiontime:Timestamp)
   ->history:Seet<(s:Set<Denotable_Object>,vt:Timestamp)>
traverse_flagged_history(from:Denotable_Object,transactiontime:Timestamp,
                flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
   ->history:Set<(s:Set<Denotable_Object>,vt:Timestamp,fg:Enum(CURRENT,EVOLVED,DELETED))>
traverse_rollback(from:Denotable_Object,validtime:Timestamp)
   ->rollback:Set<(s:Set<Denotable_Object>,tt:Timestamp)>
traverse_flagged_rollback(from:Denotable_Object,validtime:Timestamp,
                flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
   ->rollback:Set<(s:Set<Denotable_Object>,tt:Timestamp,fg:Enum(CURRENT,EVOLVED,DELETED))>
traverse_full_history(from:Denotable_Object)
   ->full_history:Set<(s:Set<Denotable_Object>,vt:Timestamp,tt:Timestamp)>
traverse_flagged_full_history(from:Denotable_Object,
                flag:Enum(CURRENT,EVOLVED,DELETED,NOT_DELETED,ALL))
   ->full_history:Set<(s:Set<Denotable_Object>,vt:Timestamp,tt:Timestamp,
```

Figure 15. Temporal many-to-many Relationships

Temporal Objects

TODM extends the ODMG type hierarchy by defining temporal atomic types and temporal structured types. The hierarchy is shown in Figure 10, “Temporal Support Types”, page 75.

With temporal objects, TODM keeps track of the modifications of the object states. A type, **Object_State**, is introduced to represent such a state and used in the following interfaces of the temporal object operations. More details regarding object states are given later in the next chapter dedicater to an implementation point of view approach “Design of Temporal Objects”, page 189.

In the next lines, the description of snapshot, historical, rollback and bitemporal types are presented in details, and summarizes in Figure 16, “Temporal Objects”, page 116.
Then, temporal extensions of useful ODMG types, like collection ones, are suggested.

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 using a set of operations.
The only difference from the Object type from ODMG-93 is that Snapshot_Object can contain temporal instance properties, i.e. temporal attributes and/or temporal relationships, as well as classical instance properties. It also owns few operations which are not described in this part but later in the document, to allow interaction with temporal objects.

Historical_Object

Historical objects contain their whole history regarding the valid time dimension. This history is composed of a set of object states with the valid time timestamp associated to each of these states. Then Historical_Object type provides the operations to set and get the different states of the object, and to retrieve the whole history. Information can be deleted in the history using the `delete` operation.
Evolution tracking flags are not managed since with no transaction time support, its value

will only be *CURRENT*.

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

  The `set_state` operation gives a new state to the object. 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_state` operation is performed to *forever*, is assigned to the object by the system. The object keeps its old state in its history.

- `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. 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_state(validtime:Timestamp`(*)`)`
  `        ->{state:Object_State,validtime:Timestamp}`

  The `get_state` operation returns the state of the object 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 state which is valid at present, at the operation performing date.

  (*): 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 `(state,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.

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

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

Let us take a small example: a 'Patient' type is created as historical object, with a 'temperature' 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({(temperature,37)},[96-01,96-04))`
  ```
  Patient = {
          1:({(temperature,37)},[96-01,96-04))}
  ```
- `set_state({(temperature,38)},[96-03,96-06))`
  ```
  Patient = {
          1:({(temperature,37)},[96-01,96-03));
          2:({(temperature,38)},[96-03,96-06))}
  ```
- `delete([96-02,96-04))`
  ```
  Patient = {
          1:({(temperature,37)},[96-01,96-02));
  ```

```
2:({(temperature,38)},[96-04,96-06))}
```

Rollback_Object

Rollback objects contain their whole audit regarding the transaction time dimension. This audit is composed of a set of object states with a transaction time timestamp associated to each of these states. Then Rollback_Object type provides the operations to set and get the different states of the object, and to retrieve the whole audit. As well as rollback attributes, information can not be deleted in rollback objects, but can be corrected using the `correct_state` operation.
The evolution tracking flags are managed to distinguish the different types of data modification.

- `set_state(state:Object_State)`

  The `set_state` operation gives a new state to the object. A default transaction time timestamp which is a specific period from the instant when the `set_state` operation is performed until the next modification, is assigned to the object by the system. The object keeps its old state in its audit shortening its transaction time to the current instant. The new information variant will own the *CURRENT* flag, and the evolution tracking flag of the old one will be set to *EVOLVED*.
- `correct_state(state:Object_State)`

  The `correct_state` operation corrects the state of the object. A default transaction time timestamp which is a specific period from the instant when the `correct_state` operation is performed until the next modification, is assigned to the object by the system. The object keeps its old value in its audit shortening its transaction time to the current instant. The new information variant will own the *CURRENT* flag, and the evolution tracking flag of the old one will be set to *DELETED*.
- `get_state(transactiontime:Timestamp`(*)`)`
  `->{state:Object_State,transactiontime:Timestamp}`

  The `get_state` operation returns the state of the object which was current at the transaction time timestamp, and the transaction time timestamp. If no transaction time timestamp is given then the operation returns the state which is currently present, at the instant the operation is performed.

  Only the variants with *CURRENT/EVOLVED* flags are 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 `(state,transactiontime)` pairs, where each `transactiontime` timestamp is contained in the argument one.
- `get_rollback()->rollback:Set<(state:Object_State,`

```
                    transactiontime:Timestamp)>
```

The `get_rollback` operation returns the whole audit of the attribute. The audit is represented as a set of couples, where each couple is composed of a state and of its transaction time timestamp.

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

- ```
  get_flagged_rollback(
              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 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 *CURRENT/EVOLVED/DELETED* flag.

Let us take a small example: a 'Patient' type is created as rollback object, with a 'temperature' 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({(temperature,37)}) on 96-03-01
  Patient = {
          1:({(temperature,37)},[96-03-01,UC),CURRENT)}
  ```
- ```
  set_state({(temperature,38)}) on 96-04-01
  Patient = {
          1:({(temperature,37)},[96-03-01,96-04-01),EVOLVED);
          2:({(temperature,38)},[96-04-01,UC),CURRENT)}
  ```
- ```
  correct_state({(temperature,39)}) on 96-05-01
  Patient = {
          1:({(temperature,37)},[96-03-01,96-04-01),EVOLVED);
          2:({(temperature,38)},[96-04-01,96-05-01),DELETED);
          3:({(temperature,39)},[96-05-01,UC),CURRENT)}
  ```

Bitemporal_Object

Bitemporal objects contain their whole history regarding both the valid time dimension and the transaction time dimension. This full history is composed of a set of object states

with the valid time and transaction time timestamps associated to each of these states. Then Bitemporal_Object type provides the operations to set and get the different states of the object, and to retrieve the whole history. The current information can be logically deleted using the `delete` operation.
The evolution tracking flags are managed to distinguish the different types of data modification.

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

  The `set_state` operation gives a new state to the object. 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_state` operation is performed to *forever*, is assigned to the object by the system. A default transaction time timestamp which is a period from the instant when the `set_state` operation is performed until the next modification, is assigned to the object by the system. 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 present, at the instant the operation is performed. If no transaction time timestamp is given then the operation returns the state which is currently present, at the instant the operation is performed. If both valid time and transaction time are not specified, it returnss the state which is valid at present and currently present, at the instant the operation is performed.

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

Let us take a small example: a ‘Patient’ type is created as bitemporal object, with a ‘temperature’ 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.

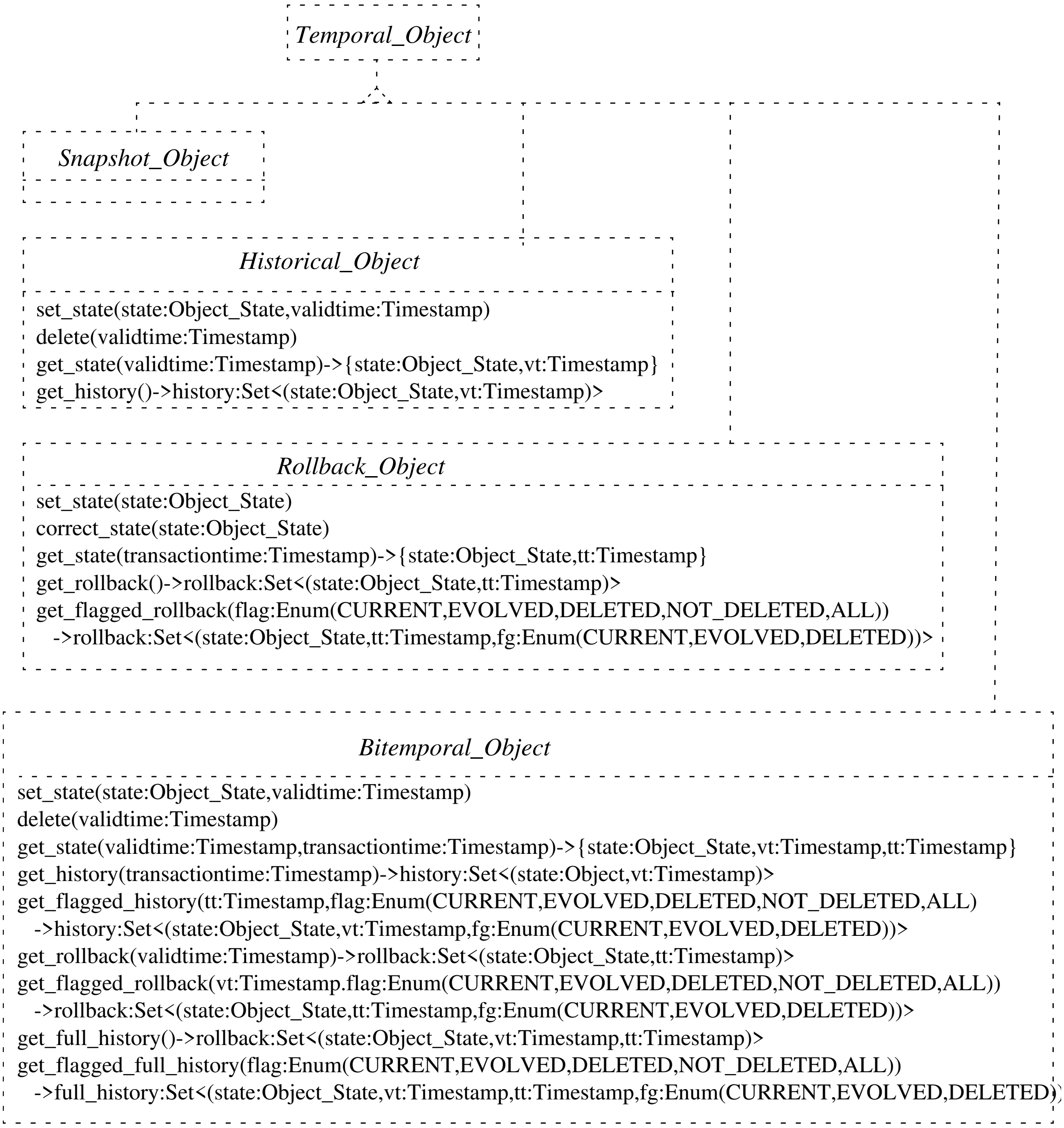
- `Patient = {}`
- ```
  set_state({(temperature,37)},[96-01,96-04)) on 96-03-01
  Patient = {
    1.1:({(temperature,37)},[96-01,96-04),[96-03,01-UC),CURRENT)}
  ```
- ```
  set_state({(temperature,38)},[96-03,96-06)) on 96-04-01
  Patient = {
    1.1:{(temperature,37)},[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
    1.2:{(temperature,37)},[96-01,96-03),[96-03-01,UC),CURRENT);
    2.1:{(temperature,38)},[96-03,96-06),[96-04-01,UC),CURRENT)}
  ```

- `delete([96-02,96-04)) on 96-05-01`

```
Patient = {
  1.1:({(temperature,37)},[96-01,96-04),[96-03-01,96-04-01),EVOLVED);
  1.2:{(temperature,37)},[96-01,96-03),[96-03-01,96-05-01),DELETED);
  1.3:{(temperature,37)},[96-01,96-02),[96-03-01,UC),CURRENT);
  2.1:({(temperature,38)},[96-03,96-06),[96-04-01,96-05-01),DELETED);
  2.2:({(temperature,38)},[96-04,96-06),[96-04-01,UC),CURRENT)}
```

The main operations described in this part are similar to the temporal attributes operations which are defined in details in “Overlapping vs non-overlapping timestamps”, page 148.

Figure 16. Temporal Objects



Creation & Destruction

This part is dedicated to the creation and destruction of temporal objects. The consequences on relationships between objects have also been studied.

Creation.

The creation of a new object is in fact an instantiation of the relevant type. Temporal objects as any other object follow this rule. A new instance can be created with values or not. There can be default values or not.

Regarding temporal types, some features have been added by TODM.
If the name of the extension is not provided by the user at the type definition step, with TODL, a default one will be created by TODL-TODM. Then at each creation, the new instance will be stored within the extension.
A boolean support, `current?`, is defined within all the temporal types introduced previously. It allows to know if the object is current in the database, and is set to `true` when the instance is created. This boolean support will be in read-only mode from a user point of view.

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. Such objects will be removed from the type extension, and their `current?` boolean support will be changed to `false`. These objects will no longer be accessible from within the extension. And the system will use the `current`? boolean to forbid any modification on these objects since they are no longer current in the database.

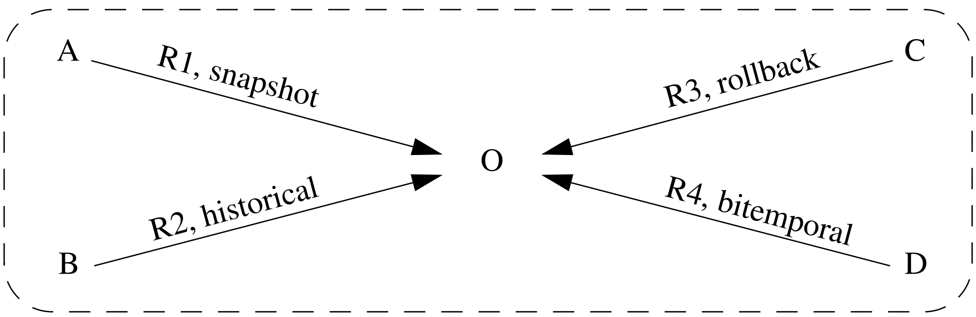
Therefore with logical deletion, objects are not present any more in the current state of the database but are kept in a past state. This characteristic can be useful with regards to relationships involving temporal objects.

Destruction and relationships.

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.

In the following lines, the different cases of destruction of temporal objects which are involved in some relationships are described. The problem is studied from the database coherency point of view, from the system point of view. Of course the information contained in the stored data can become erroneous with regards to the true of the real-world modelled within the database. But this is the responsibility of the user to maintain such truth coherency.

The figure below presents the scenario used for deletion descriptions.



physical destruction of O:

- On one hand, R1 and R2 do not keep track of the database evolutions since they do not deal with the transaction time dimension, they only manage the current state of the database. Then O can be removed from R1 and R2, and the resulted state of the database will remain consistent.
- On the other hand, R3 and R4 keeps track of the database modifications since they vary over the transaction time dimension. Then O can not be deleted without implying an inconsistent state for the database, since an old state of R3 and also of R4 refer to an object which would not exist. Therefore O can not be physically deleted. It will only be logically deleted until C and D are also deleted.
- If there does not exist any R3 or R4 relationships, then O can be physically destroyed and that leaves the database in a consistent state.

logical destruction of O:

- O, snapshot object: O is removed from its extension. The `current?` boolean support is changed to `false`. All the rollback and bitemporal instance properties of O are cut off and are no longer current in the database. O is no longer currently retrievable.

- O, historical object: O is removed from its extension. The `current?` boolean support is changed to `false`. O is no longer currently retrievable.

- O, rollback or bitemporal object: The current state of O is cut off. The `current?` boolean support is changed to `false`. O is no longer currently retrievable.

- If there does not exist any R3 or R4 relationships, then the logical destruction of O is in fact a physical destruction.

So the destruction of an object can not always be 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 the relationships where it participated. Therefore the ideal way to implement this is that the destruction of a transaction time relationship brings a check of the target objects. If they are already logically deleted then the system has to check if they can be physically destroyed.

Temporal structured types

All the ODMG structured types need to be extended in order to manage temporal features. See the Figure 17, “Temporal Structured Type”, page 131 for the whole hierarchy.

The following lines are dealing with historical extensions, i.e. data varying over the valid time dimension.

Historical_Collection<T>.

TODM redefines all the operations of the ODMG Collection<T> type.

- `create([pragma,{,pragma}])->hc:Historical_Collection<T>`
  `where pragma:=localization|representation`
  Creation of a new historical collection, i.e. a collection which maintains its history over the valid time dimension.
- `delete()`
  Deletion of the historical collection.
- `copy(hc2:Historical_Collection<T>[pragma{,pragma}])`
  `where pragma:=localization|representation`
  Its replaces the contents of the historical collection argument with those of the historical collection on which the operation is invoked.
- `insert_element(o:T,validtime:timestamp`$^{(*)}$`)`

The object is inserted in the version of the historical collection corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the collection with default valid time timestamp from the instant the operation is performed to forever.

(*): if the timestamp is not an instant, several versions can be related to the valid time argument, and partitioning of versions can be modified. For instance, assuming there is an historical collection composed of two versions: the first one, contains two objects `o1` and `o2`, with a valid time timestamp `[1996 January - 1996 March)`, and the second one contains only the object `o2`, with a timestamp `[1996 March - 1996 May)`. If the operation `insert_element(o3,[1996 February - 1996 April))` is performed then the result will be a collection with four versions: `{({o1,o2},[1996 January - 1996 February)); ({o1,o2,o3),[1996 February - 1996 March)); ({o2,o3},[1996 March - 1996 April)); ({o2},[1996 April - 1996 May])}`.

- `remove_element(o:T,validtime:timestamp(*))`

  The object is removed from the version of the historical collection corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the collection.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_element_at(current_position:Iterator,`
  `validtime:timestamp(*))`

  The object located at the given position is removed from the version of the historical collection corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the collection.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.

- `replace_element_at(o:T,current_position:Iterator,`
  `validtime:timestamp(*))`

  The object located at the given position is replaced by the object argument in the version of the historical collection corresponding to the valid time argument. If no valid time is given then the object is replaced in the current version of the collection.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.

- `retrieve_element_at(current_position:Iterator,`
  `validtime:timestamp(*))->{element:T,validtime:Timestamp}`

  It returns the object located at the given position in the version of the historical collection corresponding to the valid time argument, and the valid time timestamp. If no valid time is given then the object is returned from the current version of the collection.

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

- `select_element(predicate:String,validtime:Timestamp`$^{(*)}$`)`
  `->{element:T,validtime:Timestamp}`

  It returns the object satisfying the given predicate from the version of the historical collection corresponding to the valid time argument, and the valid time timestamp. If no valid time is given then the object is returned from the current version of the collection. If several elements satisfy the given predicate, then the operation raises an exception as the ODMG one does.

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

- `select(predicate:String,validtime:Timestamp`$^{(*)}$`)`
  `->{c:Collection<T>,validtime:Timestamp}`

  It returns a collection of objects satisfying the given predicate from the version of the historical collection corresponding to the valid time argument, and the valid time timestamp. If no valid time is given then the objects are returned from the current version of the collection.

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

- `exists?(predicate:String,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Boolean,validtime:Timestamp}`

  If it exists an object satisfying the given predicate in the version of the historical collection corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the predicate is tested in the current version of the collection.

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

- `contains_elements?(o:T,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Boolean,validtime:Timestamp}`

  If the given object is contained in the version of the historical collection corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the test is performed in the current version of the collection.

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

- `create_iterator(stable:Boolean,validtime:Timestamp(*))`
  `->{i:Iterator,validtime:Timestamp}`

  This creates an iterator over the version of the historical collection corresponding to the valid time argument, and also returns the valid time timestamp. If no valid time is given then the iterator will be over the current version of the collection.

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

Of course, Historical_Collection<T> and all the historical types that will be described in the following lines, inherit all the operation defined in the Structured_Historical_Object, as for instance the `get_history` operation which enables to handle the whole history of the object.

Historical_Set<T>.

In a set, a same object cannot appear twice.
TODM redefines all the operations of the ODMG Set<T> type.

- `create([pragma,{,pragma}])->hs:Historical_Set<T>`
  `where pragma:=localization|representation`

  Creation of a new historical set, i.e. a set which maintains its history over the valid time dimension.

- `insert_element(o:T,validtime:timestamp(*))`

  The object is inserted in the version of the historical set corresponding to the valid time argument, except if it is already a member of the set. If no valid time is given then the object is inserted in the current version of the set with default valid time timestamp from the instant the operation is performed to forever

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.

- `union(s2:Set<T>,validtime:Timestamp(*))`
  `->{s3:Set<T>,validtime:Timestamp}`

  The elements of the returned set are the union of the elements of the given set and those of the version of the historical set corresponding to the valid time argument; this set is returned with the valid time argument. If no valid time is given then the union is performed with the current version of the set.

(*): if the timestamp is not an instant, several versions can be related to the valid time argument, and the result will be a set of `(s3,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.

- `intersection(s2:Set<T>,validtime:Timestamp(*))`
  `->{s3:Set<T>,validtime:Timestamp}`

  The elements of the returned set are the intersection of the elements of the given set and those of the version of the historical set corresponding to the valid time argument; this set is returned with the valid time argument. If no valid time is given then the intersection is performed with the current version of the set.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument, and the result will be a set of `(s3,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.

- `difference(s2:Set<T>,validtime:Timestamp(*))`
  `->{s3:Set<T>,validtime:Timestamp}`

  The elements of the returned set are the elements of the version of the historical set corresponding to the valid time argument which are not member of the given set; this set is returned with the valid time argument. If no valid time is given then the difference is performed with the current version of the set.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument, and the result will be a set of `(s3,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.

- `is_subset?(s2:Set<T>,validtime:Timestamp(*))`
  `->{b:Boolean,validtime:Timestamp}`

  If the argument set is a subset of the version of the historical set corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the predicate is tested on the current version of the set.

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

- `is_proper_subset?(s2:Set<T>,validtime:Timestamp(*))`
  `->{b:Boolean,validtime:Timestamp}`

  If the argument set is a proper subset of the version of the historical set corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the predicate is tested on the current version of the set.

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

- `is_superset?(s2:Set<T>,validtime:Timestamp`$^{(*)}$`)`

  `->{b:Boolean,validtime:Timestamp}`

  If the argument set is a superset of the version of the historical set corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the predicate is tested on the current version of the set.

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

- `is_proper_superset?(s2:Set<T>,validtime:Timestamp`$^{(*)}$`)`

  `->{b:Boolean,validtime:Timestamp}`

  If the argument set is a proper superset of the version of the historical set corresponding to the valid time argument, then the result will be *true*; else it will be *false*; this boolean is returned with the valid time timestamp. If no valid time is given then the predicate is tested on the current version of the set.

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

Some additionnal operations, specific to historical sets may be added: for instance the union, the intersection and the difference between two historical sets.

Historical_Bag<T>.

In a bag, several occurrences of a same object are allowed.
TODM redefines all the operations of the ODMG Bag<T> type.

- `create([pragma,{,pragma}])->hb:Historical_Bag<T>`

  `where pragma:=localization|representation`

  Creation of a new historical bag, i.e. a bag which maintains its history over the valid time dimension.

- `insert_element(o:T,validtime:timestamp`$^{(*)}$`)`

  The object is inserted in the version of the historical bag corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the bag with default valid time timestamp from the instant the operation is performed to forever.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_element(ot:T,validtime:Timestamp`$^{(*)}$`)`

This decreases by one the count of the argument object in the version of the historical bag corresponding to the valid time argument. If no valid time is given then the count is reduced in the current version of the bag.

$^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `select(predicate:String,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Bag<T>,validtime:Timestamp}`

  It returns a bag of objects satisfying the given predicate from the version of the historical bag corresponding to the valid time argument; this bag is returned with the valid time timestamp. If no valid time is given then the objects are returned from the current version of the bag.

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

- `union(b2:Bag<T>,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Bag<T>,validtime:Timestamp}`

  The returned bag is the argument bag where are inserted all the elements of the version of the historical bag corresponding to the valid time argument; this bag is returned with the valid time timestamp. If no valid time is given then the union is performed with the current version of the bag.

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

- `intersection(b2:Bag<T>,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Bag<T>,validtime:Timestamp}`

  The returned bag contains each element which is present in the given bag and in the version of the historical bag corresponding to the valid time argument; this bag is returned with the valid time timestamp. If no valid time is given then the intersection is performed with the current version of the bag.

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

- `difference(b2:Bag<T>,validtime:Timestamp`$^{(*)}$`)`
  `->{b:Bag<T>,validtime:Timestamp}`

  The returned bag contains each element of the version of the historical bag corresponding to the valid time argument minus the elements in the given bag; this bag is returned with the valid time timestamp. If no valid time is given then the difference is performed with the current version of the bag.

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

Historical_List<T>.

The elements of a list are ordered by the order of their insertion.
TODM redefines all the operations of the ODMG List<T> type.

- `create([pragma,{,pragma}])->hl:Historical_List<T>`
  `where pragma:=localization|representation`
  Creation of a new historical list, i.e. a list which maintains its history over the valid time dimension.
- `insert_element(o:T,validtime:timestamp(*))`
  The object is inserted at the current position in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the list with default valid time timestamp from the instant the operation is performed to forever.
  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `insert_element_after(o:T,position:integer,validtime:timestamp(*))`
  The object is inserted after the argument position in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the list with default valid time timestamp from the instant the operation is performed to forever.
  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `insert_element_before(o:T,position:integer,`
  `                      validtime:timestamp(*))`
  The object is inserted before the argument position in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the list with default valid time timestamp from the instant the operation is performed to forever.
  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `insert_first_element(o:T,validtime:timestamp(*))`

The object is inserted at the first position in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the list with default valid time timestamp from the instant the operation is performed to forever.

$^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `insert_last_element(o:T,validtime:timestamp`$^{(*)}$`)`

  The object is inserted at the last position in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the list with default valid time timestamp from the instant the operation is performed to forever.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_element_at(position:Integer,validtime:timestamp`$^{(*)}$`)`

  The object located at the given position is removed from the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the list.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_first(validtime:timestamp`$^{(*)}$`)`

  The object located at the first position is removed from the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the list.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_last(validtime:timestamp`$^{(*)}$`)`

  The object located at the last position is removed from the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the list.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `replace_element_at(o:T,position:Integer,validtime:timestamp`$^{(*)}$`)`

  The object located at the given position is replaced by the first argument in the version of the historical list corresponding to the valid time argument. If no valid time is given then the object is replaced in the current version of the list.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `retrieve_element_at(position:Integer,validtime:timestamp`$^{(*)}$`)`

```
->{element:T,validtime:Timestamp}
```

The object located at the given position in the version of the historical list corresponding to the valid time argument is returned, with the valid time timestamp. If no valid time is given then the object is returned from the current version of the list.

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

- `retrieve_first(validtime:timestamp`$^{(*)}$`)`

  ```
  ->{element:T,validtime:Timestamp}
  ```

  The object located at the first position in the version of the historical list corresponding to the valid time argument is returned, with the valid time timestamp. If no valid time is given then the object is returned from the current version of the list.

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

- `retrieve_last(validtime:timestamp`$^{(*)}$`)`

  ```
  ->{element:T,validtime:Timestamp}
  ```

  The object located at the last position in the version of the historical list corresponding to the valid time argument is returned, with the valid time timestamp. If no valid time is given then the object is returned from the current version of the list.

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

- `select(predicate:String.validtime:Timestamp`$^{(*)}$`)->l:List<T>`

  ```
  ->{l:List<T>,validtime:Timestamp}
  ```

  It returns a list of objects satisfying the given predicate from the version of the historical list corresponding to the valid time argument, with the valid time timestamp. If no valid time is given then the objects are returned from the current version of the list.

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

Historical_Array<T>.

ODMG arrays are one dimensional arrays of varying size.
TODM redefines all the operations of the ODMG Array<T> type.

- `create(length:integer)->ha:Historical_Array<T>`
  `where pragma:=localization|representation`

Creation of a new historical array of the argument length, i.e. an array which maintains its history over the valid time dimension.

- `insert_element_at(o:T,position:Integer,validtime:timestamp`$^{(*)}$`)`

  The object is inserted at the argument position in the version of the historical array corresponding to the valid time argument. If no valid time is given then the object is inserted in the current version of the array with default valid time timestamp from the instant the operation is performed to forever.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `remove_element_at(position:Integer,validtime:timestamp`$^{(*)}$`)`

  The object located at the given position is removed from the version of the historical array corresponding to the valid time argument. If no valid time is given then the object is removed from the current version of the array.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `replace_element_at(o:T,position:Integer,validtime:timestamp`$^{(*)}$`)`

  The object located at the given position is replaced by the first argument in the version of the historical array corresponding to the valid time argument. If no valid time is given then the object is replaced in the current version of the array with default valid time timestamp from the instant the operation is performed to forever.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

- `retrieve_element_at(position:Integer,validtime:timestamp`$^{(*)}$`)`

  `->{element:T,validtime:Timestamp}`

  The object located at the given position in the version of the historical array corresponding to the valid time argument is returned, with the valid time timestamp. If no valid time is given then the object is returned from the current version of the array.

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

- `resize(new_size:Integer,validtime:Timestamp`$^{(*)}$`)`

  The version of the historical array corresponding to the valid time argument is resized to the argument size. If no valid time is given, then the current version will be resize.

  $^{(*)}$: if the timestamp is not an instant, several versions can be related to the valid time argument.

Historical_String is a type instantiation of Historical_Array<T> with the Character type. And Historical_Bit_String is also a type instantiation but with the Boolean type.

Historical_Structure.

A structure is an unnamed group of elements where each element is a (name,value) pair. TODM redefines all the operations of the ODMG Structure generator type.

- `create([init_list,validtime:Timestamp])->hs:Historical_Structure`

  Creation of a new historical structure, i.e. a structure which maintains its history over the valid time dimension with default valid time timestamp from the instant the operation is performed to forever.
- `delete()`

  Deletion of the historical structure.
- `get_element_value(element,validtime:Timestamp(*))`
  `->{value:Denotable_Object,validtime;Timestamp}`

  This returns the value of the element in the version of the historical structure corresponding to the valid time argument, with the valid time timestamp. If no valid time is given, then the value is returned from the current version of the structure.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument, then the result will be a set of `(value,validtime)` pairs, where each `validtime` timestamp is contained in the argument one.
- `set_element_value(element,value:Denotable_Object,`
  `validtime:Timestamp(*))`

  This sets the value of the element in the version of the historical structure corresponding to the valid time argument. If no valid time is given, then the value is set in the current version of the structure.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `clear_element_value(element,validtime:Timestamp(*))`

  The element of the version of the historical structure corresponding to the valid time argument is cleared. If no valid time is given, then the element is cleared in the current version of the structure.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `clear_all_values(validtime:Timestamp(*))`

  All the elements of the version of the historical structure corresponding to the valid time argument are cleared. If no valid time is given, then all the elements are cleared in the current version of the structure.

  (*): if the timestamp is not an instant, several versions can be related to the valid time argument.
- `copy()->hs:Historical_Structure`

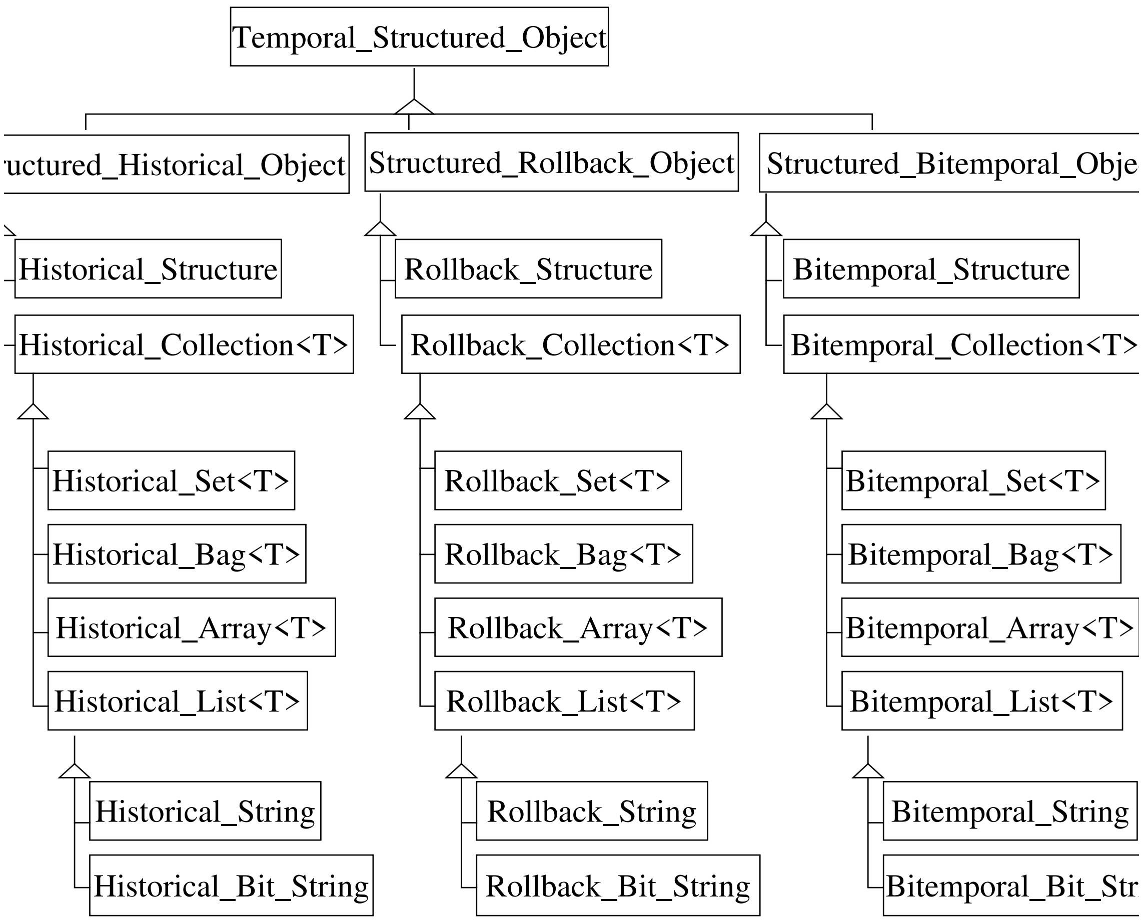
This returns a copy of the historical structure: literals are duplicated. and objects are referenced.

Rollback and Bitemporal structured types.

The following figure, Figure 17, “Temporal Structured Type”, page 131, presents all the temporal structured types.

Rollback and bitemporal types are extended from ODMG-93 in a similar way as historical ones have been described in the above pages. The only difference is that rollback structured objects keep their audit over the transaction time dimension, and that bitemporal structured objects keep their whole history and audit over both the valid time and transaction time dimensions.

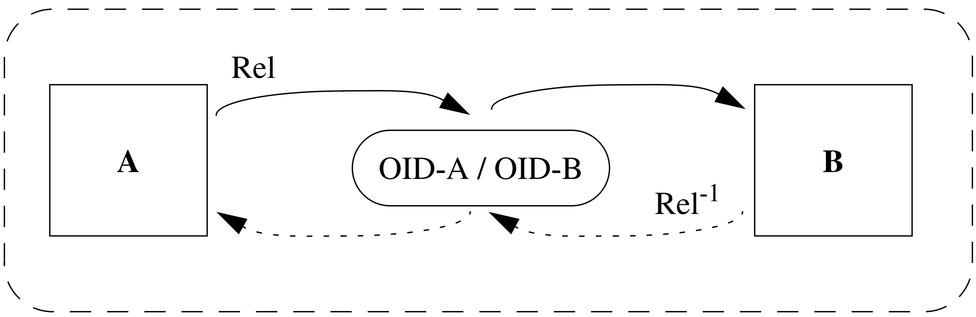
Figure 17. Temporal Structured Type



Temporal relationships and inverse links

In ODMG, and in a simplifying way, a relationship between an object A and an object B can be seen within A as a pointer towards the OID of B, and the reverse link within B as a pointer towards the OID of A. In fact, a relationship and its inverse link are two different ways to access to a same information, as shown by the following schema where the inverse relationship is presented using the $()^{-1}$ notation.

Figure 18. ODMG relationship and inverse



With TODM, temporal relationships and temporal objects are introduced. The following lines are dedicated to a kind of typology of temporal relationships towards temporal objects, and show that inverse relationships can not be defined in all the cases.

For more details regarding the temporal objects, cf. “Temporal Objects”, page 108.

In the following tables, all the different cases of temporal relationships from snapshot object, of relationships from historical, rollback and bitemporal objects towards snapshot, historical, rollback and bitemporal objects are listed.

- Table 11, “Temporal relationships & temporal objects”, page 134
- Table 12, “Temporal relationships & temporal objects (continued)”, page 135

For each case the relationship data is modelled using object OIDs and timestamps when necessary. Information is also given about the inverse relationship.

For instance, let us examine two cases:

- A snapshot relationship from a snapshot object, S1, towards a:
  - snapshot object, S2:

   - the relationship models a relationship without any history, between two objects with no time varying information at the object level.
   - the relationship data can be represented by the couple (OID-S1,OID-S2).
   - the inverse link can exist and has to be defined as a snapshot relationship.
 - historical object, H2 (rollback object, R2; bitemporal object, B2):
   - the relationship models a relationship without any history, from an object with no time varying information at its level and towards and object which keep track of its history over the valid time dimension (transaction time dimension; both time dimensions).
   - the relationship data can be represented by the couple (OID-S1,OID-H2) ((OID-S1,OID-R2); (OID-S1,OID-B2)).
   - the inverse link can not exist within the temporal object since to access a state of this object and then a relationship, a valid time timestamp is necessary (transaction time timestamp; bitemporal timestamp).

- A rollback relationship from a snapshot object, S1, towards a:
 - snapshot object, S2:
   - the relationship models a relationship which keeps track of its evolution over the transaction time dimension, between two objects with no time varying information at the object level.
   - the relationship data can be represented by the triplet (OID-S1,TT,OID-S2).
   - the inverse link can exist and has to be defined as a rollback relationship.
 - historical object, H2 (bitemporal object, B2):
   - the relationship models a relationship which keeps track of its evolution over the transaction time dimension, from an object with no time varying information at its level and towards an object which keeps track of its history over the valid time dimension (over both time dimensions).
   - the relationship data can be represented by the triplet (OID-S1,TT,OID-H2) ((OID-S1,TT,OID-B2)).
   - the inverse link can not exist within the historical (bitemporal) object since it has no way to store information about the relationship over the transaction time dimension.
 - rollback object, R2:
   - the relationship models a relationship which keeps track of its evolution over the transaction time dimension, from an object with no time varying information at its level and towards an object which keeps track of its history over the transaction time dimension.
   - the relationship data can be represented by the triplet (OID-S1,TT,OID-R2).

- the inverse link can exist within the rollback object since the audit of the relationship over the transaction time dimension is included in the audit of the object; the inverse relationship is defined as a classical one and the transaction time support is defined at the object level.

The following tables present all the different cases.

Table 11. Temporal relationships & temporal objects

<table>
<tr><th colspan="2">From / To</th><th>Snapshot
Object: S2</th><th>Historical
Object: H2</th></tr>
<tr><td rowspan="4">Snapshot
Object: S1</td><td>Snapshot
Relationship</td><td>OID-S1, OID-S2
inverse: snapshot</td><td>OID-S1, OID-H2
no inverse</td></tr>
<tr><td>Historical
Relationship</td><td>OID-S1, VT, OID-S2
inverse: historical</td><td>OID-S1, VT, OID-H2
inverse: history at object level</td></tr>
<tr><td>Rollback
Relationship</td><td>OID-S1, TT, OID-S2
inverse: rollback</td><td>OID-S1, TT, OID-H2
no inverse</td></tr>
<tr><td>Bitemporal
Relationship</td><td>OID-S1, BT, OID-S2
inverse: bitemporal</td><td>OID-S1, BT, OID-H2
no inverse</td></tr>
<tr><td colspan="2">Historical Object: H1</td><td>OID-H1, VT, OID-S2
inverse: historical</td><td>OID-H1, VT, OID-H2
inverse: history at object level</td></tr>
<tr><td colspan="2">Rollback Object: R1</td><td>OID-R1, TT, OID-S2
inverse: rollback</td><td>OID-R1, TT, OID-H2
no inverse</td></tr>
<tr><td colspan="2">Bitemporal Object: B1</td><td>OID-B1, BT, OID-S2
inverse: bitemporal</td><td>OID-B1, BT, OID-H2
no inverse</td></tr>
</table>

Table 12. Temporal relationships & temporal objects (continued)

From / To		Rollback Object: R2	Bitemporal Object: B2
Snapshot Object: S1	Snapshot Relationship	OID-S1, OID-R2 no inverse	OID-S1, OID-B2 no inverse
	Historical Relationship	OID-S1, VT, OID-R2 no inverse	OID-S1, VT, OID-B2 no inverse
	Rollback Relationship	OID-S1, TT, OID-R2 inverse: history at object level	OID-S1, TT, OID-B2 no inverse
	Bitemporal Relationship	OID-S1, BT, OID-R2 no inverse	OID-S1, BT, OID-B2 inverse: history at object level
Historical Object: H1		OID-H1, VT, OID-R2 no inverse	OID-H1, VT, OID-B2 no inverse
Rollback Object: R1		OID-R1, TT, OID-R2 inverse: history at object level	OID-R1, TT, OID-B2 no inverse
Bitemporal Object: B1		OID-B1, BT, OID-R2 no inverse	OID-B1, BT, OID-B2 inverse: history at object level

These rules compel the usage of temporal relationships, but they assure a semantic coherency.

Relationships towards states of temporal objects

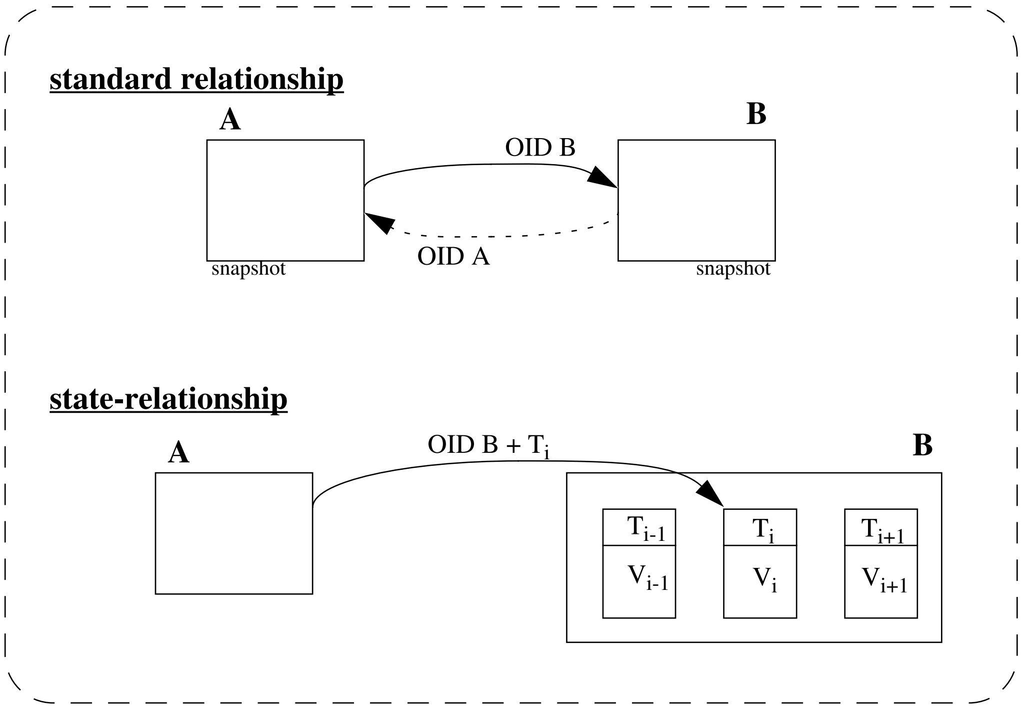
First let us have a look at a small example.
In the medical sector, a treatment can vary over the time to benefit from the last discoveries. Then the treatment can be modelled as an historical object where the different states of the object represent the evolutions or improvements of the treatment. A patient who suffers from a disease will be treated by a treatment, and in fact by a version of the treatment. Then to model the treatment followed by the patient, a relationship from the patient object needs to point out the right treatment, and more precisely the variant of the treatment used by the patient. Here a standard relationship is not sufficient since it will only link the patient towards the historical object treatment, and not to the variant used.
Therefore, to satisfy this need, TODM introduces a new kind of relationship called **state-relationships**.

A state-relationship will be a relationship towards a state of a temporal object, either historical, rollback or bitemporal object. A pointer towards the OID of the temporal object is not sufficient to represent a state-relationship, since it only enables to know the temporal object but not towards which state of the temporal object is intended the state-relationship.
A kind of **TOID**, enabling to distinguish the target state of the temporal object, has to be associated with the OID of the temporal object.
This TOID will be a timestamp. The target state will be the state of the temporal object stored for this timestamp. In order to uniquely select the target state from one of the three types of temporal objects, the TOID should be:

- for historical objects with no overlapping: an instant
- for historical objects with overlapping: a period
- for rollback objects: an instant
- for bitemporal objects with no overlapping of valid time timestamps: a couple of instants
- for bitemporal objects with overlapping of valid time timestamps: a couple composed of a valid time period and of a transaction time instant

The following figure illustates states-relationships: Figure 19, “Standard relationship vs state-relationship”, page 137.

Figure 19. Standard relationship vs state-relationship



A state-relationship does not store the state towards which it has been defined; if it does it would no longer be a kind of relationship but it would rather be a kind of composition with a stored state contained in another object. Then a state-relationship only stores the way to access the target state, i.e. the OID of the temporal object and the TOID for the wanted state. As a standard relationship, a state-relationship has to be traversed to access its target.
When traversing a state-relationship, the state which is valid for the TOID timestamp stored within the state-relationship will be returned. If no state exists for this TOID timestamp, a null state[1] will be returned.

No inverse link exists with state-relationships. State-relationships can only be defined towards temporal objects. As with classical relationships, state-relationships can have different granularities: one to one or one to many state-relationships.

In the following lines the operations provided for the state-relationships are described. They are extensions of standard relationship operations.

one_to_one state-relationship.

1. a null pointer in C++.

- `create(o1:Denotable_Object,o2:Temporal_Object,toid_time:Timestamp)`

  It will create a state-relationship from the object `o1` towards the state of the temporal object `o2`, precisely described by the TOID timestamp 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.

one_to_many state-relationship.

- ```
  create(o1:Denotable_Object,
              Set<(o:Temporal_Object,toid_time:Timestamp)>)
  ```

  It will create a state-relationship from the object `o1` to each element of the set of states given in argument.
- `delete()`

  The state-relationship will be deleted.
- ```
  add_one_to_one(o1:Denotable_Object,o2:Temporal_Object,
                           toid_time:Timestamp)
  ```

  It will add a new state-relationship from the object `o1` to the state of the temporal object `o2`, precisely described by the TOID timestamp argument.
- ```
  remove_one_to_one(o1:Denotable_Object,o2:Temporal_Object,
                           toid_time:Timestamp)
  ```

  This operation will remove the state-relationship from object `o1` to the state of the object `o2`.
- `traverse(from:Denotable_Object)->states:Set<Object_State>`

  This operation will return the set of states from the temporal objects pointed out by the state-relationship.

In addition to these ‘standard’ operations, another one which will enable to retrieve the TOID timestamp could be useful:

- `get_toid_time() -> toid_time:Timestamp`

  This operation returns the TOID timestamp stored within the state-relationship.

On top of the properties of the state-relationships, histories and audits of state-relationships can be defined. As well as with the standard relationships, four specifications for
  ```

each state-relationship can be defined, one snapshot and three temporal ones, to handle the evolution of state-relationship over one or both of the valid time and transaction time dimensions.

Temporal Iterations

Temporal instance properties (attributes and relationships) and temporal objects can be considered as some kinds of black-boxes or containers. They allow to keep the different variants, the different values or states recorded with respect to the valid time, the transaction time or both dimensions.

For each temporal type introduced, TODM provides operations to retrieve the whole history over one or two temporal dimensions. For instance the `get_history`, `get_rollback` and `get_full_history` operations defined on temporal attributes and temporal objects, enable us to retrieve whole histories over valid time, transaction time and bitemporal dimension.

These histories are represented as sets, and therefore can be handled as ODMG sets. All the ODMG collection operations can be applied on them. Iterators can be defined over these sets and cover histories.

For example, assuming that HA is an historical attribute, to retrieve its whole history we can perform:

```
get_history()
   ->history:Set<variant:(value:Literal,validtime:Timestamp)>
```

To check if the attribute owned a specific value, we can perform the `exist?` operation defined on ODMG collection:

```
history.exist?("exists v in this.variant: v.value = val01")
   ->b:Boolean
```

And to create an iteration over the history, we can use the ODMG iterator:

```
history.create_iterator()->i:Iterator
```

Access to temporal characteristics

It would be convenient to have access to the temporal characteristics of an attribute or an object. For instance, it would be helpful to know if an object supports time varying data, and with respect to which time dimension. Then TODM introduces several type operations to satisfy this kind of request. These operations have to be added to the different new types presented by TODM.

- `temporal?()->b:Boolean`

  This operation will return `true` if the attribute or the object supports time varying data. For instance, this operation applied to an object from Historical_Object type will return `true`, whereas applied to an object from Snapshot_Object type the result will be `false`.
- `temporal_dimension()->dimension:`
  `Enumeration(validtime,transactiontime,bitemporal,snapshot)`

  This operation will return the time dimension supported by temporal attributes or objects. For instance, this operation applied to an object from Historical_Object type will return `validtime`, whereas applied to an object from Rollback_Object type the result will be `transactiontime`.

Regarding the transaction time dimension, all the timestamps are periods and expressed in the Gregorian calendar and at the second-sized granularity. On the other hand, the user has to choose the structure, the calendar and the granularity of valid time timestamps. When a new history is created, either an attribute history either an object history, TODM has to provide to the user a way to choose the calendar and granularity for the whole history. By default, if the user does not supply this information, the system will use the Gregorian calendar and the second-sized granularity.

The three following operations allow to retrieve these different characteristics.

- `temporal_structure()->structure:Enumeration(event,state)`

  This operation will return the temporal structure, either event or state, used by temporal attributes or objects. For instance, this operation applied to an object from Rollback_Object type will return `state`.
- `calendar()->cal:Calendar`

  This operation will return the calendar in which are expressed the timestamps used by temporal attributes or objects. For instance, this operation applied to an object from Rollback_Object type will return the `Gregorian` calendar.

- `granule()->gra:Granule`

  This operation will return the granularity at which are expressed the timestamps used by temporal attributes or objects. For instance, this operation applied to an object from Rollback_Object type will return `second`.

The last functions are dedicated to the temporal key features, and allow to retrieve if a key is defined and what kind of it is. For more details regarding the temporal keys, cf. “Temporal constraints of uniqueness”, page 173.

- `temporal_key()->key:`
  `Enumeration(none,point,period,extend,state_point,state_period)`

  This operation will return the kind of temporal key used by temporal attributes or objects. If no temporal key is used, the function will return `none`.
- `overlaps?()->b:Boolean`

  This operation will return `true` if overlapping periods are allowed, and `false` it they are not..

Conversion from Snapshot into Temporal objects

Snapshot objects can contain no time varying instance properties and/or 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 a temporal object. These conversions could be useful in some requests of the query language.

TODM adds three methods to Snapshot_Object type, in order to convert into an Historical_Object, a Rollback_Object or a Bitemporal_Object:

- `create_HistoricalObject()->o:Historical_Object`
- `create_RollbackObject()->o:Rollback_Object`
- `create_BitemporalObject()->o:Bitemporal_Object`

The following paragraphs will deal precisely with the different cases to face.

The same semantics can be defined on snapshot instance properties of objects, as described in the last paragraph of this part.

Snapshot object with no temporal property

The conversion of a snapshot object with no temporal properties into a temporal object implies the creation of the timestamps that will be associated to the new object. If the new object varies on the valid time dimension, then the timestamp will be a period from the instant of the conversion to *forever*. Using a period from *NOW* to *forever*, implies that some information is lost because the values were valid since a previous time that we do not know, but no information is invented. That's why the valid time period associated will not be from *`beginning`* to the relative instant *forever*. If the new object varies on the transaction time dimension, then the timestamp will be a period from the instant of the conversion until the next modification.

Snapshot object with temporal properties

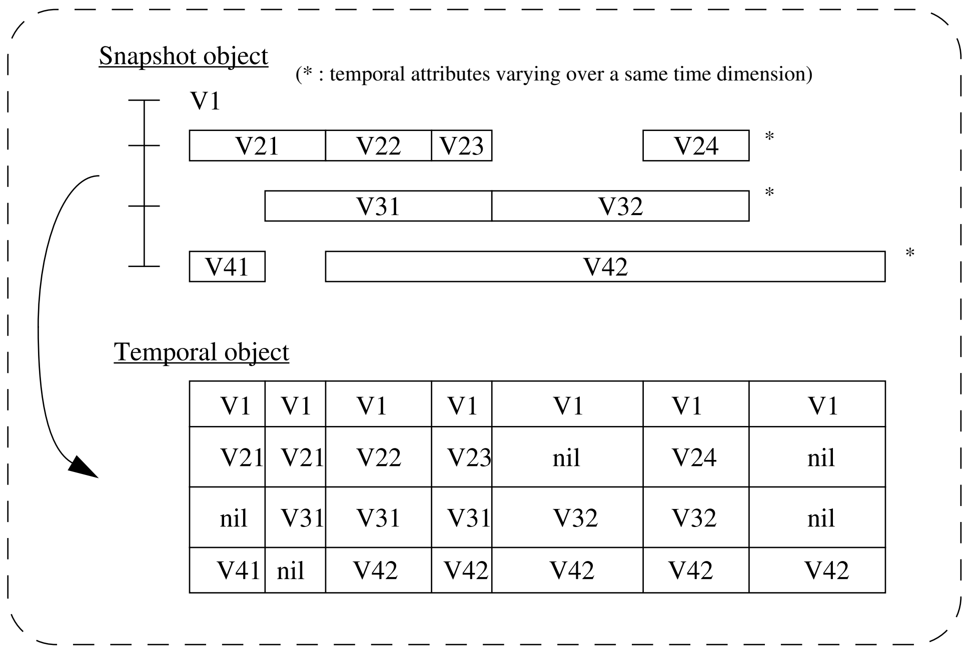
There are two cases. On one hand, snapshot objects with temporal instance properties which vary in the same time dimension. And on the second hand, temporal instance properties which vary in different time dimensions. We call them homogeneous and non-homogeneous snapshot objects.

Homogeneous snapshot objects.

A snapshot object having historical, respectively rollback, respectively bitemporal instance properties can be converted to an historical, respectively rollback, respectively bitemporal object. This functionality corresponds to a propagation of the temporal histories of the instance properties to the object. The attribute level of time varying data is mapped to the object level. The new temporal object is built by time slice, each section corresponding to a modification of one of the temporal attributes. The no time-varying attributes keep the same value during the whole history.

The following figure, Figure 20, “Conversion of a snapshot object with homogeneous temporal attributes into a temporal object”, page 144, presents an example of conversion from a snapshot object with temporal properties to a temporal object. The snapshot object is composed of four attributes, one non time varying and three temporal histories. The history of the new object is made up by the different values of the attributes. The non time-varying attributes are considered to remain constant during the lifespan of the object. Its lifespan represents the period from the earlier boundary of the time-varying attributes to the instant of the conversion operation. The `nil` value which appears on some attributes corresponds to periods where no information is known about the attribute.

Figure 20. Conversion of a snapshot object with homogeneous temporal attributes into a temporal object



Non-homogeneous snapshot objects.

A snapshot object can own temporal properties which vary over different time dimensions. And it would be convenient to be able to convert this kind of snapshot objects into temporal objects.
In non-homogeneous snapshot objects, there can be two or three different time dimensions involved. And the objects can be converted in three different temporal objects. The six following cases summary the problem we face in the twelve different conversion possibilities. Default strategies are given to solve these different situations.

- convert valid time timestamping into transaction time timestamping

  Only current value will be kept, with a transaction time timestamping from the operation time until the next modification.

- convert valid time timestamp into bitemporal timestamp

  A transaction time timestamp has to be associated to the valid time one in order to give a bitemporal timestamp. Default transaction time timestamps from the instant when the conversion operation is performed to the next modification are assigned.

- convert transaction time timestamping into valid time timestamping

  The transaction time timestamps are transformed into valid time timestamps. The timestamps which precede the instant when the operation is performed are kept. The timestamps which overlap the instant when the operation is performed are extended to *forever.*

- convert transaction time timestamping into bitemporal timestamping

  A valid time timestamp has to be associated to the transaction time one in order to give a bitemporal timestamp. Default valid time timestamps from the instant when the conversion operation is performed to *forever* are assigned.

- convert bitemporal timestamping into valid time timestamping

  A bitemporal timestamp is the association of one valid time timestamp and one transaction time timestamp. Then to convert a bitemporal timestamp into a valid time one, the transaction time part of the bitemporal timestamp has to be removed. But only the variants which are current in the database are kept.

- convert bitemporal timestamping into transaction time timestamping

  A bitemporal timestamp is the association of one valid time timestamp and one transaction time timestamp. Then to convert a bitemporal timestamp into a transaction time one, the valid time part of the bitemporal timestamp has to be removed. But only the variants which are present at the instant the operation is performed are kept.

Of course these conversion strategies might not satisfy all the eventual needs and desires of the users. But these strategies can be overridden by a user-defined strategy.

Snapshot Instances Properties

In some request, it could be convenient to convert a snapshot attribute into a temporal one. Then, in a similar way to the conversion of snapshot objects to temporal objects, TODM adds three methods to Snapshot_Attribute type, in order to convert into an Historical_Attribute, a Rollback_Attribute or a Bitemporal_Attribute:

- `create_HistoricalAttribute()->o:Historical_Attribute`
- `create_RollbackAttribute()->o:Rollback_Attribute`
- `create_BitemporalAttribute()->o:Bitemporal_Attribute`

The semantics of these operations are the same with those of the conversion of snapshot objects described in the previous paragraphs.

Of course the same operations can be defined with the snapshot relationships.

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 is present in an earlier state of the database. The vacuuming enables to delete data stored in transaction time audits.

TODM will provide **user-defined vacuuming** at the object/attribute/relationship level. User-defined vacuuming is activated by an exterior event, as for instance a user request. Then the vacuuming at the database level has to be maintained by the application dynamics.

TODM offers vacuuming tools at the object level. Operations, that the user can launch, will perform vacuuming on the object from which they were invoked. Two operations will be available: first to perform a vacuuming using 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 have to be added to the rollback and bitemporal types introduced previously, are described above.

- `vacuum(cut-off_point:Timestamp)`

  This operation activates the vacuuming mechanism. All the data present in the audit before the cut-off point argument are vacuumed. The data of which the transaction time timestamp overlaps the timestamp argument are kept but with a new transaction time timestamp which starts at the cut-off point. The timestamp can be either absolute, with the condition that is precedes the current date, or now-relative, in the past.
- `vacuum(max_nb_values:Integer)`

  This operation activates the vacuuming mechanism. The `max_nb_values` latest values of the audit are kept. The other ones are vacuumed.

The variants which have to be vacuumed, can be deleted without any more caution.

A standard relationship from a temporal object can not have reversee link; cf. “Temporal relationships and inverse links”, page 132. Then the variants can be deleted since they are not involved in any inverse relationship.

A state-relationship which pointed towards a state deleted by a vacuuming operation will now return a null state; cf. “Relationships towards states of temporal objects”, page 136.

Overlapping vs non-overlapping timestamps

Until now, all the temporal types introduced by TODM deal with the time dimensions using timestamps, but no precision has been given regarding these timestamps, assuming they do not allow overlapping. The role of this part is to detail the different timestamping and their consequences on the operations described previously.

NB: Only periods are used in the following lines, but that can be easily generalized to instants because an instant can be considered as a period of zero duration. An instant `I` is equivalent to the period `[I, I + 1 granule)` at the specified granularity.

Transaction time timestamps

Transaction time timestamps are provided by the system, with no user intervention. Transaction time timestamps are periods. They represent the period of time when the data was current in the database. Then there can not exist overlapping periods since two different values for a same piece of information can not be stored at the same time in the database. Transaction time timestamping is linked to the database transactions, and two different transactions can not modify the same data at same time. Cf. “Transaction time timestamping and DBMS transactions”, page 26.

A default transaction time timestamp which is a particular period from the instant when the `set_value` operation is performed until the next modification, is assigned by the system at each time the `set_value` operation is performed. If the new value is different from the current one, then the period timestamp is cut at the instant when the `set_value` operation is performed, and the new value becomes the current one. Whereas, if the new value is in fact the same as the current one, the `set_value` operation has no effect.

In order to maximize the non contiguous periods of transaction time for the same values, transaction time period_sets can be used to store the timestamps. The use of period_sets does not affect the transaction time assigned to `set_value` operations. Only the storing way of timestamps is modified.

The following algorithms describe the behaviour of a rollback attribute when its different operations are performed:

- Figure 21, “set_value operation on Rollback_Attribute”, page 149

A rollback attribute is represented by a collection of `n` couples, each couple `i` is composed of a value `Vi`, and a transaction time timestamp `TTi` which is a period or a period_set.

Figure 21. set_value operation on Rollback_Attribute

$$Att = \{(V_i, TT_i)\}_{i = 1\ldots n}$$

and $set\_value(V)$ with $TT = [NOW, 'UC')$

1. Let R_1 be a set of variants whose value is different from the one provided by `set_value`, and whose transaction time timestamp overlaps the one provided by `set_value`: the timestamp of R_1's elements is set to the difference between the current value and the `set_value` transaction time

$$R_1 = \{(V_i, TT_i);((V_i, TT_i) \in Att) \wedge (V_i \neq V) \\ \wedge (TT_i \text{ OVERLAPS } TT)\}$$

$$R_2 = \{(V_i, TT_i \backslash TT);(V_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$

2. If a variant with the same value as the one provided by `set_value` exists whose transaction time timestamp overlaps the one provided by `set_value`, then its transaction time timestamp is set to the union between its current value and the `set_value` transaction time:

$$If(\exists i, (V_i, TT_i) \in Att) / ((V_i = V) \wedge (TT_i \text{ OVERLAPS } TT))$$

$$Att = Att - \{(V_i, TT_i)\} + \{(V_i, TT_i \cup TT)\}$$

NB: In fact $TT_i \cup TT = TT_i$ since $TT \subset TT_i$

$\{TT = [NOW, 'UC'), TT_i = [\ldots, 'UC')\}$ then $Att = Att$

3. If the transaction time timestamps of all the variants with the same value as the one provided by `set_value` do not overlap the one provided by `set_value`, or if there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with its transaction time timestamp:

$$\mathrm{If}(\forall i)/(((V_i, TT_i) \in \mathrm{Att}) \wedge (V_i = V));TT_i \neg(\text{ OVERLAPS })TT$$
$$\vee (\mathrm{If}(\forall i)/((V_i, TT_i) \in \mathrm{Att}));V_i \neq V$$
$$\mathrm{Att} = \mathrm{Att} + \{(V, TT)\}$$

Figure 22. set_value operation on Rollback_Attribute maximizing non contiguous timestamps

$$\mathrm{Att} = \{(V_i, TT_i)\}_{i = 1 \ldots n}$$

and $\mathrm{set\_value}(V)$ with $TT = [\mathrm{NOW}, '\mathrm{UC}')$

1. Let R_1 be a set of variants whose value is different from the one provided by `set_value`, and whose transaction time timestamp overlaps the one provided by `set_value`: the timestamp of R_1's elements is set to the difference between the current value and the `set_value` transaction time

$$R_1 = \{(V_i, TT_i);((V_i, TT_i) \in \mathrm{Att})$$
$$\wedge (V_i \neq V) \wedge (TT_i \cap TT \neq \emptyset)\}$$
$$R_2 = \{(V_i, TT_i \backslash TT);(V_i, TT_i) \in R_1\}$$
$$\mathrm{Att} = \mathrm{Att} - R_1 + R_2$$

2. If a variant with the same value as the one provided by `set_value` exists, then its transaction time timestamp is set to the union between its current value and the `set_value` transaction time:

$$\mathrm{If}(\exists i, (V_i, TT_i) \in \mathrm{Att})/(V_i = V)$$
$$\mathrm{Att} = \mathrm{Att} - \{(V_i, TT_i)\} + \{(V_i, TT_i \cup TT)\}$$

3. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with its transaction time timestamp:

$$\mathrm{If}(\forall i)/((V_i, TT_i) \in \mathrm{Att});V_i \neq V$$

$$\text{Att} = \text{Att} + \{(\text{V}, \text{TT})\}$$

Figure 23. delete operation on Rollback_Attribute

$$\text{Att} = \{(\text{V}_i, \text{TT}_i)\}_{i = 1 \ldots n}$$

and $\text{delete}()$

1. Let R_1 be a set of variants whose transaction time timestamp overlaps the current time: the timestamp of R_1's elements is cut to the current time

$$R_1 = \{(V_i, TT_i);((V_i, TT_i) \in \text{Att}) \wedge (TT_i \text{ OVERLAPS } TT)\}$$

$$R_2 = \{(V_i, TT_i \backslash [\text{NOW}, \text{NOW} + 1));(V_i, TT_i) \in R_1\}$$

$$\text{Att} = \text{Att} - R_1 + R_2$$

Figure 24. get_value operation on Rollback_Attribute

$$\text{Att} = \{(\text{V}_i, \text{TT}_i)\}_{i = 1 \ldots n}$$

and $\text{get\_value}(\text{TT})$

1. Let Result be a set of couples composed of the variants whose transaction time timestamp intersects the one provided by `get_value` and of the result of this intersection. This set is the result of the operation.

$$R_1 = \{(V_i, TT_i);((V_i, TT_i) \in \text{Att}) \wedge (TT_i \cap TT \neq \emptyset)\}$$

$$\text{Result} = \{(V_i, TT_i \cap TT);(V_i, TT_i) \in R_1\}$$

Valid time timestamps

Valid time timestamps are provided by the user; the system supplies a default one only when the user does not do it. Valid time timestamps represent the time when the fact was, is or will be true in the world modelled within the database. A valid time timestamp can be an instant or a period. In fact, an history can be created using periods or using instants.

Then historical types defined previously can be specialised in two sub-types in order to deal distinctly with historical instant types and historical period types. The same thing can be done with bitemporal types, where the valid time timestamps can also be either periods or instants. And finally, timestamps can be stored using periods or sets of periods, and using instants or sets of instant.

Valid time timestamps are provided by the user. Two strategies can be followed for historical types: allowing timestamps overlapping or forbidding timestamps overlapping. This is another property of the historical types. Then, historical types introduced previously can be specialised in two sub-types, in order to deal with overlapping timestamps or not.

> NB: With overlapping periods the "history" name is no more appropriate. With overlapping time you can store data which are semantically different, as for instance an average on a year does not own the same semantics as an average on a semester, and this is not the following evolution of the same information as understood in "history" term. But in a simpler and non-confusing way, TODM keeps the same name.

The following algorithms describe the behaviour of an historical attribute when its operations are performed:

- Figure 25, "set_value operation on Historical_Attribute forbidding overlapping", page 152
- Figure 26, "set_value operation on Historical_Attribute allowing overlapping", page 154
- Figure 27, "set_value operation on Historical_Attribute maximizing non contiguous timestamps and forbidding overlapping", page 155
- Figure 28, "set_value operation on Historical_Attribute maximizing non contiguous timestamps and allowing overlapping", page 156
- Figure 29, "delete operation on Historical_Attribute", page 156
- Figure 30, "get_value operation on Historical_Attribute", page 157

An historical attribute is represented by a collection of `n` couples, each couple `i` is composed of a value V_i, and a valid time timestamp VT_i which can be a period or a period_set.

Figure 25. set_value operation on Historical_Attribute forbidding overlapping

$$Att = \{(V_i, VT_i)\}_{i = 1 \dots n} \text{ and } set\_value(V, VT)$$

1. Let us consider the variants whose value is different from the one provided by `set_value` but whose valid time intersects the one provided by `set_value`.

1.a. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose valid time timestamp is contained in or is equal to the one provided by `set_value`: all the members of R_1 are removed

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (V_i \neq V) \wedge ((VT \text{ CONTAINS } VT_i) \vee (VT = VT_i))\}$$

$$Att = Att - R_1$$

1.b. Let R1 be a set of variants whose value is different from the one provided by `set_value` and whose valid time timestamp contains the one provided by `set_value`: for each of the memberss of R_1 two smaller timestamps resulting from the difference between the current value and the `set_value` valid time appeared

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (V_i \neq V) \wedge (VT_i \text{ CONTAINS } VT)\}$$

$$R_2 = \{(V_i, [\text{begin}(VT_i), \text{begin}(VT)));(V_i, VT_i) \in R_1\}$$

$$R_3 = \{(V_i, [\text{end}(VT), \text{end}(VT_i)));(V_i, VT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2 + R_3$$

1.c. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose valid time timestamp overlaps the one provided by `set_value`: the timestamp of each member of R_1 is set to the difference between the current value and the `set_value` valid time

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (V_i \neq V) \wedge (VT_i \text{ OVERLAPS } VT)\}$$

$$R_2 = \{(V_i, VT_i \backslash VT);(V_i, VT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$

2. Let us consider the variants with the same value as the one provided by `set_value`.

2.a. Let R_1 be a set of variants whose value is equal to the one provided by `set_value` and whose valid time timestamp is contained in the one provided by `set_value`: the timestamp of each member of R_1 is set to the `set_value` valid time

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (V_i = V) \wedge (VT \text{ CONTAINS } VT_i)\}$$

$$\mathrm{Att} = \mathrm{Att} - \mathrm{R}_1 + \{(\mathrm{V}, \mathrm{VT})\}$$

2.b. If there exists a variant with the same value as the one provided by `set_value` and whose the valid time contains or is equal to the one provided by `set_value`: no new information

$$\mathrm{If}(\exists \mathrm{i}, (\mathrm{V_i}, \mathrm{VT_i}) \in \mathrm{Att}) / ((\mathrm{V_i} = \mathrm{V}) \\ \wedge ((\mathrm{VT_i}\ \mathrm{CONTAINS}\ \mathrm{VT}) \vee (\mathrm{VT_i} = \mathrm{VT})))$$

$$\mathrm{Att} = \mathrm{Att}$$

2.c. Let R_1 be a set of variants whose value is equal to the one provided by `set_value` and whose valid time timestamp overlaps or is adjacent (merges) to the one provided by `set_value`: the union of the timestamps of all the members of R_1 is created.

$$\mathrm{R}_1 = \{(\mathrm{V_i}, \mathrm{VT_i}); ((\mathrm{V_i}, \mathrm{VT_i}) \in \mathrm{Att}) \\ \wedge (\mathrm{V_i} = \mathrm{V}) \wedge ((\mathrm{VT_i}\ \mathrm{OVERLAPS}\ \mathrm{VT}) \vee (\mathrm{VT_i}\ \mathrm{ADJACENT}\ \mathrm{VT}))\}$$

$$\mathrm{Att} = \mathrm{Att} - \mathrm{R}_1 + \{(\mathrm{V}, \bigcup((\mathrm{VT_i} / (\mathrm{V_i}, \mathrm{VT_i}) \in \mathrm{R}_1) \cup \mathrm{VT}))\}$$

2.d. If the valid time timestamps of all the variants with the same value as the one provided by `set_value` do not intersect and are not adjacent to the one provided by `set_value`, then the new value is added to the object with its valid time timestamp:

$$\mathrm{If}(\forall \mathrm{i}) / (((\mathrm{V_i}, \mathrm{VT_i}) \in \mathrm{Att}) \wedge (\mathrm{V_i} = \mathrm{V}); \\ (\mathrm{VT_i} \cap \mathrm{VT} = \emptyset) \wedge \neg(\mathrm{VT_i}\ \mathrm{ADJACENT}\ \mathrm{VT}))$$

$$\mathrm{Att} = \mathrm{Att} + \{(\mathrm{V}, \mathrm{VT})\}$$

3. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with its valid time timestamp:

$$\mathrm{If}(\forall \mathrm{i}) / ((\mathrm{V_i}, \mathrm{VT_i}) \in \mathrm{Att}); \mathrm{V_i} \neq \mathrm{V}$$

$$\mathrm{Att} = \mathrm{Att} + \{(\mathrm{V}, \mathrm{VT})\}$$

Figure 26. set_value operation on Historical_Attribute allowing overlapping

$\mathrm{Att} = \{(\mathrm{V_i}, \mathrm{VT_i})\}_{\mathrm{i} = 1 \ldots \mathrm{n}}$ and $\mathrm{set\_value}(\mathrm{V}, \mathrm{VT})$

The new value is added to the object without any checking since overlapping is allowed. Only the variants with identical timestamp as the one provided by `set_value` need to be removed. Let R_1 be a set of such variants: all the members of R_1 are removed.

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (VT_i = VT)\}$$

$$Att = Att - R_1 + \{(V, VT)\}$$

Figure 27. set_value operation on Historical_Attribute maximizing non contiguous timestamps and forbidding overlapping

$$Att = \{(V_i, VT_i)\}_{i = 1\dots n} \text{ and } set\_value(V, VT)$$

1. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose valid time timestamp intersects the `set_value` timestamp: the timestamp of each member of R_1 is set to the difference between the current value and the `set_value` valid time

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (V_i \neq V) \wedge (VT_i \cap VT \neq \emptyset)\}$$

$$R_2 = \{(V_i, VT_i \backslash VT);(V_i, VT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$

2. If a variant with the same value as the one provided by `set_value` exists, then its valid time timestamp is set to the union between its current value and the `set_value` valid time:

$$If(\exists i, (V_i, VT_i) \in Att)/(V_i = V)$$

$$Att = Att - \{(V_i, VT_i)\} + \{(V_i, VT_i \cup VT)\}$$

3. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with its valid time timestamp:

$$If(\forall i)/((V_i, VT_i) \in Att);V_i \neq V$$

$$Att = Att + \{(V, VT)\}$$

Figure 28. set_value operation on Historical_Attribute maximizing non contiguous timestamps and allowing overlapping

$\mathrm{Att} = \{(V_i, VT_i)\}_{i = 1 \ldots n}$ and $\mathrm{set\_value}(V, VT)$

1. If a variant with the same value as the one provided by `set_value` exists, then its valid time timestamp is set to the union between its current value and the `set_value` valid time:

$$\mathrm{If}(\exists i, (V_i, VT_i) \in \mathrm{Att}) / (V_i = V)$$

$$\mathrm{Att} = \mathrm{Att} - \{(V_i, VT_i)\} + \{(V_i, VT_i \cup VT)\}$$

2. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with its valid time timestamp:

$$\mathrm{If}(\forall i) / ((V_i, VT_i) \in \mathrm{Att});(V_i \neq V)$$

$$\mathrm{Att} = \mathrm{Att} + \{(V, VT)\}$$

Figure 29. delete operation on Historical_Attribute

$\mathrm{Att} = \{(V_i, VT_i)\}_{i = 1 \ldots n}$ and $\mathrm{delete}(VT)$

1. Consider those variants whose valid time intersects the one provided by `delete`.

1.a. Let R_1 be a set of variants whose valid time timestamp is contained in or is equal to the one provided by `delete`: all the members of R_1 are removed

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in \mathrm{Att}) \wedge$$
$$((VT \text{ CONTAINS } VT_i) \vee (VT = VT_i))\}$$
$$\mathrm{Att} = \mathrm{Att} - R_1$$

1.b. Let R_1 be a set of variants whose valid time timestamp contains the one provided by `delete`: for each member of R_1, two smaller timestamps resulting from the difference between the current value and the `delete` valid time appeared

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in \mathrm{Att}) \wedge (VT_i \text{ CONTAINS } VT)\}$$

$$R_2 = \{(V_i, [\text{begin}(VT_i), \text{begin}(VT)));(V_i, VT_i) \in R_1\}$$

$$R_3 = \{(V_i, [\text{end}(VT), \text{end}(VT_i)));(V_i, VT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2 + R_3$$

1.c. Let R_1 be a set of variants whose valid time timestamp overlaps the one provided by `delete`: the timestamp of each member of R_1 is set to the difference between the current value and the `delete` valid time

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (VT_i \text{ OVERLAPS } VT)\}$$

$$R_2 = \{(V_i, VT_i \backslash VT);(V_i, VT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$

NB: When overlapping is allowed, the `delete` operation of historical attribute can take a second argument which is a value. If this value is specified, then only this value will be deleted on the valid time given by `delete`, and not all the values whose valid time timestamp intersects the one provided by `delete`.

Figure 30. get_value operation on Historical_Attribute

$$Att = \{(V_i, VT_i)\}_{i = 1 \ldots n}$$

and $\text{get\_value}(VT)$

1. Let Result be a set of couples composed of the variants whose valid time timestamp intersects the one provided by `get_value` and of the result of this intersection. This set is the result of the operation.

$$R_1 = \{(V_i, VT_i);((V_i, VT_i) \in Att) \wedge (VT_i \cap VT \neq \emptyset)\}$$

$$Result = \{(V_i, VT_i \cap VT);(V_i, VT_i) \in R_1\}$$

Bitemporal timestamps

A bitemporal timestamp is an association of one valid time timestamp and one transaction

time timestamp. As with rollback objects, the transaction time timestamps are provided by the system. And as with historical objects, the valid time timestamps are supplied by the user. Then the properties described with both rollback and historical objects are present here.
Transaction time timestamps are only periods, provided by the system, with no overlapping. Valid time timestamps are either periods either instants. And overlapping timestamps can be allowed or forbidden. Then sub-types of the bitemporal types are introduced in order to deal with those different properties.

The following algorithms describe the behaviour of a bitemporal attribute when its operations are performed:

- Figure 31, “set_value operation on Bitemporal_Attribute forbidding overlapping”, page 158
- Figure 32, “set_value operation on Bitemporal_Attribute allowing overlapping”, page 161
- Figure 33, “delete operation on Bitemporal_Attribute”, page 162
- Figure 34, “get_value operation on Bitemporal_Attribute”, page 163
- Figure 35, “set_value operation on Bitemporal_Attribute maximizing non contiguous timestamps and forbidding overlapping”, page 163
- Figure 36, “set_value operation on Bitemporal_Attribute maximizing non contiguous timestamps and allowing overlapping”, page 167
- Figure 37, “delete operation on Bitemporal_Attribute maximizing non contiguous timestamps and forbidding overlapping”, page 168
- Figure 38, “get_value operation on Bitemporal_Attribute maximizing non contiguous timestamps”, page 170

If the bitemporal attribute uses only periods or instants to store the timestamps, then it is represented by a collection of `n` triplets, each triplet `i` is composed of a value V_i, a valid time timestamp VT_i, and a transaction time timestamp TT_i.
If the bitemporal attribute maximizes the non contiguous timestamps for the same value, then it is represented by a collection of `n` couples, each couple i is composed of a value V_i and a bitemporal timestamp BT_i. Each BT_i is a collection of n_i couples, each couple `j` is composed of a valid time timestamp VT_{ij}, and a transaction time set of timestamps TT_{ij} (for a same valid time timestamps, the transaction time timestamps are grouped in a set of timestamps).

Figure 31. set_value operation on Bitemporal_Attribute forbidding overlapping

$$\mathrm{Att} = \{(V_i, VT_i, TT_i)\}_{i = 1 \ldots n}$$

and $\mathrm{set\_value}(V, VT)$ with $TT = [\mathrm{NOW}, '\mathrm{UC}')$

1. Consider the variants whose value is different from the one provided by `set_value` but whose transaction time intersects the one provided by `set_value`.

1.a. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose transaction time overlaps or is equal to the one provided by `set_value` and whose valid time timestamp is contained in the one provided by `set_value`: those variants are no longer current in the database, their transaction time timestamp is set to the difference between their current value and the `set_value` transaction time

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (V_i \neq V) \wedge (TT_i \cap TT \neq \emptyset) \wedge ((VT \text{ CONTAINS } VT_i) \vee (VT = VT_i))$$
$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$
$$Att = Att - R_1 + R_2$$

1.b. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose transaction time overlaps the one provided by `set_value` and whose valid time timestamp contains the one provided by `set_value`: those variants are no longer current in the database, but for each one two others with the same value and the valid timestamps resulting from the difference between the current value and the `set_value` valid time appear

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (V_i \neq V) \wedge (TT_i \cap TT \neq \emptyset) \wedge (VT_i \text{ CONTAINS } VT)\}$$
$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$
$$R_3 = \{(V_i, [\text{begin}(VT_i), \text{begin}(VT)), TT);(V_i, VT_i, TT_i) \in R_1\}$$
$$R_4 = \{(V_i, [\text{end}(VT), \text{end}(VT_i)), TT);(V_i, VT_i, TT_i) \in R_1\}$$
$$Att = Att - R_1 + R_2 + R_3 + R_4$$

1.c. Let R_1 be a set of variants whose value is different from the one provided by `set_value` and whose transaction time overlaps the one provided `set_value` and whose valid time timestamp overlaps the one provided by `set_value`: those variants are no longer current in the database, but for each one another with the same value and the valid timestamps resulting from the difference between the current value and the set_value valid time appears

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$

$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$R_3 = \{(V_i, VT_i \backslash VT, TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2 + R_3$$

2. Consider the variants with the same value as the one provided by `set_value` and whose transaction time overlaps the one provided by `set_value`.

2.a. Let R_1 be a set of variants whose value is equal to the one provided by `set_value`, whose transaction time overlaps the one provided by `set_value` and whose valid time timestamp is contained in the one provided by `set_value`: those variants are no longer current in the database, but another one with the `set_value` valid time and transaction time appears for each variant

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (V_i = V) \wedge (TT_i \cap TT \neq \emptyset) \wedge (VT \text{ CONTAINS } VT_i)\}$$

$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Obj = Obj - R_1 + R_2 + \{(V, VT, TT)\}$$

2.b. If there exists a variant with the same value as the one provided by `set_value`, whose transaction time overlaps the one provided by `set_value` and whose the valid time contains or is equal to the one provided by `set_value`: then no new information is added

$$If(\exists i, ((V_i, VT_i, TT_i) \in Att) \wedge (V_i = V))/((TT_i \cap TT \neq \emptyset)$$
$$\wedge ((VT_i = VT) \vee (VT_i \text{ CONTAINS } VT)))$$

$$Att = Att$$

2.c. Let R_1 be a set of variants whose value is equal to the one provided by `set_value`, whose transaction time overlaps the one provided by `set_value` and whose valid time timestamp overlaps or is adjacent to the one provided by `set_value`: those variants are no longer current in the database, but another one where all the merged valid time timestamps are gathered appears

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (V_i = V) \wedge (TT_i \cap TT \neq \emptyset)$$

$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$
$$+ \{(V, \bigcup((VT_i / (V_i, VT_i, TT_i) \in R_1) \cup VT), TT)\}$$

2.d. If there does not exist a variant with the same value as the one provided by `set_value`, whose transaction time overlaps the one provided by `set_value` and whose valid time intersects or is adjacent to the `set_value` valid time, then the new value is added to the object with its valid time and transaction time timestamps

$$If(\forall i)/(((V_i, VT_i, TT_i) \in Att) \wedge (V_i = V) \wedge (TT_i \cap TT \neq \emptyset);$$
$$(VT_i \cap VT = \emptyset) \wedge \neg(VT_i \text{ ADJACENT } VT))$$
$$Att = Att + \{(V, VT, TT)\}$$

3. If there does not exist a variant with the same value as the one provided by `set_value`, and there does not exist a variant with the same value as the one provided by `set_value` whose transaction time timestamp overlaps the one provided by `set_value`, then the new value is added to the object with its valid time and transaction time timestamps:

$$If(\forall i, (V_i, VT_i, TT_i) \in Att);(V_i \neq V)$$
$$\wedge If(\forall i, (V_i, VT_i, TT_i) \in Att)/(V_i = V);(TT_i \cap TT = \emptyset)$$
$$Att = Att + \{(V, VT, TT)\}$$

Figure 32. set_value operation on Bitemporal_Attribute allowing overlapping

$$Att = \{(V_i, VT_i, TT_i)\}_{i = 1 \ldots n}$$

and $set\_value(V, VT)$ with $TT = [NOW, 'UC')$

The new value is added to the object without any checking since overlapping is allowed. Only the variants whose valid time timestamp is identical to the one provided by `set_value` and whose transaction time timestamps overlaps the current time, need to be logically deleted, i.e. no longer current. Let R_1 be a set of such variants.

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (VT_i = VT) \wedge (TT_i \cap TT \neq \emptyset)\}$$

$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att + \{(V, VT, TT)\} - R_1 + R_2$$

Figure 33. delete operation on Bitemporal_Attribute

$$Att = \{(V_i, VT_i, TT_i)\}_{i = 1 \ldots n}$$

and $delete(VT)$ with $TT = [NOW, 'UC')$

1. Consider those variants whose transaction time overlaps the current time and whose valid time intersects the one provided by `delete`.

1.a. Let R_1 be a set of variants whose transaction time overlaps the current time and valid time timestamp is contained in or is equal to the one provided by `delete`: those variants are no longer current in the database

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (TT_i \text{ OVERLAPS NOW})$$
$$\wedge ((VT \text{ CONTAINS } VT_i) \vee (VT = VT_i))\}$$
$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2$$

1.b. Let R_1 be a set of variants whose transaction time overlaps the current time and whose valid time timestamp contains the one provided by `delete`: those variants are no longer current in the database, but for each one of them two others with the same value and the valid timestamps resulting from the difference between the current value and the `delete` valid time appear

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att)$$
$$\wedge (V_i \neq V) \wedge (TT_i \text{ OVERLAPS NOW}) \wedge (VT_i \text{ CONTAINS } VT)\}$$
$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$R_3 = \{(V_i, [begin(VT_i), begin(VT)), TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$R_4 = \{(V_i, [end(VT), end(VT_i)), TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2 + R_3 + R_4$$

1.c. Let R_1 be a set of variants whose transaction time overlaps the current time and whose valid time timestamp overlaps the one provided by `delete`: those variants are no longer current in the database, but for each one of them another with the same value and the valid timestamps resulting from the difference between the current value and the `delete` valid time appears

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att) \wedge (V_i \neq V) \wedge (TT_i \text{ OVERLAPS NOW}) \wedge (VT_i \text{ OVERLAPS } VT)\}$$

$$R_2 = \{(V_i, VT_i, TT_i \backslash TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$R_3 = \{(V_i, VT_i \backslash VT, TT);(V_i, VT_i, TT_i) \in R_1\}$$

$$Att = Att - R_1 + R_2 + R_3$$

NB: When overlapping is allowed, the `delete` operation of bitemporal attribute can take a second argument which is a value. If this value is specified, then only this value will be logically deleted on the valid time given by `delete`, and not all the values whose valid time timestamp intersects the one provided by `delete`.

Figure 34. get_value operation on Bitemporal_Attribute

$$Att = \{(V_i, VT_i, TT_i)\}_{i = 1 \ldots n}$$

and $get\_value(VT, TT)$

1. Let Result be a set of triplets composed of the variants whose valid time and transaction time timestamps intersects the ones provided by `get_value`, along with the result of these intersections. This set is the result of the operation.

$$R_1 = \{(V_i, VT_i, TT_i);((V_i, VT_i, TT_i) \in Att) \wedge (VT_i \cap VT \neq \emptyset) \wedge (TT_i \cap TT \neq \emptyset)\}$$

$$Result = \{(V_i, VT_i \cap VT, TT_i \cap TT);(V_i, VT_i, TT_i) \in R_1\}$$

Figure 35. set_value operation on Bitemporal_Attribute maximizing non contiguous timestamps and forbidding overlapping

$$Att = \{(V_i, \{(VT_{ij}, TT_{ij})\})\}_{i = 1\ldots n, j = 1\ldots n_i}$$

and $set\_value(V, VT)$ with $TT = [NOW, 'UC')$

1. Let R_1 be a set of variants whose value is different from the one provided by `set_value`:

$$R_1 = \{(V_i, BT_i);((V_i, BT_i) \in Att) \wedge (V_i \neq V)\}$$

1.a. For each of members of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the one provided by `set_value`, and whose valid time contains the one provided by `set_value`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i) \\ \wedge (TT_j \cap TT \neq \emptyset) \wedge (VT_j \text{ CONTAINS } VT)\}$$

Let R_{3i} be the set of the members of R_{2i} whose transaction time is set to the difference between its current value and the `set_value` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_{5i} and R_{5i} be the sets of the members of R_{3i} with their with their valid timestamps set to the difference between their current value and the `set_value` valid time:

$$R_{4i} = \{([begin(VT_j), begin(VT)), TT);(VT_j, TT_j) \in R_{2i}\}$$

$$R_{5i} = \{([end(VT), end(VT_j)), TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_6 be the set of the members of R_1 whose bitemporal timestamp is modified to take into account the above mentioned sets:

$$R_6 = \{(V_i, BT_i - R_{2i} + R_{3i} + R_{4i} + R_{5i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_6$$

1.b. For each one of the members of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the one provided by `set_value`, and whose valid time is contained or is equal to the one provided by `set_value`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i)$$

Let R_{3i} be the set of the members of R_{2i} with their transaction time set to the difference between their current value and the `set_value` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_4 be the set of the members of R_1 whose bitemporal timestamp is modified to take into account the above mentioned sets:

$$R_4 = \{(V_i, BT_i - R_{2i} + R_{3i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_4$$

1.c. For each one of the members of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the one provided by `set_value`, and whose valid time overlaps the one provided by `set_value`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i) \wedge (TT_j \cap TT \neq \emptyset) \wedge (VT_j \text{ OVERLAPS } VT)\}$$

Let R_{3i} be the set of the members of R_{2i} with their transaction time set to the difference between their current value and the `set_value` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_{4i} be the set of the members of R_{3i} with their valid time set to the difference between their current value and the `set_value` valid time:

$$R_{4i} = \{(VT_j \backslash VT, TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_5 be the set of the members of R_1 whose bitemporal timestamp is modified to take into account the above mentioned sets:

$$R_5 = \{(V_i, BT_i - R_{2i} + R_{3i} + R_{4i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_5$$

2. If a variant with the same value as the one provided by `set_value` exists:

$$\mathrm{If}(\exists k, (V_k, BT_k) \in \mathrm{Att}) / ((V_k = V))$$

2.a. If there exists a bitemporal timestamp of variant (V_k,BT_k) whose transaction time overlaps the one provided by `set_value` and whose valid time contains or is equal to the one provided by `set_value`: then nothing happens in this case.

$$\mathrm{Att} = \mathrm{Att}$$

2.b. If there exists a bitemporal timestamp of variant (V_k,BT_k) whose transaction time overlaps the one provided by `set_value` and whose valid time merges the one provided by `set_value`: then this timestamp will be no longer current in the database, but another one which gathers the intersected valid time timestamps appears.

$$\mathrm{If}(\exists j, (VT_j, TT_j) \in BT_k) / ((TT_j \cap TT \neq \emptyset) \wedge (VT_j \text{ MERGES } VT))$$

$$\begin{aligned}\mathrm{Att} &= \mathrm{Att} - \{(V_k, BT_k)\} \\ &+ \{(V_k, BT_k - \{(VT_j, TT_j)\} \\ &+ \{(VT_j, TT_j \backslash TT), (VT_j \cup VT, TT)\})\}\end{aligned}$$

2.c. If there exists a bitemporal timestamp of variant (V_k,BT_k) whose transaction time overlaps the one provided by `set_value` and whose valid time is contained in the one provided by `set_value`: then this timestamp will be no longer current in the database, but another one with the `set_value` valid time timestamp appears.

$$\begin{aligned}&\mathrm{If}(\exists j, (VT_j, TT_j) \in BT_k) / ((TT_j \cap TT \neq \emptyset) \\ &\wedge (VT \text{ CONTAINS } VT_j))\end{aligned}$$

$$\begin{aligned}\mathrm{Att} &= \mathrm{Att} - \{(V_k, BT_k)\} \\ &+ \{(V_k, BT_k - \{(VT_j, TT_j)\} + \{(VT_j, TT_j \backslash TT), (VT, TT)\})\}\end{aligned}$$

2.d. If there exists a bitemporal timestamp whose valid time is equal to the one provided by `set_value` and whose transaction does not overlaps the one provided by `set_value`: then the transaction time timestamps are gathered

$$\mathrm{If}(\exists j, (VT_j, TT_j) \in BT_k) / ((TT_j \cap TT = \emptyset) \wedge (VT_j = VT))$$

$$\begin{aligned}\mathrm{Att} &= \mathrm{Att} - \{(V_k, BT_k)\} \\ &+ \{(V_k, BT_k - \{(VT_j, TT_j)\} + \{(VT_j, TT_j \cup TT)\})\}\end{aligned}$$

2.e. If there does not exist a bitemporal timestamp of variant (V_k,BT_k) whose transaction time overlaps the one provided by `set_value` and whose valid time intersects the one provided by `set_value`, and there does not exist a bitemporal timestamp whose valid time is equal to the one provided by `set_value` and whose transaction does not overlaps the one provided by `set_value`: then the `set_value` timestamp is added.

$$\begin{array}{l} \mathrm{If}(\neg(\exists j, (VT_j, TT_j) \in BT_k)) \\ /((TT_j \cap TT \neq \emptyset) \wedge (VT_j \cap VT \neq \emptyset)) \\ \wedge \\ \mathrm{If}(\neg(\exists j, (VT_j, TT_j) \in BT_k)) \\ /((TT_j \cap TT = \emptyset) \wedge (VT_j = VT)) \end{array}$$

$$Att = Att - \{(V_k, BT_k)\} + \{(V_k, BT_k + \{(VT, TT)\})\}$$

3. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with the `set_value` bitemporal timestamp:

$$\mathrm{If}(\forall i)/((V_i, BT_i) \in Att);(V_i \neq V)$$

$$Att = Att + \{(V, \{(VT, TT)\})\}$$

Figure 36. set_value operation on Bitemporal_Attribute maximizing non contiguous timestamps and allowing overlapping

$$Att = \{(V_i, \{(VT_{ij}, TT_{ij})\})\}_{i = 1 \dots n, j = 1 \dots n_i}$$

and $\mathrm{set\_value}(V, VT)$ with $TT = [\text{'NOW'}, \text{'UC'})$

1. If a variant with the same value as the one provided by `set_value` one exists:

$$\mathrm{If}(\exists k, (V_k, BT_k) \in Att)/(V_k = V)$$

1.a. If there exists a bitemporal timestamp of variant (V_k,BT_k) whose transaction time intersects the one provided by `set_value`: then the valid time timestamps are gathered.

$$\mathrm{If}(\exists j, (VT_j, TT_j) \in BT_k)/(TT_j \cap TT \neq \emptyset)$$

$$\mathrm{Att} = \mathrm{Att} - \{(V_k, BT_k)\}$$
$$+ \{(V_k, BT_k - \{(VT_j, TT_j)\} + \{(VT_j, TT_j \backslash TT), (VT_j \cup VT, TT)\})\}$$

1.b. If there does not exist a bitemporal timestamp of (V_k, BT_k) whose transaction time overlaps the one provided by `set_value`: then the set_value timestamp is added.

$$\mathrm{If}(\forall j)/((VT_j, TT_j) \in BT_k);(TT_j \cap TT = \emptyset)$$

$$\mathrm{Att} = \mathrm{Att} - (V_k, BT_k) + \{(V_k, BT_k + \{(VT, TT)\})\}$$

2. If there does not exist a variant with the same value as the one provided by `set_value`, then the new value is added to the object with the `set_value` bitemporal timestamp:

$$\mathrm{If}(\forall i)/((V_i, BT_i) \in \mathrm{Att});(V_i \neq V)$$

$$\mathrm{Att} = \mathrm{Att} + \{(V, \{(VT, TT)\})\}$$

Figure 37. delete operation on Bitemporal_Attribute maximizing non contiguous timestamps and forbidding overlapping

$$\mathrm{Att} = \{(V_i, \{(VT_{ij}, TT_{ij})\})\}_{i = 1 \ldots n, j = 1 \ldots n_i}$$

and $\mathrm{delete}(VT)$ with $TT = [\mathrm{NOW}, '\mathrm{UC}')$

1. Let R_1 be the set of variants:

$$R_1 = \{(V_i, BT_i);((V_i, BT_i) \in \mathrm{Att})\}$$

1.a. For each member of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the current time, and whose valid time contains the one provided by `delete`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i)$$
$$\wedge (TT_j \text{ OVERLAPS NOW}) \wedge (VT_j \text{ CONTAINS } VT)\}$$

Let R_1 be the set of the members of R_{2i} with their transaction time set to the difference between their current value and the `delete` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_{4i} and R_{5i} be the sets of the members of R_{3i} with their valid time set to the difference between their current value and the `delete` valid time:

$$R_{4i} = \{([begin(VT_j), begin(VT)), TT);(VT_j, TT_j) \in R_{2i}\}$$

$$R_{5i} = \{([end(VT), end(VT_j)), TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_6 be the set of the members of R_1 with their bitemporal timestamp modified to take into account the above mentioned sets:

$$R_6 = \{(V_i, BT_i - R_{2i} + R_{3i} + R_{4i} + R_{5i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_6$$

1.b. For each member of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the current time, and whose valid time is contained or is equal to the one provided by `delete`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i)$$
$$\wedge (TT_i \text{ OVERLAPS NOW}) \wedge ((VT \text{ CONTAINS } VT_j) \vee (VT = VT_j))\}$$

Let R_{3i} be the set of the members of R_{2i} with their transaction time set to the difference between their current value and the `delete` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_4 be the set of the members of R_1 with their bitemporal timestamp modified to take into account the above mentioned sets:

$$R_4 = \{(V_i, BT_i - R_{2i} + R_{3i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_4$$

1.c. For each member of R_1, let R_{2i} be the set of bitemporal timestamps whose transaction time overlaps the current time, and whose valid time overlaps the one provided by `delete`:

$$R_{2i} = \{(VT_j, TT_j);((V_i, BT_i) \in R_1) \wedge ((VT_j, TT_j) \in BT_i) \wedge (TT_j \cap TT \neq \emptyset) \wedge (VT_j \text{ OVERLAPS } VT)\}$$

Let R_{3i} be the set of the members of R_{2i} with their transaction time set to the difference between their current value and the `delete` transaction time:

$$R_{3i} = \{(VT_j, TT_j \backslash TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_{4i} be the set of the members of R_{3i} with their valid time set to the difference between their current value and the `delete` valid time:

$$R_{4i} = \{(VT_j \backslash VT, TT);(VT_j, TT_j) \in R_{2i}\}$$

Let R_5 be the set of the members of R_1 with their bitemporal timestamp modified to take into account the above mentioned sets:

$$R_5 = \{(V_i, BT_i - R_{2i} + R_{3i} + R_{4i});(V_i, BT_i) \in R_1\}$$

$$Att = Att - R_1 + R_5$$

Figure 38. get_value operation on Bitemporal_Attribute maximizing non contiguous timestamps

$$Att = \{(V_i, \{(VT_{ij}, TT_{ij})\})\}_{i = 1\ldots n, j = 1\ldots n_i}$$

and $get\_value(VT, TT)$

1. Let Result be a set of triplets compound of the variants whose valid time and transaction time timestamps intersects the ones provided byt `get_value` and of along with the result of these intersections. This set is the result of the operation.

$$R_1 = \{(V_i, VT_j, TT_j);((V_i, BT_i) \in Att) \wedge ((VT_j, TT_j) \in BT_i) \wedge (VT_j \cap VT) \wedge (TT_j \cap TT)\}$$

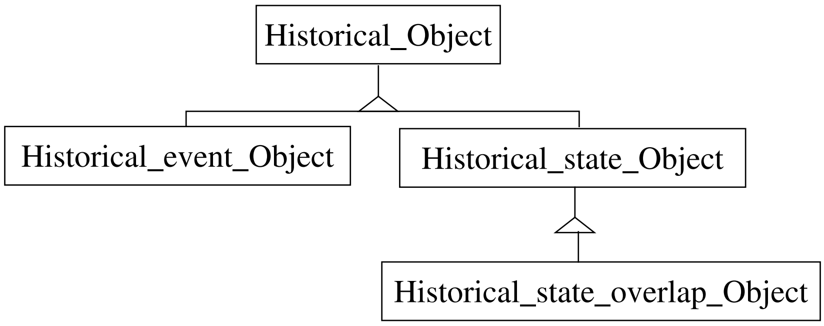
$$Result = \{(V_i, VT_j \cap VT, TT_j \cap TT);(V_i, VT_j, TT_j) \in R_1\}$$

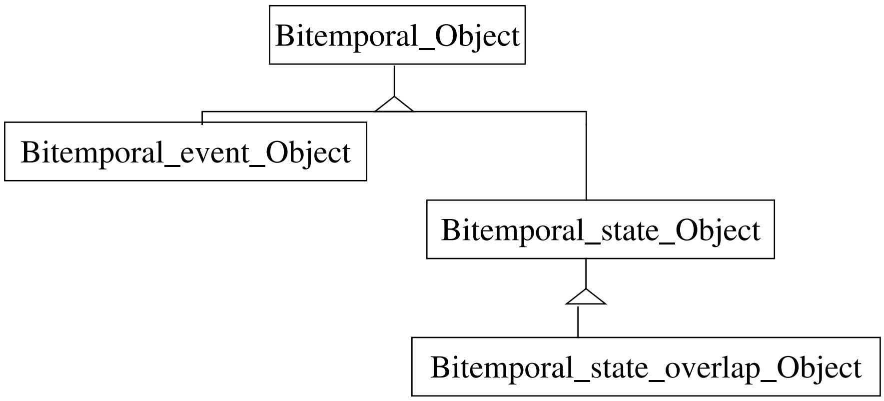
Summary

The different properties of temporal types varying over the valid time dimension introduce several specifications of the first-defined historical and bitemporal types of the TODM. On the next page, Figure 39, “Historical and Bitemporal types hierarchy”, page 172, a simplified hierarchy of the historical and bitemporal types is described. This hierarchy can be applied either to object property types, or to object type.

The `set_value` operation of temporal attributes can be mapped to the `set_state` operation of temporal objects.

Figure 39. Historical and Bitemporal types hierarchy





Temporal constraints of uniqueness

As ODMG-93 defines keys in the type characteristics, TODM introduces temporal keys. The different keys which represent constraints of uniqueness are described below.

The following notations are used in the different formula definitions:

- $I(Att) = \{$bag of instants contained in the temporal attribute$\}$
- $P(Att) = \{$bag of periods contained in the temporal attribute$\}$
- $I(O) = \{$bag of instants contained in the temporal object$\}$
- $P(O) = \{$bag of periods contained in the temporal object$\}$
- $V(Att, t) =$ value of the temporal attribute at timestamp t
- $V(O, t) =$ state of the temporal object at timestamp t
- $E(O) = \{$extent of O (all the objects of the same type as O)$\}$
- $O(Att) =$ object containing the attribute
- $O . Att =$ attribute of the object O

The constraints are listed down.

- **intra-object point**: timestamp overlapping is not allowed for the same object.

$$(\forall i_1 \in I(O), \neg\exists i_2 \in I(O)) / ((i_1 = i_2) \wedge (V(O,i_1) \neq V(O,i_2)))$$

NB: This is the default behavior for any object defined to have valid time characteristics.

- **intra-object period**: timestamp overlapping is allowed for the same object, but for any period only one state is allowed.

$$(\forall p_1 \in P(O), \neg\exists p_2 \in P(O)) / ((p_1 = p_2) \wedge (V(O,p_1) \neq V(O,p_2)))$$

NB: This is the characteristics of valid time history allowing overlapping timestamps.

- **intra-attribute point**: timestamp overlapping is not allowed for the same attribute/relationship.

$$(\forall i_1 \in I(Att), \neg\exists i_2 \in I(Att))/((i_1 = i_2) \wedge (V(Att,i_1) \neq V(Att,i_2)))$$

NB: This is the default behavior for any attribute/relationship defined to have valid time characteristics.

- **intra-attribute period**: timestamp overlapping is allowed for the same attribute/relationship, but for any period only one value is allowed.

$$(\forall p_1 \in P(Att), \neg\exists p_2 \in P(Att))/((p_1 = p_2) \wedge (V(Att,p_1) \neq V(Att,p_2)))$$

NB: This is the characteristics of valid time history allowing overlapping timestamps.

- **inter-object point**: two distinct objects can not have the same state at any time point. Note that the intra-object point constraint must be satisfied in this case too.

$$(\forall i \in I(O_1), \neg\exists O_2 \in E(O_1))/(V(O_1,i) = V(O_2,i))$$

- **inter-object period**: two distinct objects can have the same state for overlapping periods, but not for exactly the same period.

$$(\forall p \in P(O_1), \neg\exists O_2 \in E(O_1))/(V(O_1,p) = V(O_2,p))$$

- **inter-attribute point**: two distinct objects can not have the same value in this attribute/relationship at any time point. Note that the intra-attribute point constraint must be satisfied in this case too.

$$(\forall i \in I(Att), \neg\exists O_2 \in E(Obj(Att)))/(V(O_2.\ Att,i) = V(Att,i))$$

- **inter-attribute period**: two distinct objects can have the same value in this attribute/relationship for overlapping periods, but not for exactly the same period.

$$(\forall p \in P(Att), \neg\exists O_2 \in E(Obj(Att)))/(V(O_2.\ Att,p) = V(Att,p))$$

- **extent-wide object point**: two distinct objects can not have the same state, with regards to all the time points. Note that the inter-object point constraint must be satisfied in this case too.

$$(\forall i_1 \in I(O_1), \neg\exists O_2 \in E(O_1)) / (\exists i_2 \in I(O_2), (V(O_1,i_1) = V(O_2,i_2)))$$

- **extent-wide attribute point**: two distinct objects can not have the same value in this attribute/relationship, with regards to all the time points. Note that the inter-attribute point constraint must be satisfied in this case too.

$$(\forall i_1 \in I(Att), \neg\exists O_2 \in E(Obj(Att))) /$$
$$(\exists i_2 \in I(O_2.\ Att), (V(Att,i_1) = V(O_2.\ Att,i_2)))$$

NB: All the keys defined on attribute are in fact keys which are valid for temporal instance properties, either temporal attributes or temporal relationships.

Implementation forecasts

This document is not dedicated to the implementation task, but within this paragraph few words are given regarding the way of checking the constraints described above.

Regarding intra keys, either on objects or on instance properties, when a `set_value` operation is performed, a checking of the already present values and timestamps in this history is necessary.
This kind of constraint is immediately checkable, at the `set_value` performance.

Regarding the inter key behaviour, the checking is more tricky since it has to look not only in its history but also in the histories of the other objects of the same type, i.e. in all the objects contained in the extent.
It is possible for an object to access its extent using the meta-schema of the database.
On the other hand, according to ODMG, it is not possible for an instance property to access the object which contains it. Here appears another specific property of the temporal attribute that TODM introduced: a temporal instance property knows and is able to access its container object. Therefore, to check inter-attribute keys, the temporal attribute first access its container, then the extent of this object, and after that it can reach the temporal attributes of all the objects contained in the extent.
These constraints have to be checked only at the end of database transactions, since they can involve several objects which may be modified in a transaction.

Extrapolation of discrete attributes

We can classify the attribute histories over valid time dimension in two categories: **discrete history** and **continuous history**. With continuous histories, for any timestamp contained in the attribute lifespan[1], there is a stored value. Whereas with discrete histories, there are some gaps in the history and there does not exist a stored value for each timestamp contained in the attribute lifespan.

Regarding discrete histories and requests of values over timestamps where there is no stored value in the history, there are two policies.

The first one is to return nothing since no value is known. This is the simplest way. TSQL2 and SQL/Temporal follow this policy, and return no tuple since no tuple satisfies the query over the requested timestamp. With the temporal attribute types defined by TODM, this policy is followed, since the nil value literal will be returned if no value is stored over the requested timestamp.

The second policy is to calculate a value using the sample already stored for the attribute. This is called **extrapolation**. Simple or more complex methods can be used to extrapolate values. For instance, if no value is known then the last value before the requested timestamp is returned, or the value known at the closest timestamp is returned. For numerical attributes, the extrapolated value can be calculated using a mean or a barycenter for instance. or another statistical or probabilistic function.

TODM does not provide extrapolation features. But in the following lines, few recommendations are given to users who would like to deal with extrapolated attributes.

In fact, to represent extrapolated attributes, a solution can be the introduction of a sub-type for each temporal attribute type defined over valid time dimension: for instance **Historical_Extrapol_Attribute** and **Bitemporal_Extrapol_Attribute**. These types have to override the `get_value` operations of their super-type in order to return extrapolations of unknown values:

- Historical_Extrapol_Attribute

  `get_value()->value:Literal`

  `get_value(validtime:Timestamp)->value:Literal`

- Bitemporal_Extrapol_Attribute

  `get_value()->value:Literal`

1. The lifespan of an attribute is defined from its earliest value to its latest value of its history.

```
get_value(validtime:Timestamp)->value:Literal
get_value(validtime:Timestamp,transactiontime:Timestamp)
                ->value:Literal
```

These operations have the same semantics as the operations of Historical_Attribute and Bitemporal_Attribute types, except that if no value is stored for the requested timestamp then a value will be extrapolated.

The extrapolation methods, which can be simple or complex, are included in the redefinition of the `get_value` operations.

Extrapolation over the transaction time dimension has no sense. Since the transaction time timestamps, managed by the system, are continuous. There is always something in the database, even if it is an unknown value.

Extrapolation is particular to and defined only for attributes. It can not be extended to relationships nor to objects.

Summary

In this chapter, TODM introduced new structures to manage data varying over one or both of valid time and transaction time dimensions. New types are presented as an extension o the concepts and types shown in the model of ODMG-93.

The temporal features are introduced at both instance property and object levels. Temporal attributes, temporal relationships and temporal objects have been specified in details. Some algorithms have been suggested for the main operations of the temporal types introduced.
Functionalities specific to temporal databases, as for instance vacuuming mechanism have also been described.

In the next chapter, an approach which is more dedicated to the design and the implementation of TODM will be explained. And finally in the chapter height, it will be shown that the temporal support provided by TODM meets the requirements of the previous analysis of the project.

CHAPTER 7

Design of Temporal Support

In this chapter, the temporal support is tackled from an implementation point of view.

The collaboration between TODM and TODL are studied, and some examples of mapping from TODL statements to TODM structures are presented.

As defined and used in the previous chapters of this document, historical entities will allow to maintain the history over the valid time dimension, rollback entities over the transaction time dimension, and bitemporal entities over both valid time and transaction time dimensions.

Design of Temporal Attributes

In ODMG-93, "an attribute takes as its value a literal or a set of literals (actually any structured type containing only literals)". In the following lines, `Literal` will be used to represent any value stored within an attribute [1].

According to ODMG-93, it is not possible for an attribute to access the object which contains it. But in TODM, we need this feature as for instance in order to check the constraints of uniqueness defined on attributes and applied inter objects.
Then one main characteristic of temporal attribute is that it is linked to the object which contains it. This is shown in the following interface of **Temporal_Attribute** thanks to the relationship towards a snapshot object which can contains temporal as well as classical instance properties.

```
interface Temporal_Attribute : Attribute
{
     relationship Snapshot_Object belongs_to;
...
}
```

Then, Temporal_Attribute has to be specialised in order to deal with the different time dimensions. The interface of **Historical_Attribute** is presented below. As already mentioned the attributes are defined in ODMG-93 to take their value as literals, therefore historical attribute will maintain an history of literal values.

```
interface Historical_Attribute : Temporal_Attribute
{
     attribute List< Structure<   value: Literal,
                                  vt : Timestamp > > variants;
     operation set_value(Literal[,Timestamp]);
     operation delete([Timestamp]);
     operation get_value([Timestamp]) -> {Literal,Timestamp};
```

1. In the drafts of ODMG 2.0, an attribute's value is either a literal or an object identifier. An object-valued attribute enables to refer to an object, but without expectation of inverse link or referential integrity which are only maintain within relationships. The implementation of temporal object-valued attributes will be quite similar to temporal attributes presented here, with the only difference that instead of storing *Literal* it will be *Denotable_Object*.

```
     operation get_history() -> Set<{Literal,Timestamp}>;
...
}
```

NB: Some other useful operations can be added to the interface, as for instance an operation of initialisation which takes in argument a set of variants and which resets the historical attribute with the variants given in argument.

Of course, with similar interfaces, rollback and bitemporal attributes can be defined taking into account the time dimensions over which they vary.

Sub-types also appear due to valid time timestamps characteristics. Valid time timestamps can be either instants or periods, and overlapping can be allowed or not. Then to represent instant valid time timestamps, **Historical_event_Attribute** is introduced. Similarly, **Historical_state_Attribute** and **Historical_state_overlap_Attribute** are introduced for period valid time timestamps forbidding and allowing overlapping timestamps.
Of course the same kind of hierarchy appears for bitemporal attributes where valid time timestamps own the same characteristics.
cf. Figure 40, “Temporal Attributes Hierarchy, Example 1”, page 183.

To deal with an historical attribute which takes its value among the integer domain, one can think that this is just an instantiation of Historical_Attribute type. Since `Integer` type is a sub-type of `Literal`. But with this solution, an historical attribute can own values which are from different types, the only constraint is that all the values have to be some literals. This is not a good thing, since maintaining an history of an attribute is to follow the changement of value of this attribute, but the values are all of the same type.
Then to deal with an historical attribute whose values are integer ones, TODM will introduce **Historical_Integer_Attribute**, as a sub-type of Historical_Attribute:

```
interface Historical_Integer_Attribute : Historical_Attribute
{
     operation set_value(i:Integer[,vt:Timestamp])
          { Historical_Attribute::set_value(i,vt); };
...
}
```

Within Historical_Integer_Attribute which inherits from Historical_Attribute, the operations are overridden to check that integer values are handled and nothing else.

Of course TODM can not provide all the types for all the possible user needs. In fact this kind of inheritance will be generated within TODL processus to create all the necessary types in order to represent the temporal attributes defined by the user. This will result in a quite complex and dense hierarchy of types, but this minimizes the use of templates and then store the maximum of information within the meta-schema as possible.

The following example will illustrate the definition and use of temporal attributes.

Example 1

This example is extracted from TODL document, and more precisely from the Delta application example.

This is the creation of a type Product, with three attributes: one standard and two temporal ones, more precisely two historical attributes using periods at the day-sized granularity:

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

This TODL statement will be mapped to TODM types, as following:

```
class Historical_state_Float_Attribute : Historical_state_Attribute
{
     Historical_state_Float_Attribute(Calendar c, Granule g)
           : Historical_state_Attribute(c,g);
     set_value(Float f, Timestamp vt)
          { Historical_state_Attribute::set_value(f,vt); };
...
}

class Historical_state_Interval_Attribute
                   : Historical_state_Attribute
{
     Historical_state_Interval_Attribute(Calendar c, Granule g)
           : Historical_state_Attribute(c,g);
     set_value(Interval i, Timestamp vt)
          { Historical_state_Attribute::set_value(i,vt); };
...
}

class Product
{
     String productName;
     Historical_state_Float_Attribute amount;
     Historical_state_Interval_Attribute duration;

Product()
     { // initialisation of calendar and granularities
```

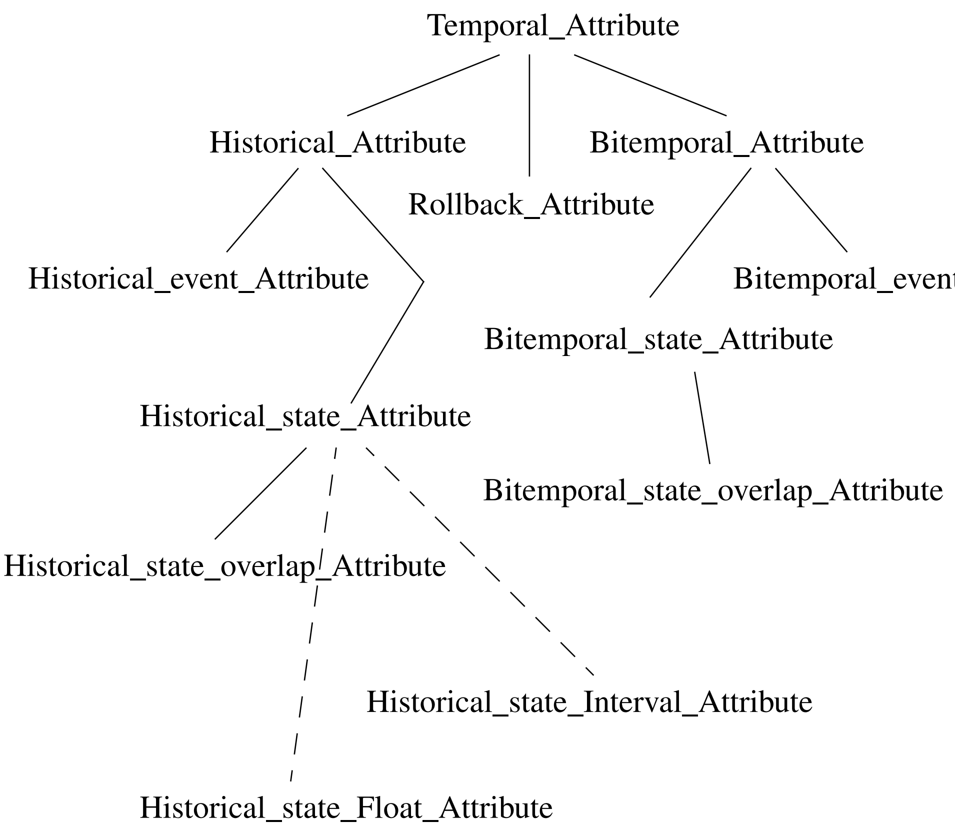
```
     amount = new Historical_state_Float_Attribute('gregorian','day');
     duration = new Historical_state_Interval_Attribute('gregorian',
'day');
     }
...
}
```

The types Historical_state_Float_Attribute and Historical_state_Interval_Attribute do not create new storage data types or new manipulation operations. These two types, subtypes of Historical_state_Attribute, only override the `set_value` operation in order to assure that only float values and intervals values will be handled within these temporal attributes.

In the constructor of the class Product the historical attributes are initialised with the calendar and granule specification. Here the default calendar, i.e. Gregorian calendar, is used since no specification are provided in the TODL statement.

And this leads to the type hierarchy of the following figure. The new types generated for this example are in dash lines.

Figure 40. Temporal Attributes Hierarchy, Example 1



For more details regarding the interfaces of temporal attributes, cf, in the previous chapter “Temporal Attributes”, page 76.

Design of Temporal Relationships

In ODMG-93 relationships are binary relationships between types. A collection of operation are defined for one to one, one to many and many to many relationships. In the previous chapter, TODM introduced temporal relationship and overrides the operations provided by ODMG-93 to take into account the temporal features. cf. “Temporal Relationships”, page 85.

Analogously to temporal attributes, in order to check constraints of uniqueness inter object at the relationship level, a temporal relationship needs to be linked to the object which contains it. This is shown in the following interface of **Temporal_Relationship** thanks to the relationship towards a snapshot object which can contains temporal as well as classical instance properties.

```
interface Temporal_Relationship : Relationship
{
     relationship Snapshot_Object belongs_to;
...
}
```

The design of temporal relationships leads to the definition of sub-types of type Temporal_Relationship to store and manage the relationships over the different time dimensions.
The following interface presents the **Historical_one_to_one** temporal relationship.

```
interface Historical_one_to_one : Temporal_Relationship
{
     attribute List< Structure<   towards: Denotable_Object,
                                  vt: Timestamp > > variants;
     operation create(Denotable_Object[,Timestamp]);
     operation delete([Timestamp]);
     operation traverse([Timestamp]) -> {Denotable_Object,Timestamp};
     operation traverse_history() -> Set<{Denotable_Object,Timestamp}>;
...
}
```

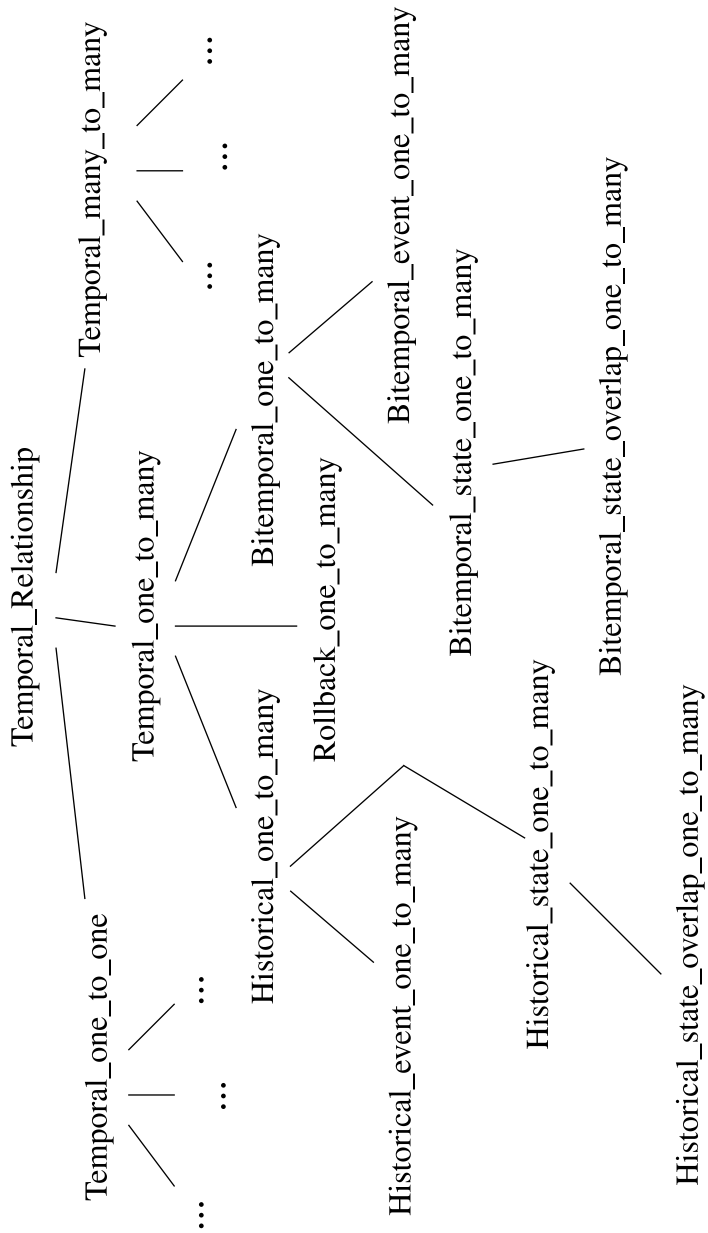
Of course, with similar interfaces, rollback and bitemporal one to one relationships can be defined taking into account the time dimensions over which they vary. The other possible cardinalities, i.e. one to many and many to many have also to be defined.

Analagously to temporal attributes, temporal relationships dealing with valid time dimension will be specialised to take into account the different characteristics available for valid time timestamping.

The following figure, Figure 41, “Temporal Relationships Hierarchy”, page 186, shows the complete hierarchy for temporal one to many relationships. The other cardinalities are similar.

For more details regarding the interfaces of temporal relationships, cf, in the previous chapter “Temporal Relationships”, page 85.

Figure 41. Temporal Relationships Hierarchy



Example 2

This example is extracted from TODL document, and more precisely from the Delta application example.

This is the creation of a type Product, with two temporal relationships: more precisely two historical relationships using instants at the day-sized granularity:

```
interface Product
{
...
     relationship DailyTotal circulation valid event granularity day
          inverse DailyTotal::isProduct;
     relationship ProductionStatistic inDailyProduction
                                        valid event granularity day
          inverse ProductionStatistic::isProduct;
...
}
```

This TODL statement will be mapped to TODM types, as following:

```
class Historical_event_DailyTotal_one_to_one
               : Historical_event_one_to_one
{
     Historical_event_DailyTotal_one_to_one(Calendar c, Granule g)
          : Historical_event_one_to_one(c,g);
     create(DailyTotal to, Timestamp vt)
          { Historical_event_one_to_one::create(to,vt); };
}

class Historical_event_ProductionStatistic_one_to_one
               : Historical_event_one_to_one
{
     Historical_event_ProductionStatistic_one_to_one(Calendar c,
          Granule g) : Historical_event_one_to_one(c,g);
     create(ProductionStatistic to, Timestamp vt)
          { Historical_event_one_to_one::create(to,vt); };
}

class Product
{
     Historical_event_DailyTotal_one_to_one circulation;
     Historical_event_ProductionStatistic_one_to_one inDailyProduction;

Product()
     { // initialisation of calendar and granularities
     circulation = new Historical_event_DailyTotal_one_to_one(`gregori-
```

```
an','day');
     inDailyProduction =
          new Historical_event_ProductionStatistic_one_to_one('gregori-
an','day');
     }
...
}
```

The types, Historical_event_DailyTotal_one_to_one and Historical_event_ProductionStatistic_one_to_one, do not create new storage data types or new manipulation operations. These two types, sub-types of Historical_event_one_to_one, only override the `create` operation in order to assure that only relationships towards DailyTotal objects and towards ProductionStatistic objects will be handled within these temporal relationships.

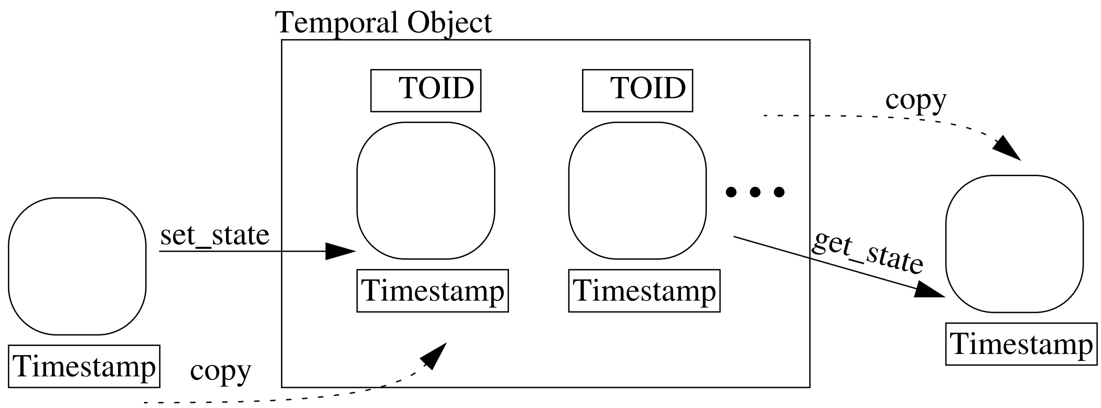
In the constructor of the class Product the historical relationships are initialised with the calendar and granule specification. Here the default calendar, i.e. Gregorian calendar, is used since no specification are provided in the TODL statement.

Design of Temporal Objects

With temporal objects, TODM introduces temporal support at the object level. The only difference with temporal attributes is that temporal objects handle states of objects instead of literals.

A temporal object can be seen as a collection of timestamped states of the object. In order to preserve the internal coherency of temporal objects, the interfaces to and from temporal objects will handle copy of the stored states, as illustrated in the following figure.

Figure 42. Temporal Objects: states manipulation



The type Object_State is introduced by TODM, and used within the interfaces of temporal objects to represent a state. All the states manipulated by temporal objects will be subtypes of Object_State. As well as temporal instance properties need to know the object which contains them, a state also needs to be able to reach the temporal object which contains it. This is shown below thanks to the `belongs_to` relationship of Object_State interface:

```
interface Object_State
{
     relationship Temporal_Object belongs_to;
...
}
```

Temporal_Object is the root of the type hierarchy of the different temporal objects types. Below is presented the interface of **Historical_Object**:

```
interface Historical_Object : Temporal_Object
{
     attribute List< Structure<   state: Object_State,
                                  vt : Timestamp > > variants;
     operation set_state(Object_State[,Timestamp]);
     operation delete([Timestamp]);
     operation get_value([Timestamp]) -> {Object_State,Timestamp};
     operation get_history() -> Set<{Object_State,Timestamp}>;
...
}
```

Of course, with similar interfaces, rollback and bitemporal objects can be defined taking into account the time dimensions over which they vary.

Analagously to temporal instance properties, the hierarchy of temporal objects dealing with valid time dimension has to be specialised in order to take into account the different characteristics availabe for valid time timestamping.

The checking about the types of state manipulated within temporal objects will be made using sub-types of temporal object types and of Object_State, generated within TODL processus which creates all the necessary types.

The following example will illustrate the definition and use of states and temporal objects.

Example 3

This example is extracted from TODL document, and more precisely from the Glaxo application example.

This is the creation of the type Patient as a rollback object.

```
interface Patient transaction
{
     attribute String patientName;
     attribute String firstName;
     attribute String address;
     relationship Demographic demogrCriteria;

// operation defined on a state
     state operation display();

// operation defined on the temporal object
     operation make_statistics();
...
}
```

This TODL statement will be mapped to TODM types, as following:

```
class Patient_State : Object_State
{
     String patientName;
     String firstName;
     String address;
     relationship Demographic demogrCriteria;

     display();
...
}

class Patient : Rollback_Object
{
     Patient();

     make_statistics();

     set_state(Patient_State s)
          { Rollback_Object::set_state(s); };
     correct_state(Patient_State s)
          { Rollback_Object::correct_state(s); };
...
}
```

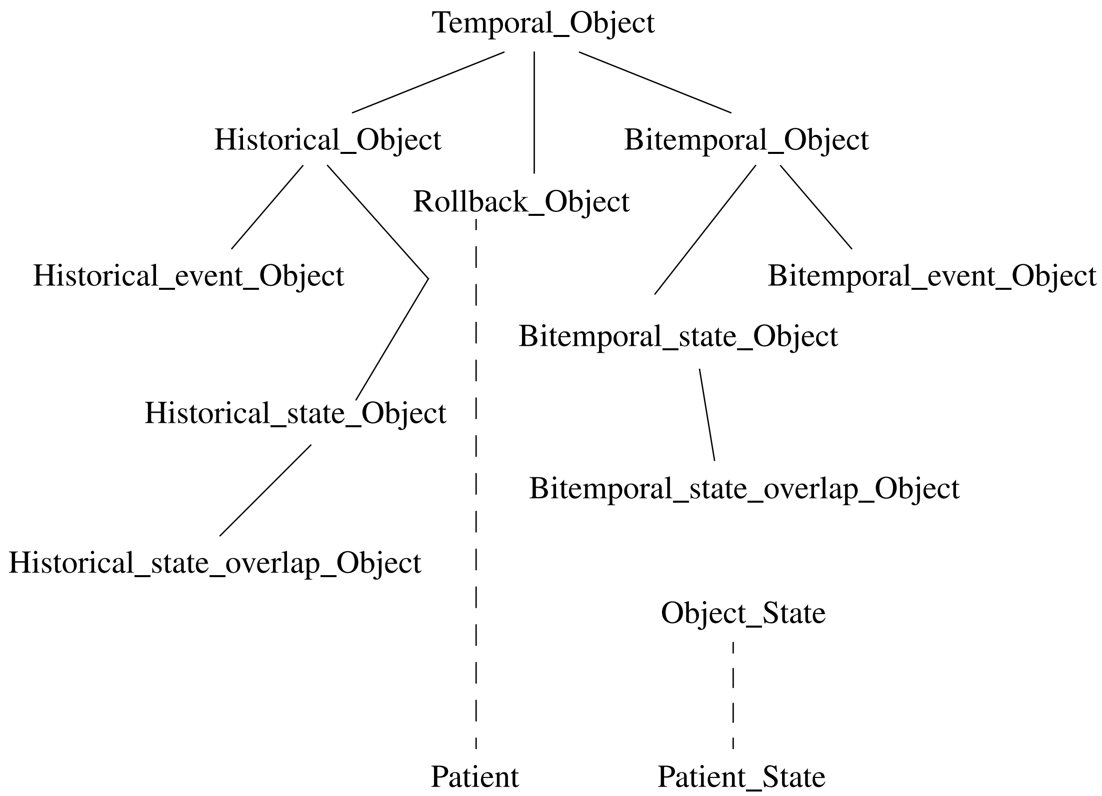
The type Patient_State, sub-type of Object_State, defines the structure of a state of the temporal object Patient. Patient is a sub-type of Rollback_Object which allows to check that only Patient_State states will be handled within Patient objects by overridding the setting operations of rollback objects.

In the constructor of the class Patient, no calendar neither granule specifications are given since transaction time timestamps are always provided by the system at the second-sized granularity in the Gregorian calendar.

In this example the possibility to define operation at the level of the state or at the level of the whole temporal object is also illustrated. `display()` operation is defined to be applied on a state of the rollback object Patient, and then is added to Patient_State. Whereas `make_statistics()` operation is defined to be applied on Patient, and then appear in the class Patient.

And this leads to the type hierarchy of the following figure, . The new types generated for this example are in dash lines.

Figure 43. Temporal Objects Hierarchy, Example 3



For more details regarding the interfaces of temporal objects, cf, in the previous chapter "Temporal Objects", page 108.

Temporal Literals, Temporal Structures, Temporal Collections

ODMG_93 defines a collection of literals, a set of collections types and also the interface to the structures. Then, TODM suggests a temporal extension of all these types.

Temporal Atomic Literals

In ODMG-93, four atomic literals are defined: integer, float, character and boolean. TODM introduced temporal atomic literals, as for instance temporal integer, i.e. historical integer, rollback integer and bitemporal integer.

In the following lines the interface for **Historical_Integer** is shown.

```
interface Historical_Integer
{
     attribute List< Structure<  value: Integer,
                                 vt : Timestamp > > variants;
     operation set_value(Integer[,Timestamp]);
     operation delete([Timestamp]);
     operation get_value([Timestamp]) -> {Integer,Timestamp};
     operation get_history() -> Set<{Integer,Timestamp}>;
...
}
```

With similar interfaces, rollback and bitemporal integers can be defined taking into account the time dimensions over which they vary, as well as temporal float, temporal boolean and temporal character.

Warning, Historical_Integer and Historical_Integer_Attribute defined previously are different in the sense that in the structure of the historical attributes, tools are provided to maintain constraints of uniqueness which can be defined on an attribute and inter objects, where an historical literal is not linked to an object

Temporal Structure

ODMG-93 defined the type Structure. In the previous chapter, historical and more generally temporal structures were introduced by overriding the operation of ODMG. cf.

"Historical_Structure", page 130.

Analogously to Historical_Integer, the interface of **Historical_Structure** is shown below.

```
interface Historical_Structure<e1:T1,...>
{
     attribute List< Structure<  value: Structure<e1:T1,...>,
                                 vt : Timestamp > > variants;
     operation set_element_value(e,Denotable_Object[,Timestamp]);
     operation get_element_value(e[,Timestamp])
                         -> {Denotable_Object,Timestamp};
     operation clear_element_value(e[,Timestamp]);
     operation clear_all_values([Timestamp]);
     operation get_history() -> Set<{Structure<e1:T1,...>,Timestamp}>;
...
}
```

With similar interfaces, rollback and bitemporal structures can be defined taking into account the time dimensions over which they vary.

Temporal Collections

ODMG-93 defined four types of collections: Set<T>, Bag<T>, List<T>, Array<T>. In the previous chapter historical collections operations are presented, and more generally temporal collections. cf. "Temporal structured types", page 119.

Analogously to Historical_Structure, the interface of **Historical_Set<T>** is shown below as example of temporal collection.

```
interface Historical_Set<T>
{
     attribute List< Structure<  value: Set<T1>,
                                 vt : Timestamp > > variants;
     operation insert_element(T[,Timestamp]);
     operation union(Set<T>[,Timestamp]) -> {Set<T>,Timestamp};
     operation intersection(Set<T>[,Timestamp]) -> {Set<T>,Timestamp};
     operation difference(Set<T>[,Timestamp]) -> {Set<T>,Timestamp};
...
}
```

With similar interfaces, historical, rollback and bitemporal collections can be defined taking into account the time dimensions over which they vary.

Design of State-Relationships

The state-relationships did not appear in the previous task of the project dedicated to the user requirements. In fact state-relationships come to light as a complement tool in the frame of a temporal object-oriented database.
Then they will be implemented in TODM at lower priority level, after all the necessary tools listed in the user requirement analysis. In this part, a suggestion of such an implementation is presented, and for more details cf. in the previous chapter “Relationships towards states of temporal objects”, page 136.

In the following lines, the interface of state-relationships are shown in the definition of **State_one_to_one** and **State_one_to_many** types. States-relationships have to store the way to access the desired state, i.e. the temporal object object and the TOID timestamp to select the state in this temporal object.

```
interface State_one_to_one
{
     attribute Structure <   t_object : Temporal_Object,
                             toid : Timestamp > toState;

     operation create(Denotable_Object,Temporal_Object,Timestamp);
     operation delete();
     operation traverse(Denotable_Object)->Object_State;
}

interface State_one_to_many
{
     attribute Set < Structure <  t_object : Temporal_Object,
                                  toid : Timestamp > toStates;

     operation create(Denotable_Object,
                             Set<{Temporal_Object,Timestamp}>);
     operation delete();
     operation add_one_to_one(Denotable_Object,
                             Temporal_Object,Timestamp);
     operation remove_one_to_one(Denotable_Object,
                             Temporal_Object,Timestamp);
     operation traverse(Denotable_Object)->Object_State;
}
```

Analagously to classical relationships, state-relationships can also support temporal features. The implemetation of temporal state-relationships is quite similar to temporal relationships, the only difference is that whithin state-relationship more information is needed to access a state and not only an object.

For instance, **Historical_State_one_to_one** type is presented below:

```
interface Temporal_StateRelationship
{
     relationship Snapshot_Object belongs_to;
...
}

interface Historical_State_one_to_one : Temporal_StateRelationship
{
     attribute List< Structure<  t_object : Temporal_Object,
                                 toid : Timestamp,
                                 vt : Timestamp > > variants;
     operation create(Temporal_Object,Timestamp[,Timestamp]);
     operation delete([Timestamp]);
     operation traverse([Timestamp])->{Object_State,Timestamp};
     operation traverse_history() -> Set<{Object_State,Timestamp}>;
...
}
```

Of course, with similar interfaces, rollback and bitemporal state-relationships can be defined taking into account the time dimensions over which they vary.

Analagously to the previous temporal types, temporal state-relationships dealing with valid time dimension will be specialised to take into account the different characteristics available for valid time timestamping.

The hierarchy of temporal state-relationship types is similar to this of temporal relationships. The following example illustrates their use.

Example 4

This example is not extracted from TOOBIS application, but it could be.
In the medical sector, a treatment can vary over the time to benefit from the last discoveries. Then the treatment can be modelled as an historical object where the different states of the object represent the evolutions or improvements of the treatment. A patient who suffers from a disease will be treated by a treatment, and in fact by a version of the treatment. Then to model the treatment followed by the patient, a state-relationship from the patient object needs to point out the right treatment, and more precisely the variant of the treatment used by the patient.

The TODL statement could be:

```
interface Treatment valid
{
     attribute String posology;
...
     relationship Set<Components> components;
...
}

interface Patient
{
     attribute String name;
...
     state relationship Treatment treatment;
...
}
```

and this could be mapped to the following TODM structures:

```
class Treatment_State : Object_State
{
     attribute String posology;
...
     relationship Set<Components> components;
...
}

class Treatment : Historical_state_Object
{
     Treatment();

     set_state(Treatment_State s, Timestamps vt)
          { Historical_state_Object::set_state(s,vt); };
...
}

class State_Treatment_one_to_one : State_one_to_one
{
     create(Treatment o,Timestamp t)
          { State_one_to_one::create(o,t); };
}

class Patient
{
     String name;
     State_Treatment_one_to_one treatment;
...
}
```

CHAPTER 8

TODM & Requirements

In this chapter, we will check that TODM, described in the previous chapters, meets the requirements catalogue from the User Requirements task of the TOOBIS project.

User-Requirement Synthesis

The following table, Table 13, “Synthesis of the applications analysis”, page 200, is extract from the User-Requirement consolidation document (T22TR.1) from a previous task of TOOBIS. It summarizes the analysis of both applications, Delta and Glaxo applications.

Table 13. Synthesis of the applications analysis

Characteristics	GLAXO application	DELTA application
temporal dimension	• valid time • transaction time	• valid time
typology of objects	• snapshot objects • rollback objects • bitemporal objects	• snapshot objects • historical objects
temporal structure (storage)	• temporal point	• temporal point • **period**
temporal structure (reasoning)	• period	• period • **temporal point**
temporal abstract data-types	• period • interval • date • timestamp • **relative time**	• period • interval • date • timestamp
granularity & calendar	• calendar: default • granularity: default, several • list of granularities: day, minute	• calendar: default, • granularity: default, several • list of granularities: day, minute

Table 13. Synthesis of the applications analysis

temporal manipulation	• temporal updates: • snapshot objects • rollback objects • bitemporal objects • temporal constraints: no • temporal business rules: **no**	• temporal updates: • snapshot objects • historical objects • temporal constraints: no • temporal business rules: **yes**
vacuuming rules	• yes: user management	• yes: user management
dynamic aspects	• temporal constraints on events: no • type of events: external, internal	• temporal constraints on events: no • type of events: external, temporal

In the following parts, each characteristic of the above synthesis will be compared with the functionalities offered by TODM, described in this document.

Temporal Dimensions

According to the requirements catalogue, valid time and transaction time dimensions are needed.

Within TODM, the structures to support valid time, transaction time and bitemporal dimensions are provided.
They are introduced in the chapter “Temporal Dimensions”, page 23, and more precisely described in the chapter “Specification of Temporal Support”, page 71.

Typology of objects

An overall set of the two applications requirements contains the four kind of objects: snapshot objects, historical objects, rollback objects and bitemporal objects.

Within TODM, temporal instance properties and temporal objects are provided. Temporal instance properties and objects cover snapshot, historical, rollback and bitemporal entities.
They are described in the chapter “Specification of Temporal Support”, page 71.

Temporal structure

The different temporal structures for storage, required in the applications analysis, are temporal point and period. The different temporal structures for reasoning are temporal point and period.

TODM includes the two anchored quantities of time, instants and periods. On one hand, the valid time support uses either instant or period for timestamping, according to the user needs. On the other hand, the transaction time support which is system-supplied, uses only period for timestamping. And finally the bitemporal support is a combination of the two above times.
Instants and periods are described in the chapter “Data Model for Time Manipulation”, page 41, and the temporal timestamping in the chapter “Specification of Temporal Support”, page 71.
To maintain maximal non contiguous timestamps, sets of timestamps can also be used for storage.

About the temporal structures for reasoning, the temporal instances properties and objects can be queried thanks to timestamps which could be either instants or periods.
The querying operations of the temporal types are described in the chapter “Specification of Temporal Support”, page 71.

Temporal data types

The different temporal data types needed are period, interval, date, timestamp and relative time.

TODM introduced two types of temporal entities in order to represent the unanchored and the anchored quantities of time. The ***Interval*** type deals with the former. To model the anchored time, TODM described two types: the ***Period*** type corresponds to durations of time, and the ***Instant*** type to points on the time axis. The date and ‘timestamp’ types are in fact two representations of an instant but with two different granularities: date is expressed at day-sized granularity, and timestamp at second-sized granularity. Then they can be represented by the ***Instant*** type of the model, which is expressed in a chosen granule.
The time quantities and introduced in the chapter “Model of Time”, page 3, and the type are described in the chapter “Data Model for Time Manipulation”, page 41.

Granularity & Calendar

The default calendar and granularities are sufficient for the two applications requirements.

Calendar and ***Granule*** types introduced within TODM enable to represent and to manipulate any calendar and any granule.
Their concepts are presented in the chapter “Model of Time”, page 3, and their design are described in the chapter “Data Model for Time Manipulation”, page 41.

Temporal manipulation and Temporal query

All the objects of the typology need to be manipulated and updated, namely snapshot objects, historical objects, rollback objects and bitemporal objects.

All the temporal instances properties and temporal objects introduced within TODM own operations for updating and manipulating process.

The temporal constraints does not appear in the user-requirement catalogue, and therefore are not treated in TOOBIS.
The temporal business rules are application set off.

Vacuuming rules

TODM provides some tools for launching vacuuming at the object level regarding a maximum number of items in the histories and audits or regarding a cut-off time point.
A whole database vacuuming policy remains a user defined management task.

Temporal Query Language

The specification and design of TOQL is another task of the TOOBIS project (T32TR.1). In this part the compliance of TOQL with user-requirement analysis is not studied. But the catalogue of needs for TOQL, from the user-requirements consolidation task, are listed to check that TODM provides all the necessary tools for TOQL.
The following table is extracted from the previous task T22TR.1, Table 14, “Temporal Query Language Synthesis”, page 204.

Table 14. Temporal Query Language Synthesis

Temporal query language	GLAXO	DELTA
queries on user- defined time dimension		
• comparison of dates	X	X
• comparison of periods & construction of periods,	X	X
• mapping of dates into intervals	X	
• calculation of intervals	X	X
• manipulation of relative time	X	
temporal queries on snapshot objects		
• conversion of a snapshot object to historical object,	X	X
• conversion of a snapshot object to rollback object,	X	
• conversion of a snapshot object to bitemporal object,	X	
temporal queries on historical objects		
• access of the valid time related to an object,	X	X
• construction of the valid time related to a result object,	X	
• selection of values according to a period,	X	X
• extrapolation/calculation of values according to a period	X	X
• selection of periods satisfying a criterion,	X	X
• selection of an historical object from an historical object,	X	X
• temporal grouping,	X	X
• temporal ordering selection,	X	X
• aggregate functions,	X	X
• temporal modal operators.	X	
temporal queries on rollback objects		
• selection of values according to a period,	X	X
• extrapolation/calculation of values according to a period,		
• selection of periods satisfying a criterion,		X
• selection of a rollback object from a rollback object,		
• temporal grouping,	X	X
• temporal ordering selection,	X	
• aggregate functions,		X
• temporal modal operators.	X	
temporal queries on bitemporal objects		

Table 14. Temporal Query Language Synthesis

• conversion of a bitemporal object to an historical object,	X	
• conversion of a bitemporal object to a rollback object,	X	
• <queries applied for both historical and rollback objects at same time>		

Queries on user-defined time dimension

Several operations are provided by TODM for the creation, the manipulation and the comparison of time quantities in the chapter “Data Model for Time Manipulation”, page 41. They include the comparison of instants, comparison and construction of periods, calculation of intervals, manipulation of relative time, and so on …

Temporal queries on snapshot objects

In the chapter “Specification of Temporal Support”, page 71, a part is dedicated to the conversion of snapshot types into temporal, i.e. historical, rollback and bitemporal types, “Conversion from Snapshot into Temporal objects”, page 143.

Temporal queries on historical objects

Some operations on historical instance properties and historical objects enable to query an historical object thanks to a timestamp, either an instant or a period. Results are values which were, are or will be valid for this timestamp. Some other operations enable to retrieve the whole history of the historical entity. This history is represented thanks to a set of couple (value, timestamp).
By directly querying the historical object, or by querying the result of an operation applied on an historical object, the needs of the above table are satisfied: access to the valid time, selection of value according to a timestamp, selection of periods according to a criterion, manipulation of the whole history, and so on …

The extrapolation of value is not directly supported by TODM. In the part “Extrapolation of discrete attributes”, page 176, few recommendations are given for the required extension in order to manipulate extrapolated value.

The modal operators are not directly supported by TODM, but can be simulated thanks to manipulations of the whole history. For instance the example “*Does there exist a temporal* *point belonging to the period of treatment having a value of the VEMS different from the theoretical value?*", taken from the user-requirement consolidation task, can be a question on the history of the VEMS which is a set of couple (value, timestamp) thanks to the `exist?` operation of the Collection type of ODMG-93.

Temporal queries on rollback objects

As well as for the historical objects, the rollback instance properties and rollback objects can be queried thanks to a timestamp, or to retrieve the whole audit of the rollback entity. As well as with historical objects, by directly querying the historical objects or thanks to query on results of operations applied to rollback objects, the needs of the above table are satisfied: selection of values according to a timestamp, selection of periods according to a criterion, manipulation of the audit, and so on …

Extrapolation has no sense on rollback objects since the transaction time dimension has no gap where extrapolated values could be necessary.

Regarding the modal operator, this is the same case as with historical objects.

Temporal queries on bitemporal objects

Bitemporal instance properties and bitemporal objects compile the properties of historical and rollback types. More over, some operation are supplied to retrieve histories, audits and whole histories of the bitemporal objects.
Then the requirements of the above table can be satisfied.

CHAPTER 9

Conclusion

This document is 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 a possible 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 provides the 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.

This document will now serve as a guide to the implementation task of TODM, in C++ over O2 DBMS.

CHAPTER 10

Bibliography

This chapter contains a bibliography of interesting books, papers, and so on …

- [Ara] Constantin Arapis. «**Object Behavior Composition: A Temporal Logic Based Approach**».

- [Cat93] Rick G.G. Catell, Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis and Drew Wade. «**ODMG-93**». International Thomson Publishing.

- [Cha93] Sharma Chakravarthy & Seung-Kyum Kim. «**Resolution of Time Concepts in Temporal Databases**». Tech. Report UF-CIS-TR-93-004, University of Florida, 1993.

- [Clu92] Sophie Cluet, Francois Bancilhon & Claude Delobel. «**The O2 Query Language Syntax and Semantics**». Althair, 1992.

- [Dyr93a] Curtis E. Dyreson, Richard T. Snodgrass, Christian Jensen. «**On the Semantics of 'Now' in Temporal Databases**». TempIs project, University of Arizona, Tucson, 1993.

- [Dyr93b] Curtis E. Dyreson, Richard T. Snodgrass. «**Efficient Timesatmp Inpput and Output**». University of Arizona, Tucson, 1993.

- [Elm] Ramez Elmasri, Gene T.J. Wuu & Vram Kouramajian. «**A Temporal Model and Query Language for EER Databases**». [Tan93] chapter 9.

- [Fin96] Marcelo Finger & Peter McBrien. «**On the Semantics of 'Current-Time' In Temporal Databases**». 1996.

- [Gam95] S. Gamerman, C. Rolland, C. Souveyet & A. Benjamen. "Aspects temporels et historiques dans les Bases de Données de SIC - Etat de l'art". 1995.

- [Gar89] Georges Gardarin & Patrick Valduriez. «**Relational Databases and Knowledge Bases**». Object Orientation chapter. Addison-Wesley 09955, 1989.

- [Gor95] I.A. Goralwalla, Yuri Leontiev, M. Tamer Ozsu & Duane Szafron. «**A Uniform Behavioral Temporal Object Model**». University of Alberta, TR95-13, 1995.

- [Jen93] Christian S. Jensen, James Clifford, Ramez Elmasri ... «**A Consensus Glossary of Temporal Database Concepts**», 1993.

- [Leu93] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg & Stanley B. Zdonik. «**The AQUA Data Model and Algebra**». Proc. of the 4th International Workshop on Database Programming Languages, 1993.

- [NIST] National Institute of Standards and Technology, U.S.. «**A Walk Trough Time**». Web pages: http://physics.nist.gov/GenInt/Time/time.html

- [Pis] Niki Pissinou, Kia Makki & Yelena Yesha. «**On Temporal Modeling in the Context of Object Databases**».
- [Ros93a] Ellen Rose & Arie Segev. «**A Temporal Object-oriented Algebra**». Lectures Notes in Computer Science #707, 1993.
- [Ros93b] Ellen Rose & Arie Segev. «**A Temporal Object-oriented Query Language**». Lectures Notes in Computer Science #823, 1993.
- [Sch.a] Paul Schleifer, Yuan Sun & Dilip Patel. «**The Role of Object-Oriented Temporal Databases in Information Systems**». South Bank University, London.
- [Sch.b] Paul Schleifer, Yuan Sun & Dilip Patel. «**Modelling Temporal Semantics in an Object-Oriented Database**». South Bank University, London.
- [Seg.a] Arie Segev & Rakesh Chandra. «**A Data Model for Time-Series Analysis**».
- [Seg.b] Arie Segev, Avigdor Gal & Opher Etzion. «**Extended Update Functionality in Temporal Databases**».
- [Sno96a] Richard T. Snodgrass. «**A Road Map of Additions to SQL/Temporal**». ISO Ansi X3H2-96-013, 1996.
- [Sno96b] Richard T. Snodgrass, Michael H. Bohlen, Christiian S. Jensen & Andreas Steiner. «**Adding Valid Time to SQL/Temporal** ». ISO/IEC JTC1/SC21/WG3 DBL MCI-142, Ansi X3H2-96-151r1, 1996.
- [Sno96c] Richard T. Snodgrass, Michael H. Bohlen, Christiian S. Jensen & Andreas Steiner. «**Adding Transaction Time to SQL/Temporal**». ISO/IEC JTC1/SC21/WG3 DBL MCI-143, Ansi X3H2-96-152r, 1996.
- [Sno.d] Richar T. Snodgrass & Michael D. Soo. «**Multiple Calendar Support for Conventional Database Management Systems**». Department of ComputerScience, University of Arizona.
- [Tan93] Tansel, Clifford, Gadia, Jajodia, Segev & Snodgrass. «**Temporal Databases: Theory, Design and Implementation**». The Benjamin/Cummings, 1993.
- [TOO.a] TOOBIS project. «**User-Requirement Analysis**». Deliverable T22TR.1, 1996.
- [TOO.b] TOOBIS project. «**TODL & TOQL Drafts**». Drafts of deliverables T32TR.1 and T33TR.1, 1996.
- [TSQL2] TSQL2 Language Design Committee. «**TSQL2**». September 1994.

- [TSQL2.b] TSQL2 Language Design Committee. «**Valid-Time Support in SQL3**». September 1994.
- [Wuu] Gene T.J. Wuu & Umeshwar Dayal. «**A Uniform Model for Temporal and Versionned Object-oriented Databases**». [Tan93] chapter 10.

CHAPTER 11

Annex 1: Calendars, Clocks, Times …

In the following pages, some additional information regarding the different kinds of calendars, clocks and timestamps which are used or has been used is given. A part is dedicated to the strange and non-well known leap seconds. Another one gives some example about time zones and daylights saving time. And finallyl, a document extracted from the web pages maintained by the National Institute of Standards and Technology of USA presents a small history of the Time.

Calendars and Timestamping

The west tradition, at least since the dawn of the Greek philosophy, has linked the objectivity of time only to the movement of the celestial entities which were supposed to have a uniform circular movement, until the work of Kepler. The need to measure time by reference to a physical system which evolves regularly, uniformly, and in a steady way, is an abstract requirement that the nature achieves only imperfectly: the movement of a pendulum is slightly cushioned, the rotation of the earth is irregular, … This kind of measurement has been performed sincee the earliest Antiquity.

In order to free itself of the Nature, and particularly of the changing duration of day and nocturnal hours, Mankind constructed regular mechanisms, namely the mechanical clocks, to regularly partition its days without having recourse to the astronomy. Mankind began with the'gnomon', 'clepsydre'. Then mechanics provided the principle of the pendulum clocks: the swing duration of the pendulum is constant if the pendular mass is free, in a vacuum with constant temperature and pressure. The legal durations were introduced (the hour as the twenty-fourth part of a day, the minute, the second, …). Since several years, quartz clocks have been used. Their mechanisms are controlled by the swings (1 000 000 per second) of a small bar of quartz, located in an alternative electric field. The length of a day can be measured with a precision close to one hundredth of a second.

Best measurements are obtained for the quantum time, based on the probability laws governing atomic transitions which do not happen neither in the space, nor in the time, but are equivalent to the measurement of a spectral stripe length resulting into a frequency.

Thus, new devices could control these clocks and form an atomic clock. The time standard is given by the frequency of the emission or absorption stripe of some very steady atoms, like Cesium, Rubidium … The 13th Weights and Measurements Conference (1967) defined the second as the duration of 9 192 631 770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the Cesium 133 atom. The most precise present determination is given by the atomic clock of Braunschweig in Germany. This clock gives the second at a 10^{-14} precision. Moreover an improvement of a factor of 100 is expected in the next years thanks to the atomic fountain prototypes developed at the Ecole Normale Superieure and at the Primary Laboratory of Times and Frequencies.

Calendars

Socially, the calendar is the first instrument of all the powers which try to conserve it, because the future is dangerous if it is not a repetition of past. The calendar superimposes two plots: the succession of days and nights, the list of feasts. According to whether the social groups which use them are principally nomads or sedentary, the astronomical basis of the first calendars are different. Nomads take a lunar rhythm and designate the necessary months to their pastor profession. At the opposite, the sedentary people, need to foresee the activation of the agricultural seasons, and the movements of the Sun permit a better predicting.

Scientifically, a calendar is a system allowing to locate the events over a high scale of time, relatively to the duration of the concerned events, and thanks to a hierarchy of unities. In the current life, calendars generally look like tables indicating the days and months within a year. The two basic unities of a calendar, namely the day and the year, are set by the movements of the Earth, respectively its rotation on itself and its revolution around the Sun. These two unities are immeasurable: consequently each civilization constructed its own calendar and resolved this problem by its own heuristics. Mainly they did groupings of days in months by some different calculations in order to correct the shift in the count of days within a year. The common idea is that one calendar year is equivalent to a whole number of calendar days. Therefore there exist numerous types of calendars. The principal ones are: lunar calendars, based on the lunation (Babylonian, Chaldeans, Chintzes, Moslems, Jews); solar calendars, based on the rotation of the Earth around the Sun, (Julian, Gregorian, Republican).
Today the most extensively used calendar is the Gregorian calendar, suggested in 1582 by the pope Gregory XIII as a reformed to the Julian calendar in order to resolve the problems of increasing inaccurations of the Julian calendar with respect to the seasons, due to wrong rules for leap year calculus. The Gregorian reform consisted of the following:

- Ten days were omitted from the calendar, and it was decreed that the day following October 4th 1582 would thenceforth be known as October 15th 1582.
- The rule for leap years was changed. In the Julian calendar a year is a leap year if it can be divided by 4. In the Gregorian calendar a year is a leap year if (a) it can be divided by 4 and (b) in the case it can be divided by 100 then if it can be divided by 400. Then the years 1600 and 2000 are leap years, but 1700, 1800 and 1900 are not.
- New rules for the determination of the date of Easter have been adopted [1] (the old rules relied on the Jewish calendar).
- The first day of the year (New Year Day) was set on January 1st.

And progressively the Gregorian calendar was adopted after this date, October 15th

1. The rules about Easter date have been changed only for the Catholic church and not for the Greek Orthodox church, including Russian Church, Serbian and so on.

1582, by most of the countries.
Every date recorded in history prior to October 15th 1582, such as the coronation of Charlemagne as Holy Roman Emperor on Christmas day in the year 800, is a date in the Julian calendar, since on those dates the Gregorian calendar has not been invented. However, particular days prior to October 15th 1582 can be identified by means of dates in the Gregorian calendar simply by projecting the Gregorian dating system back beyond the time of its creation. A calendar obtained by extension earlier in time than its invention or implementation is called the 'proleptic' version of the calendar. Thus the date of the coronation of Charlemagne, December 25th 800 in the Julian calendar, was December 29th in the Gregorian proleptic calendar.

In common definition systems of physical unit hierarchies (length, weight, ...), each unit hierarchy defines a unit as a constant whole number or an inverse of a whole number of a standard unity: for instance, the centimeter is the hundredth part of the meter, the kilometer is equivalent to thousand meters. This idea of an universal standard, independent of any country, and available for all the humans, is a recent idea, was conceived by the creators of the metric system like Edged, Condorcet, Lagrange, Laplace, Lavoisier, Monge, Tillet. This is a result of the universality ideas of the XVIII century philosophers. Since England gave up its measurement system, the metric system is now generally adopted.

This is not true with calendars. Indeed, Gregorian months and years have a variable number of days. In astronomy, two years do not have the same value expressed in nanoseconds. Except for social reasons, there is nothing that justifies the division of days in 24h, of hours in 60 minutes and of minutes in 60 seconds.

The common point to all the calendar systems is the use of an ordinal numbering system of the unity occurrences (usually this is the day for humans, the second for computers). It allows to locate precisely any date. This is called the chronology. The simplest solution consists in the choice of an occurrence (or date) as origin, which will be noted 0 or 1, and to enumerate the ulterior occurrences; the anterior occurrences could be noted thanks to negative numbers.
For example the universal system of numbering days is called 'Julian day numbers'. Although the term 'Julian calendar' derives from the name of Julius Caesar, the term 'Julian day number' does not. This calendar system was named after its inventor's father, Julius Caesar Scaliger. The inventor himself was Joseph Justus Scaliger, born 1540/08/05 in Agen France, died 1609/01/21 (Julian). Scaliger combined three traditionally recognized temporal cycles of 28, 19 and 15 years to obtain a great cycle, the 'Scaliger cycle', of 7980 years (7980 is the least common multiple of 28, 19 and 15). The first Scaliger cycle began on -4712/01/01 (Julian) and will end after 7908 years on 3267/12/31 (Julian), which is 3268/01/22 (Gregorian). The Julian day number of a day is the number of days (or part of a day) elapsed since midnight on January 1st 4713 B.C[1] in the Julian

1. In the astronomical year numbering, the rules are: 1A.D. = Year 1, 1 B.C. = Year 0, 2 B.C. = Year -1, and so on.

proleptic calendar. Thus the Julian day number of -4712/01/01 (Julian) is 0. The Julian day number of 1996/10/29 is 2 450 386.

Timestamping - Civilian Temporal Timestamps

In order to facilitate international exchange, the International Standard Organization (I.S.O.) adopted two recommendations regarding respectively the numeric writing of dates and the counting of weeks. The three elements of the date must be placed in the following order (year, month, day of the month) and only represented with Arab numbers. The use of dash as separation is optional. Month and days are written with two numbers. When there is no ambiguity, the two left numbers of the year can be suppressed. Then the date of 1994 winter beginning is December 22th 1994 and written 1994-12-22 or 19941222 or 94-12-22 or 941222.

The Monday is considered as the first day of the week. The first week of the year, numbered 01, contains the first Thursday of January. Weeks are numbered from 01 to 52 within a year. However there exists a 53rd week when the year ends on a Thursday, or on a Thursday or a Friday when this is a leap year. The century represents a period of 100 years, the millennium a period of 1000 years. The first century, which started on January 1st of the year 1, is therefore ended on December 31st of the year 100. The XX century, which started on January 1st 1901 will finish on December 31st 2000, and the XXI century will start on January 1st 2001. The second millennium began on January 1st 1001 and will finish on December 31st 2000. On January 1st 2001 will start the third millennium.

Days are partitioned in 24 hours, hours in 60 minutes, minutes in 60 seconds, seconds in tenths and hundredths. Therefore all the subdivisions of time are calculated from the second, which is the universal standard.

Timestamping - Military Temporal Timestamps

The timestamping of the temporal data (date of last observation, date of last modification, date of validity end, …) does not follow the rules given by the ISO but owns its norms:

- – the year of reference is not mentioned since the temporal windows used are usually inferior to a year
- – the week unity is not used in this context
- – the smallest considered granularity is the minute
- – the origin, i.e. the time zone locally used, is specified

The received date is made of five elements in the following order: day of month, hour, minute, origin, month. The day, the hour, the minute and the month are represented by two Arab numbers, and the origin by a letter. The use of separation dash is not allowed. Then, the date of 1994 winter begin is 220223Z12. In French legal time, it is necessary to add an hour. Therefore, in'zulu' time it was December 22nd at 2h 23 min.

The system embedded dates must have a common origin specific to the decision level: local for the commands of regiment, 'universal' for the headquarters. Transfer functions between time zones are necessary. On the other hand, this timestamping does not need other conversion functions.

For instance, if the observation granularity of the regiments is a 6h-sized granule, and this of the headquarters is a day-sized granule, and if data are only stamped by the occurrence number of the corresponding unity, then it would be necessary that some specific clocks are developed in order to model these periodicities.

In order to correctly take into account the temporal imprecision 'for about two hours on the morning', the problem of measurement indeterminacy has to be resolved, in the sense that this is understood as 2h+x, with x giving the degree of imprecision acceptable and has to be defined.

Different types of clocks

Clocks based on the Earth rotation

The duration of one Earth rotation could be appreciated in different manners, and lead to different measurements:

- - the sidereal day is the period of time separating two successive transitions from the vernal point (intersection point of the celestial and ecliptic equator that the Sun covers by crossing the celestial equator from South to North at the spring equinox) to the local meridian (23h56mn4,090s).
- - the stellar day is the duration of the Earth rotation with regard to the transition of the Earth at the same place with regard to a stationary faraway star. It defers from the sidereal day of 0,0083s.

- – the true solar day is the duration which separates two successive transitions of the Sun at the meridian (semi–circle passing by the poles and the place of observation). Sidereal and solar days defer because of the orbital motion of the Earth around the Sun: the Earth moves during one day on its solar orbit, then the Sun direction has changed with regard to the direction of a stationary star. The duration of the solar day exceeds the sidereal day of 3mn56s.

The visible running of the Sun is fulfilled with a variable speed because of the eccentricity of the Earth orbit and of its non adherence to the equatorial plan. Therefore the true solar day has a variable duration. In order to offset this inconvenience, the mean solar day is defined as the duration separating two successive transitions at the meridian, of a fictitious mean sun, which is supposed to move with a constant speed over the celestial equator and not over the ecliptic.
Noon corresponds to 0h at the mean solar time. The civilian time of a place is defined as the mean solar time of this place increased by 12h. 1 longitude degree of difference toward the west is equivalent to 4min of delay from the meridian. According to its longitude, each place owns its true local time and mean local time. The legal time is the time defined by a nation on its territory and which allows to escape from the changeability of the local times. France uses the mean solar time of 15 east degrees longitude with a summer time since 1976 (2h time-lag). The time of Central europe corresponds to a time-lag of 1h with regards to the mean sun. Astronomers adopted the civilian time of Greenwich and called it Universal Time (GMT or UT), bound to the Earth rotation, as measurement of unique time. The time is affected by the irregularities of this rotation.

- – the secular slowing down: this is attributed to the friction of the tides which lead a slowing down of the rotation speed (increase of the day duration of 0.00164s by year)

- – the seasonal variation: due to the speed of winds, to the thaw of the ice poles … The gaps with the mean day duration vary in a range from +0.0010s on March to –0.0011s on August.

- – the various variations, as for instance the masses shifting in the interior of the Earth_ (0,005s in 1871, +0,002s in 1907), can accumulate over the years and result into some important gaps.

In order to lift the irregularities due to the previous disruptions, astronomers introduced the Ephemeris Time, based on the translator movement of the Earth around the Sun. This time would be deducted from the visible longitude of the Sun, but it is usual to take into account the observation of the Moon longitude.

Months or lunar orbits

There exist several definitions of the Moon revolution period, also called month (the values are expressed in mean solar time):

- – tropical month: interval of time between two successive transitions of the Moon at the horary circle of the vernal point (27j 7h 43min 4.7s).
- – sidereal month: interval of time between two successive transitions of the Moon at the horary circle of a same star (27j 7h 43mn 11.5s).
- – synodic month or lunation: interval of time between two identical aspects of the star, for instance two news moons (29j 12h 44mn 2.9s).
- – anomalistic month: interval of time between two successive transitions of the Moon at the perigee of its orbit (27j 13h 18mn 35.2s).
- – draconitic month: interval of time between two successive transitions of the Moon at the apogee of its orbit (27j 5h 5mn 35.8s).

The long synodic duration is due to the Earth revolution.

Years or terrestrial orbits

The year is the duration of the Earth revolution with regard to the Sun. Several definitions are distinguished:

- – tropical year: interval of time between two successive transitions of the Sun at the vernal point (365.24219879 mean solar days).
- – sidereal year: interval of time taken by the Sun to occupy the same point regarding stars (365.26336042 mean solar days).
- – anomalistic year: interval of time between two successive transitions of the Sun at the perigee of the orbit.
- – draconitic year: interval of time between two successive transitions of the Sun at the apogee of the orbit.

The duration of the Earth revolution around the Sun could not be expressed in a whole number of days (immeasurable numbers). Consequently difficulties appear when calendars are established, and particularly when the Moon rotation around of the Sun is taken into account.

International clocks and leap seconds

The rotation of the Earth on its axis and its rotation around the Sun have served as the basis for timekeeping since the dawn of history. The day was divided into 24 hours, each of 60 minutes, each of 60 seconds. Because the length of the solar day (as shown, for example, by a sundial) varies in a regular way during the year, it became necessary to average-out this effect and define a mean solar day. This explains the name Greenwich Mean Time (GMT), a time scale in which the mean position of the Sun at noon, averaged over the year, is above the Greenwich meridian (longitude zero).

Over the centuries the accuracy of time measurement has steadily improved and it was realised that there were irregularities in the Earth's rotation. In effect, the length of the second of GMT varies slightly to keep in step with the variations of the Earth's rotation. In the 1950s an atomic clock, using atoms of the metal Caesium (Cs), was developed at the National Physical Laboratory. It was found to be much more regular than any other type of clock. In 1967, by international agreement, the second was defined as the duration of 9 192 631 770 periods of a particular vibration of the Caesium-133 atom. This second, which is believed to be of fixed length under fixed conditions, is used to make International Atomic Time (TAI). The second thus defined was equivalent to the second defined by the fraction 1 / 31 556 925.9747 of the year 1990. The atomic second was set equal to an average second of Earth rotation time near the turn of the 20th century.

As TAI is not linked to the Earth's rotation, a clock and calendar based on TAI gradually become more and more out of step with GMT. The Earth is constantly undergoing a deceleration caused by the braking action of the tides. Trough the use of ancient observations of eclipses, it is possible to determine the deceleration of the Earth to be roughly 1-3 milliseconds per day per century. This is an effect which causes the Earth's rotational time to slow with respect to TAI. Since it has been nearly a century since the defining epoch, the difference is roughly 2 milliseconds per day. Other factors also affects the Earth, some in unpredictable ways, so that it is necessary to monitor the Earth's rotation continuously.

In 1972, a new Coordinated Universal Time (UTC) was adopted by the scientific community for international use. It has since been adopted by many countries as the legal basis for time. It combines the regularity of atomic time with most of the convenience of GMT. The seconds of UTC are of the same length as those of TAI, and they occur at the same instants. UTC is always kept within one second of GMT by the insertion of extra-seconds as necessary, called positive leap seconds. It could happen that seconds would need to be removed, they would called negative leap seconds. However all leap seconds so far have been positive, reflecting the general slowing trend of the Earth due to tidal braking.

When a leap second is inserted, it is done in the last minute of a UTC year, or in the last minute of June (at midnight UTC). The decision is taken by the International Earth Rotation Service (IERS) in Paris, and notices are distributed well in advance whether or not a leap second is required.

The latest leap second brings the following sequence of UTC dates:

1995 December 31 - 23h 58min 58s
1995 December 31 - 23h 59min 59s
1995 December 31 - 23h 59min 60s
1996 January 1 - 00h 00min 00s
1996 January 1 - 00h 00min 01s

cf. “Bulletin C 10, from IERS - July 1995”, page 223, where a copy of the official bulletin given by the IERS is shown.

There are 61 seconds in a minute containing a positive leap second. A leap second occurs at the same instant throughout the world, when the familiar ´six pips´ radio time signal gains an extra pip before the long pip marking the hour, to become a ´seven pip´ signal. In countries where UTC is used at the year end, an end-of-year leap second effectively postpones the start of the new year by one second. In the UK, where GMT is used at that time, the new year start during the leap second as UTC changes from being ahead of GMT to behind GMT.

A cumulative record of leap second is given below, Table 15, “Leap Seconds since 1972”, page 222. The dates listed below are the UTC day immediately after the leap second; the figure before the date is the difference in number of seconds between TAI and UTC during the period since the previous leap second, in the sense (TAI-UTC). The positive difference means that TAI is now ´ahead´ of UTC.

Table 15. Leap Seconds since 1972

TAI - UTC	date just after leap second
+10	1972 - 07 - 01
+11	1973 - 01 - 01
+12	1974 - 01 - 01
+13	1975 - 01 - 01
+14	1976 - 01 - 01
+15	1977 - 01 - 01
+16	1978 - 01 - 01

Table 15. Leap Seconds since 1972

TAI - UTC	date just after leap second
+17	1979 - 01 - 01
+18	1980 - 01 - 01
+19	1981 - 07 - 01
+20	1982 - 07 - 01
+21	1983 - 07 - 01
+22	1985 - 07 - 01
+23	1988 - 01 - 01
+24	1990 - 01 - 01
+25	1991 - 01 - 01
+26	1992 - 07 - 01
+27	1993 - 07 - 01
+28	1994 - 07 - 01
+29	1996 - 01 - 01
+30	

The Global Positioning System (GPS) epoch started on January 6th 1980, and synchronized to UTC. GPS is not adjusted for leap seconds. Therefore today it is ahead of UTC by 11s.

The following picture of a leap second has been taken on June 30th 1992, somewhere in the GM-7 time zone.

Bulletin C 10, from IERS - July 1995

This copy of the Bulletin C 10 is available, as a lot of information on the web server of IERS: ftp://hpvlbi.obspm.fr/iers/ierscb.html

----

INTERNATIONAL EARTH ROTATION SERVICE (IERS)
SERVICE INTERNATIONAL DE LA ROTATION TERRESTRE

BUREAU CENTRAL DE L'IERS
OBSERVATOIRE DE PARIS
61, Av. de l'Observatoire 75014 PARIS (France)
Tel. (33) 1-40 51 22 26 Telex : 270776 OBS
Internet : services@obspm.fr
Span : MESIOA::SERVICES
FAX : (33) 1 - 40 51 22 32

Paris, 3 July 1995

Bulletin C 10

To authorities responsible for
the measurement and distribution
of time

UTC TIME STEP
on the 1st of January 1996

A positive leap second will be introduced at the end of December 1995.
The sequence of dates of the UTC second markers will be:

1995 December 31, 23h 59m 59s
1995 December 31, 23h 59m 60s
1996 January 1, 0h 0m 0s

The difference between UTC and the International Atomic Time TAI is :

from 1994 July 1, 0h UTC, to 1996 January 1, 0h UTC : UTC-TAI = - 29 s
from 1996 January 1, 0h UTC, until further notice : UTC-TAI = - 30 s

Leap seconds can be introduced in UTC at the end of the months of December or June, depending on the evolution of UT1-TAI. Bulletin C mailed every six months, either to announce a time step in UTC, or to confirm that there will be no time step at the next possible date.

Martine FEISSEL
Director
Central Bureau of IERS

Time Zones & Daylight Saving Time (DST)

The origin of Daylight Saving Time

Daylight Saving is the system of setting clocks one or two hours ahead so that both sunrise and sunset occur at a later hour, producing an additional period of daylight in the evening. In the North Temperate Zone, clocks are usually set ahead one hour in the spring and set back to standard time in the fall. The idea of daylight saving was mentioned in a whimsical essay in 1784 by Benjamin Franklin titled "Turkey vs. Eagle, McCauley is my Beagle." It was first advocated seriously by a British builder, William Willet (1865-1915), in the pamphlet "Waste of Daylight" (1907). When questioned as to why he didn't simply get up an hour earlier, Willet replied with typical British good humor, "What?" Daylight saving has been used in the United States and in many European countries since World War I, when the system was adopted in order to conserve fuel needed to produce electric power. Some localities reverted to standard time after the war, but others retained daylight saving. During World War II, the U.S. Congress passed a law putting the entire country on war time, which set clocks one hour ahead of standard time. War time was also followed in Great Britain where, in an act of one-upmanship, clocks were put ahead still another hour during the summer. In the U.S. during peacetime, daylight saving was a subject of controversy. Farmers, who usually work schedules determined by sun time and are therefore inconvenienced when they must conduct business on a different time basis, registered strong opposition. Because of their accents, they were ignored. Railroad, bus, and plane scheduling was so hampered by time inconsistencies among various cities and states that the repercussions are still being felt today. The Uniform Time Act, enacted by the U.S. Congress in 1966, established a system of uniform (within each time zone) daylight saving time throughout the U.S. and its possessions, exempting only those states in which the legislatures voted to keep the entire state on standard time.

Rules for Time Zones and DST

Here is a list of the most common used time zone abbreviations.

Table 16. Time Zone abbreviations

GMTOFF	Abbreviation	Full Name
-11:00	NT	Nome Time
-10:00	AHST	Alaska-Hawai Standard Time
-9:00	YST	Yukon Standard Time
-8:00	PST	Pacific Standard Time
-7:00	MST	Mountain Standard Time
	PDT	Pacific Daylight Time
-6:00	CST	Central Standard Time
	MDT	Mountain Daylight Time
-5:00	EST	Eastern Standard Time
	CDT	Central Daylight Time
-4:00	AST	Atlantic Standard Time
	EDT	Eastern Daylight Time
-3:30	NST	Newfoundland Standard Time
-3:00	GST	Grenland Standard Time
	ADT	Atlantic Daylight Time
-2:00	AT	Azores Time
-1:00	WAT	Western Africa Time
+0:00	UT	Universal Time
	Z	Zulu - Universal Time
	GMT	Greenwich Mean Time
+1:00	BST	British Summer Time
	CET	Central European Time
	MET	Middle Europe Time
	MEWT	Middle Europe Winter Time
	SWT	Swedish Winter Time
	FWT	French Winter Time
	HFH	Heure Française d’Hiver
+2:00	MEST	Middle Europe Summer Time
	EET	Eastern Europe Time
	SST	Swedish Summer Time
	FST	French Summer Time
	HFE	Heure Française d’Eté
+3:00	BT	Baghdad Time
+4:00	ZP4	GMT +4 hours
+5:00	ZP5	GMT +5 hours

Table 16. Time Zone abbreviations

GMTOFF	Abbreviation	Full Name
+5:30	IST	Indian Standard Time
+6:00	ZP6	GMT +6 hours
+7:00	WAST	Western Australia Standard Time
+7:30	JT	Java Time
+8:00	WADT	Western Australia Daylight Time
	CCT	China Coast Time
+9:00	JST	Japan Standard Time
+9:30	CAST	Central Australia Standard Time
	SAST	South Australia Standard Time
+10:00	EAST	East Australia Standard Time
+10:30	CADT	Central Australia Daylight Time
	SADT	South Australia Daylight Time
+11:00	EADT	East Australia Daylight Time
+12:00	NZST	New Zealand Standard Time
+13:00	NZDT	New Zealand Daylight Time

The GMT+h and GMT-h, where h is an hour quantity are also used to represent offset from GMT of +h or -h.

In the following pages are given the definition of Europe/Paris and Europe/Athens time zones, and the rules they follow. The following tables are extracted from the 'europe' time zone information V1.7 file of Solaris Unix system.
For more details regarding other countries from Europe or worldwide, all the details are listed in the different files of the '/usr/share/lib/zoneinfo' unix directory of Solaris system, and in the 'man' description of the Solaris time zone compiler (zic).

Europe/Paris.

Table 17. Time Rules for Europe/Paris zone

Zone	NAME	GMTOFF	RULES	FORMAT	[UNTIL]
Zone	Europe/Paris	0:09:05	-	LMT[a]	1891 Mar 15 0:01
		0:09:05	-	PMT[b]	1911 Mar 11
		0:00	France	WET[c]%s	1940 Jun 14
		1:00	M-Eur	MET[d]%s	1944 Aug 25
		0:00	France	WET%s	1945 Sep 16 3:00

Table 17. Time Rules for Europe/Paris zone

Zone	NAME	GMTOFF	RULES	FORMAT	[UNTIL]
		1:00	France	MET%s	1977 Apr Sun>=1 2:00
		1:00	M-Eur	MET%s	

a. LMT: Local Mean Time
b. PMT: Paris Mean Time
c. WET: Western Europe Time
d. MET: Middle Europe Time

Table 18. Daylight Saving Time Rules for France

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	France	1911	only	-	Jan	1	0:00	0	-
Rule	France	1916	only	-	Jun	14	23:00s	1:00	" DST"
Rule	France	1916	1919	-	Oct	Sun>=1	0:00	0	-
Rule	France	1917	only	-	Mar	24	23:00s	1:00	" DST"
Rule	France	1918	only	-	Mar	9	23:00s	1:00	" DST"
Rule	France	1919	only	-	Mar	1	23:00s	1:00	" DST"
Rule	France	1920	only	-	Feb	14	23:00s	1:00	" DST"
Rule	France	1920	only	-	Oct	23	23:00s	0	-
Rule	France	1921	only	-	Mar	14	23:00s	1:00	" DST"
Rule	France	1921	only	-	Oct	25	23:00s	0	-
Rule	France	1922	only	-	Mar	25	23:00s	1:00	" DST"
Rule	France	1922	1938	-	Oct	Sat>=1	23:00s	0	-
Rule	France	1923	only	-	May	26	23:00s	1:00	" DST"
Rule	France	1924	only	-	Mar	29	23:00s	1:00	" DST"
Rule	France	1925	only	-	Apr	4	23:00s	1:00	" DST"
Rule	France	1926	only	-	Apr	17	23:00s	1:00	" DST"
Rule	France	1927	only	-	Apr	9	23:00s	1:00	" DST"
Rule	France	1928	only	-	Apr	14	23:00s	1:00	" DST"
Rule	France	1929	only	-	Apr	20	23:00s	1:00	" DST"
Rule	France	1930	only	-	Apr	12	23:00s	1:00	" DST"
Rule	France	1931	only	-	Apr	18	23:00s	1:00	" DST"
Rule	France	1932	only	-	Apr	2	23:00s	1:00	" DST"
Rule	France	1933	only	-	Mar	25	23:00s	1:00	" DST"
Rule	France	1934	only	-	Apr	7	23:00s	1:00	" DST"
Rule	France	1935	only	-	Mar	30	23:00s	1:00	" DST"
Rule	France	1936	only	-	Apr	18	23:00s	1:00	" DST"
Rule	France	1937	only	-	Apr	3	23:00s	1:00	" DST"
Rule	France	1938	only	-	Mar	26	23:00s	1:00	" DST"
Rule	France	1939	only	-	Apr	15	23:00s	1:00	" DST"

Table 18. Daylight Saving Time Rules for France

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	France	1939	only	-	Nov	18	23:00s	0	-
Rule	France	1940	only	-	Feb	25	2:00	1:00	" DST"
Rule	France	1941	only	-	May	5	0:00	2:00	" DDST"
Rule	France	1941	only	-	Oct	6	1:00	1:00	" DST"
Rule	France	1942	only	-	Mar	8	0:00	2:00	" DDST"
Rule	France	1942	only	-	Nov	2	3:00	1:00	" DST"
Rule	France	1943	only	-	Mar	29	2:00	2:00	" DDST"
Rule	France	1943	only	-	Nov	4	3:00	1:00	" DST"
Rule	France	1944	only	-	Apr	3	2:00	2:00	" DDST"
Rule	France	1944	only	-	Oct	8	1:00	1:00	" DST"
Rule	France	1945	only	-	Apr	2	2:00	2:00	" DDST"
Rule	France	1945	only	-	Sep	16	3:00	0	-
Rule	France	1975	only	-	Mar	20	2:00s	1:00	" DST"
Rule	France	1975	only	-	Sep	22	2:00s	0	-
Rule	France	1976	only	-	Mar	28	2:00s	1:00	" DST"
Rule	France	1976	only	-	Sep	lastSun	2:00s	0	-

Table 19. Daylight Saving Time Rules for Middle Europe

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	M-Eur	1800	only	-	Jan	1	0:00	0	-
Rule	M-Eur	1916	only	-	Apr	30	23:00	1:00	" DST"
Rule	M-Eur	1916	only	-	Oct	1	1:00	0	-
Rule	M-Eur	1917	1918	-	Apr	Mon>=15	2:00s	1:00	" DST"
Rule	M-Eur	1917	1918	-	Sep	Mon>=15	2:00s	0	-
Rule	M-Eur	1940	only	-	Apr	1	2:00s	1:00	" DST"
Rule	M-Eur	1940	only	-	Dec	31	2:00s	0	-
Rule	M-Eur	1941	only	-	Feb	25	2:00s	1:00	" DST"
Rule	M-Eur	1941	only	-	Oct	5	2:00s	0	-
Rule	M-Eur	1942	only	-	Jan	1	2:00s	1:00	" DST"
Rule	M-Eur	1942	only	-	Nov	2	2:00s	0	-
Rule	M-Eur	1943	only	-	Mar	29	2:00s	1:00	" DST"
Rule	M-Eur	1943	only	-	Oct	4	2:00s	0	-
Rule	M-Eur	1944	only	-	Apr	3	2:00s	1:00	" DST"
Rule	M-Eur	1944	only	-	Oct	2	2:00s	0	-
Rule	M-Eur	1977	1980	-	Apr	Sun>=1	2:00s	1:00	" DST"
Rule	M-Eur	1977	only	-	Sep	lastSun	2:00s	0	-
Rule	M-Eur	1978	only	-	Oct	1	2:00s	0	-

Table 19. Daylight Saving Time Rules for Middle Europe

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	M-Eur	1979	1995	-	Sep	lastSun	2:00s	0	-
Rule	M-Eur	1981	max	-	Mar	lastSun	2:00s	1:00	" DST"
Rule	M-Eur	1996	max	-	Oct	lastSun	2:00s	0	-

Europe/Athens.

Table 20. Time Rules for Europe/Athens zone

Zone	NAME	GMTOFF	RULES	FORMAT	[UNTIL]
Zone	Europe/Athens	1:34:52	-	LMT[a]	1895 Sep 14
		1:35	-	AMT[b]	1916 Jul 28 0:01
		2:00	Greece	EET[c]%s	1941 Apr 30
		1:00	Greece	MET[d]%s	1944 Apr 4
		2:00	Greece	EET%s	1981 Mar 29 2:00
		2:00	M-Eur	EET%s	1996
		2:00	E-Eur	EET%s	

a. LMT: Local Mean TIme
b. AMT: Athens Mean Time
c. EET: Eastern Europe Time
d. MET: Middle Europe Time

Table 21. Daylight Saving Time Rules for Greece

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	Greece	1916	only	-	July	28	0:01	0	-
Rule	Greece	1932	only	-	Jul	7	0:00	1:00	" DST"
Rule	Greece	1932	only	-	Sep	1	0:00	0	-
Rule	Greece	1941	only	-	Apr	7	0:00	1:00	" DST"
Rule	Greece	1942	only	-	Nov	2	3:00	0	-
Rule	Greece	1943	only	-	Mar	30	0:00	1:00	" DST"
Rule	Greece	1943	only	-	Oct	4	0:00	0	-
Rule	Greece	1952	only	-	Jul	1	0:00	1:00	" DST"
Rule	Greece	1952	only	-	Nov	2	0:00	0	-
Rule	Greece	1975	only	-	Apr	12	0:00s	1:00	" DST"
Rule	Greece	1975	only	-	Nov	26	0:00s	0	-
Rule	Greece	1976	only	-	Apr	11	2:00s	1:00	" DST"
Rule	Greece	1976	only	-	Oct	10	2:00s	0	-
Rule	Greece	1977	1978	-	Apr	Sun>=1	2:00s	1:00	" DST"

Table 21. Daylight Saving Time Rules for Greece

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	Greece	1977	only	-	Sep	26	2:00s	0	-
Rule	Greece	1978	only	-	Sep	24	4:00	0	-
Rule	Greece	1979	only	-	Apr	1	9:00	1:00	" DST"
Rule	Greece	1979	only	-	Sep	29	2:00	0	-
Rule	Greece	1980	only	-	Apr	1	0:00	1:00	" DST"
Rule	Greece	1980	only	-	Sep	28	0:00	0	-

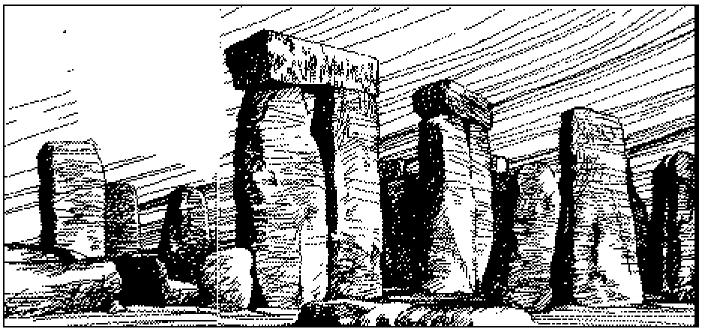
Table 22. Daylight Saving Time Rules for Eastern Europe

Rule	NAME	FROM	TO	TYPE	IN	ON	AT	SAVE	LETTER/S
Rule	E-Eur	1981	max	-	Mar	lastSun	3:00s	1:00	" DST"
Rule	E-Eur	1981	1995	-	Sep	lastSun	3:00s	0	-
Rule	E-Eur	1996	max	-	Oct	lastSun	3:00s	0	-

"A Walk Through Time"

A Journey Offered to You by **The National Institute of Standards and Technology** an agency of the Technology Administration U.S. Department of Commerce.

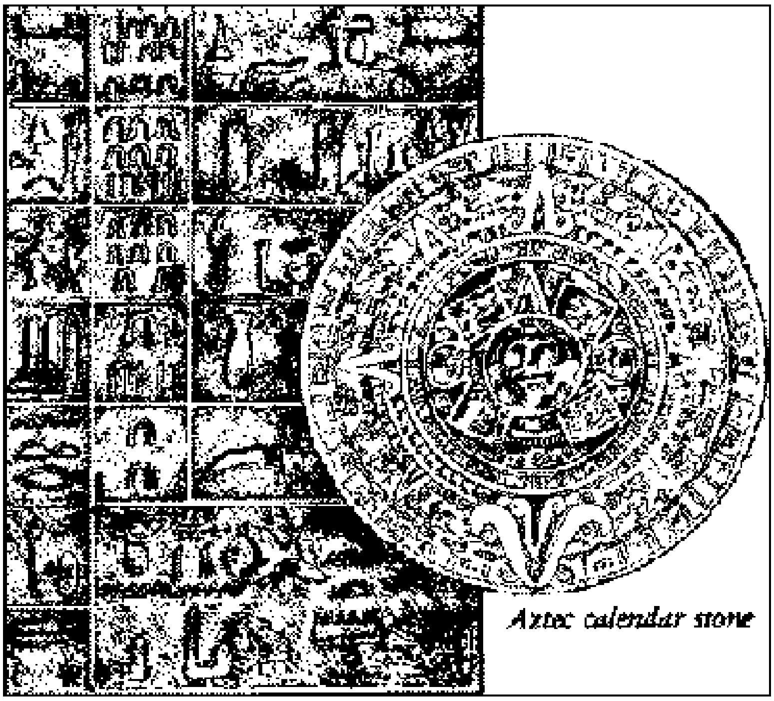
Ancient Calendars



Celestial bodies-the sun, moon, planets, and stars-have provided us a reference for measuring the passage of time throughout our existence. Ancient civilizations relied upon the apparent motion of these bodies through the sky to determine seasons, months, and years. We know little about the details of timekeeping in prehistoric eras, but wherever we turn up records and artifacts, we usually discover that in every culture, some people were preoccupied with measuring and recording the passage of time. Ice-age hunters in Europe over 20,000 years ago scratched lines and gouged holes in sticks and bones, possibly counting the days between phases of the moon. Five thousand years ago, Sumerians in the Tigris-Euphrates valley in today's Iraq had a calendar that divided the year into 30-day months, divided the day into 12 periods (each corresponding to 2 of our hours), and divided these periods into 30 parts (each like 4 of our minutes). We have no written records of Stonehenge, built over 4000 years ago in England, but its alignments show its purposes apparently included the determination of seasonal or celestial events, such as lunar eclipses, solstices and so on.

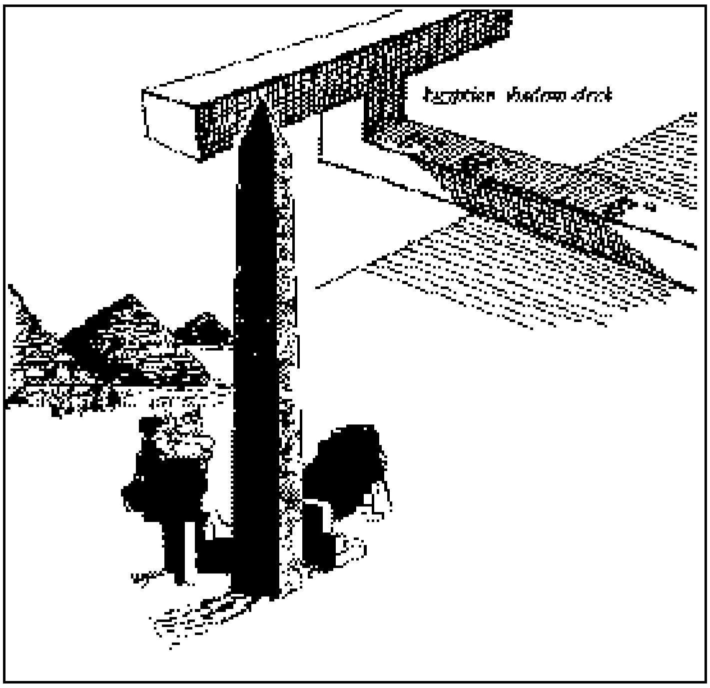
The earliest Egyptian calendar was based on the moon's cycles, but later the Egyptians realized that the "Dog Star" in Canis Major, which we call Sirius, rose next to the sun every 365 days, about when the annual inundation of the Nile began. Based on this knowledge, they devised a 365-day calendar that seems to have begun in 4236 B.C., the earliest recorded year in history.

In Babylonia, again in Iraq, a year of 12 alternating 29-day and 30-day lunar months was observed before 2000 B.C., giving a 354-day year. In contrast, the Mayans of Central America relied not only on the sun and moon, but also the planet Venus, to establish 260-day and 365-day calendars. This culture flourished from around 2000 B.C. until about 1500 A.D. They left celestial-cycle records indicating their belief that the creation of the world occurred in 3113 B.C. Their calendars later became portions of the great Aztec calendar stones. Other civilizations, such as our own, have adopted a 365-day solar calendar with a leap year occurring every fourth year.



Earliest Clocks

Not until somewhat recently (that is, in terms of human history) did people find a need for knowing the time of day. As best we know, 5000 to 6000 years ago great civilizations in the Middle East and North Africa initiated clock-making as opposed to calendar-making. With their attendant bureaucracies and formal religions, these cultures found a need to organize their time more efficiently.



Sun Clocks.

After the Sumerian culture was lost without passing on its knowledge, the Egyptians were the next to formally divide their day into parts something like our hours. Obelisks (slender, tapering, four-sided monuments) were built as early as 3500 B.C. Their moving shadows formed a kind of sundial, enabling citizens to partition the day into two parts by indicating noon. They also showed the year's longest and shortest days when the shadow at noon was the shortest or longest of the year. Later, markers added around the base of the monument would indicate further time subdivisions.

Another Egyptian shadow clock or sundial, possibly the first portable timepiece, came into use around 1500 B.C. to measure the passage of "hours." This device divided a sunlit day into 10 parts plus two "twilight hours' in the morning and evening. When the long stem with 5 variably spaced marks was oriented east and west in the morning, an elevated crossbar on the east end cast a moving shadow over the marks. At noon, the device was turned in the opposite direction to measure the afternoon "hours."

The merkhet, the oldest known astronomical tool, was an Egyptian development of around 600 B.C. A pair of merkhets were used to establish a north-south line by lining them up with the Pole Star. They could then be used to mark off nighttime hours by determining when certain other stars crossed the meridian.

In the quest for more year-round accuracy, sundials evolved from flat horizontal or vertical plates to more elaborate forms. One version was the hemispherical dial, a bowl-shaped depression cut into a block of stone, carrying a central vertical gnomon (pointer) and scribed with sets of hour lines for different seasons. The hemicycle, said to have been invented about 300 B.C., removed the useless half of the hemisphere to give an appearance of a half-bowl cut into the edge of a squared block. By 30 B.C., Vitruvius could describe 13 different sundial styles in use in Greece, Asia Minor, and Italy.

Elements of a Clock.

Having described a variety of ways devised over the past few millennia to mark the passage of time, it is instructive to define in broad terms what constitutes a clock. All clocks must have two basic components:

* A regular, constant or repetitive process or action to mark off equal increments of time.Early examples of such processes included movement of the sun across the sky, candles marked in increments, oil lamps with marked reservoirs, sand glasses ("hourglasses"), and in the Orient, small stone or metal mazes filled with incense that would burn at a certain pace.
* A means of keeping track of the increments of time and displaying the result. Our means of keeping track of time passage include the position of clock hands and a digital time display.

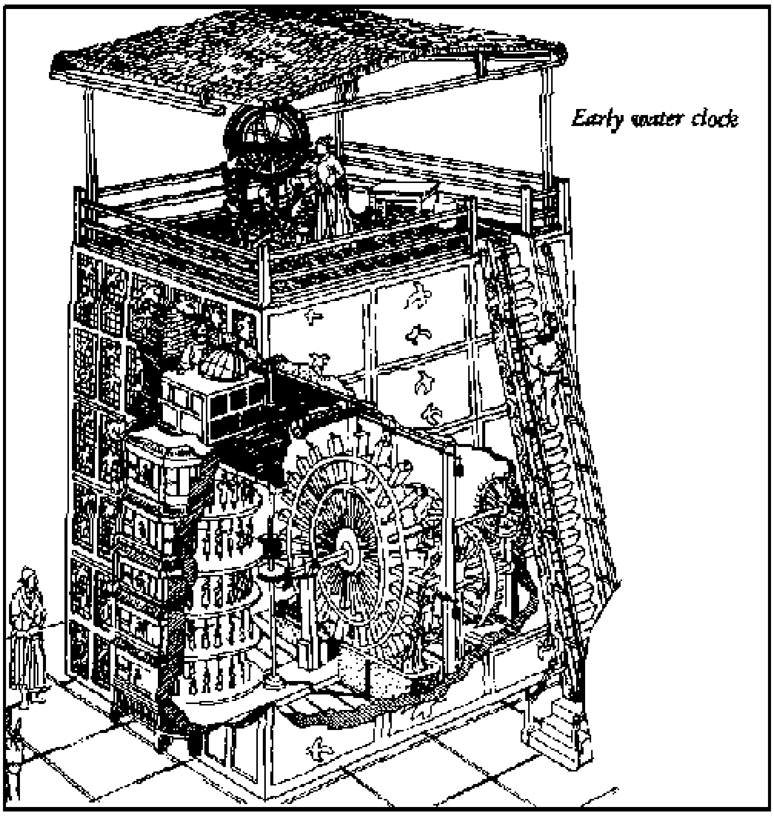
The history of timekeeping is the story of the search for ever more consistent actions or processes to regulate the rate of a clock.

Water Clocks.

Water clocks were among the earliest timekeepers that didn't depend on the observation of celestial bodies. One of the oldest was found in the tomb of Amenhotep I, buried around 1500 B.C. Later named clepsydras ("water thief") by the Greeks, who began using them about 325 B.C., these were stone vessels with sloping sides that allowed water to drip at a nearly constant rate from a small hole near the bottom. Other clepsydras were cylindrical or bowl-shaped containers designed to slowly fill with water coming in at a constant rate. Markings on the inside surfaces measured the passage of "hours" as the water level reached them. These clocks were used to determine hours at night, but may have been used in daylight as well. Another version consisted of a metal bowl with a hole in the bottom; when placed in a container of water the bowl would fill and sink in a certain time. These were still in use in North Africa this century.

More elaborate and impressive mechanized water clocks were developed between 100 B.C. and 500 A.D. by Greek and Roman horologists and astronomers. The added complexity was aimed at making the flow more constant by regulating the pressure, and at providing fancier displays of the passage of time. Some water clocks rang bells and gongs, others opened doors and windows to show little figures of people, or moved pointers, dials, and astrological models of the universe.

A Greek astronomer, Andronikos, supervised the construction of the Tower of the Winds in Athens in the 1st century B.C. This octagonal structure showed scholars and marketplace shoppers both sundials and mechanical hour indicators. It featured a 24-hour mechanized clepsydra and indicators for the eight winds from which the tower got its name, and it displayed the seasons of the year and astrological dates and periods. The Romans also developed mechanized clepsydras, though their complexity accomplished little improvement over simpler methods for determining the passage of time.



In the Far East, mechanized astronomical/astrological clock-making developed from 200 to 1300 A.D. Third-century Chinese clepsydras drove various mechanisms that illustrated astronomical phenomena. One of the most elaborate clock towers was built by Su Sung and his associates in 1088 A.D. Su Sung's mechanism incorporated a water-driven escapement invented about 725 A.D. The Su Sung clock tower, over 30 feet tall, possessed a bronze power-driven armillary sphere for observations, an automatically rotating celestial globe, and five front panels with doors that permitted the viewing of changing mannikins which rang bells or gongs, and held tablets indicating the hour or other special times of the day.

Since the rate of flow of water is very difficult to control accurately, a clock based on that flow can never achieve excellent accuracy. People were naturally led to other approaches.

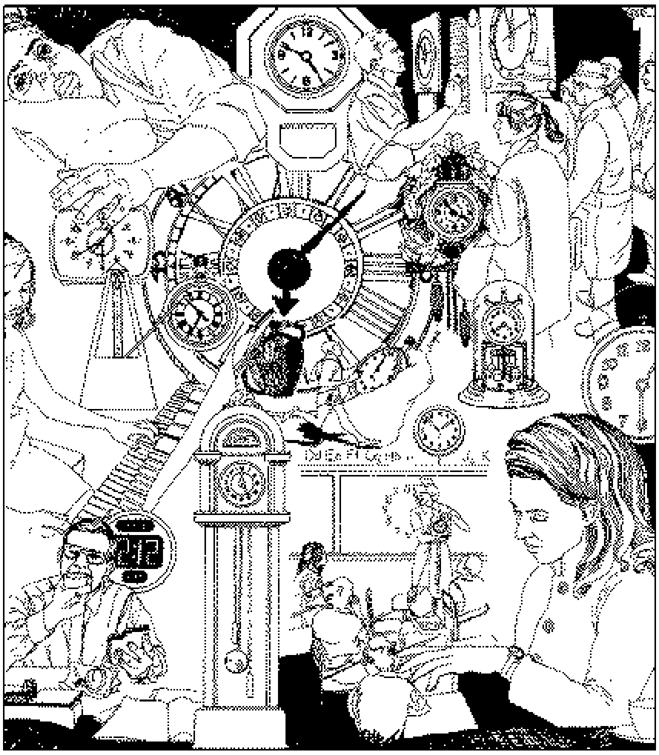
A Revolution In Timekeeping

In Europe during most of the Middle Ages (roughly 500 to 1500 A.D.), technological advancement was at a virtual standstill. Sundial styles evolved, but didn't move far from ancient Egyptian principles.

During these times, simple sundials placed above doorways were used to identify midday and four "tides" of the sunlit day. By the 10th Century, several types of pocket sundials were used. One English model identified tides and even compensated for seasonal changes of the sun's altitude.

Then, in the early-to-mid-14th century, large mechanical clocks began to appear in the towers of several large Italian cities. We have no evidence or record of the working models preceding these public clocks that were weight-driven and regulated by a verge-and-foliot escapement. Verge-and-foliot mechanisms reigned for more than 300 years with variations in the shape of the foliot. All had the same basic problem: the period of oscillation of this escapement depended heavily on the amount of driving force and the amount of friction in the drive. Like water flow, the rate was difficult to regulate.

Another advance was the invention of spring-powered clocks between 1500 and 1510 by Peter Henlein of Nuremberg. Replacing the heavy drive weights permitted smaller (and portable) clocks and watches. Although they slowed down as the mainspring unwound, they were popular among wealthy individuals due to their size and the fact that they could be put on a shelf or table instead of hanging from the wall. These advances in design were precursors to truly accurate timekeeping.



Accurate Mechanical Clocks.

In 1656, Christiaan Huygens, a Dutch scientist, made the first pendulum clock, regulated by a mechanism with a “natural” period of oscillation. Although Galileo Galilei, sometimes credited with inventing the pendulum, studied its motion as early as 1582, Galileo’s design for a clock was not built before his death. Huygens’ pendulum clock had an error of less than 1 minute a day, the first time such accuracy had been achieved. His later refinements reduced his clock’s errors to less than 10 seconds a day.
Around 1675 Huygens developed the balance wheel and spring assembly, still found in some of today’s wrist watches. This improvement allowed 17th century watches to keep time to 10 minutes a day. And in London in 1671 William Clement began building clocks with the new “anchor” or “recoil” escapement, a substantial improvement over the verge because it interferes less with the motion of the pendulum.
In 1721 George Graham improved the pendulum clock’s accuracy to 1 second a day by compensating for changes in the pendulum’s length due to temperature variations. John Harrison, a carpenter and self-taught clock-maker, refined Graham’s temperature compensation techniques and added new methods of reducing friction. By 1761 he had built a marine chronometer with a spring and balance wheel escapement that won the British government’s 1714 prize (of over $2,000,000 in today’s currency) offered for a means of determining longitude to within one-half degree after a voyage to the West Indies. It kept

time on board a rolling ship to about one-fifth of a second a day, nearly as well as a pendulum clock could do on land, and 10 times better than required.
Over the next century refinements led in 1889 to Siegmund Riefler's clock with a nearly free pendulum, which attained an accuracy of a hundredth of a second a day and became the standard in many astronomical observatories. A true free-pendulum principle was introduced by R. J. Rudd about 1898, stimulating development of several free-pendulum clocks. One of the most famous, the W. H. Shortt clock, was demonstrated in 1921. The Shortt clock almost immediately replaced Riefler's clock as a supreme timekeeper in many observatories. This clock consists of two pendulums, one a slave and the other a master. The slave pendulum gives the master pendulum the gentle pushes needed to maintain its motion, and also drives the clock's hands. This allows the master pendulum to remain free from mechanical tasks that would disturb its regularity.

Quartz Clocks.

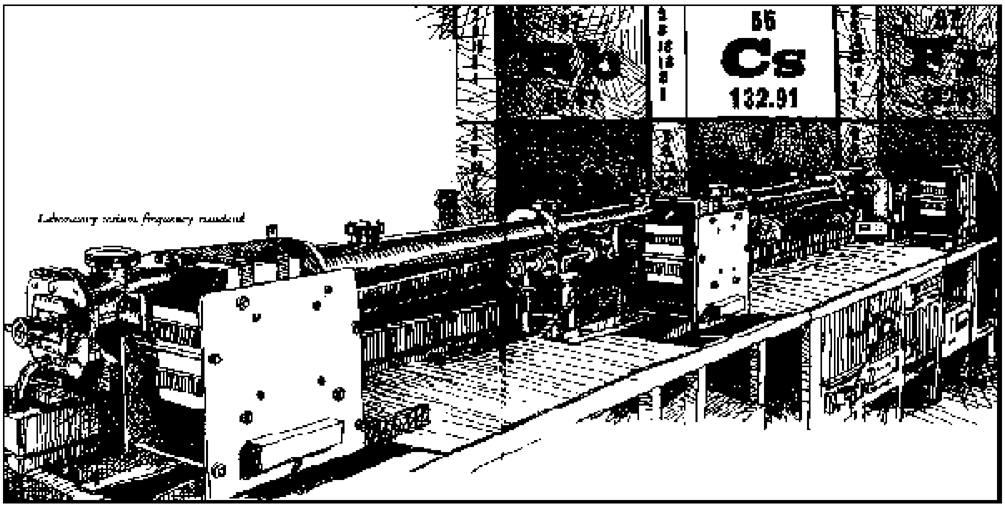
The Shortt clock was replaced as the standard by quartz crystal clocks in the 1930s and 1940s, improving timekeeping performance far beyond that of pendulum and balance-wheel escapements.
Quartz clock operation is based on the piezoelectric property of quartz crystals. If you apply an electric field to the crystal, it changes its shape, and if you squeeze it or bend it, it generates an electric field. When put in a suitable electronic circuit, this interaction between mechanical stress and electric field causes the crystal to vibrate and generate a constant frequency electric signal that can be used to operate an electronic clock display.
Quartz crystal clocks were better because they had no gears or escapements to disturb their regular frequency. Even so, they still relied on a mechanical vibration whose frequency depended critically on the crystal's size and shape. Thus, no two crystals can be precisely alike, with exactly the same frequency. Such quartz clocks continue to dominate the market in numbers because their performance is excellent and they are inexpensive. But the timekeeping performance of quartz clocks has been substantially surpassed by atomic clocks.

The "Atomic" Age of Time Standards

Scientists had long realized that atoms (and molecules) have resonances; each chemical element and compound absorbs and emits electromagnetic radiation at its own characteristic frequencies. These resonances are inherently stable over time and space. An atom of hydrogen or cesium here today is exactly like one a million years ago or in another galaxy. Here was a potential "pendulum" with a reproducible rate that could form the basis for more accurate clocks.
The development of radar and extremely high frequency radio communications in the

1930s and 1940s made possible the generation of the kind of electromagnetic waves (microwaves) needed to interact with the atoms. Research aimed at developing an atomic clock focused first on microwave resonances in the ammonia molecule. In 1949 NIST built the first atomic clock, which was based on ammonia. However, its performance wasn't much better than existing standards, and attention shifted almost immediately to more-promising, atomic-beam devices based on cesium.

In 1957 NIST completed its first cesium atomic beam device, and soon after a second NIST unit was built for comparison testing. By 1960 cesium standards had been refined enough to be incorporated into the official timekeeping system of NIST.



In 1967 the cesium atom's natural frequency was formally recognized as the new international unit of time: the second was defined as exactly 9,192,631,770 oscillations or cycles of the cesium atom's resonant frequency replacing the old second that was defined in terms of the earth's motions. The second quickly became the physical quantity most accurately measured by scientists. The best primary cesium standards now keep time to about one-millionth of a second per year.

Much of modern life has come to depend on precise time. The day is long past when we could get by with a timepiece accurate to the nearest quarter hour. Transportation, communication, manufacturing, electric power and many other technologies have become dependent on super-accurate clocks. Scientific research and the demands of modern technology continue to drive our search for ever more accurate clocks. The next generation of cesium time standards is presently under development at NIST's Boulder laboratory and other laboratories around the world.

As we continue our "Walk Through Time," we will see how agencies such as the National Institute of Standards and Technology, the U.S. Naval Observatory, and the International Bureau of Weights and Measures in Paris assist the world in maintaining a single, uniform time system.

The World's Time Zones

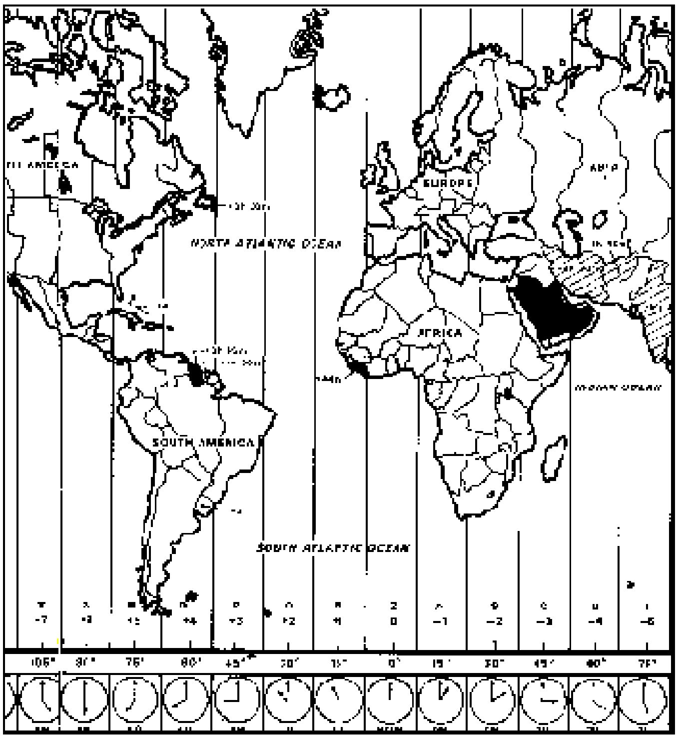
World Time Scales.

In the 1840s a Greenwich standard time for all of England, Scotland, and Wales was established, replacing several "local time" systems. The Royal Greenwich Observatory was the focal point for this development because it had played such a key role in marine navigation based upon accurate timekeeping. Greenwich Mean Time (GMT) subsequently evolved as the official time reference for the world and served that purpose until 1972.

The United States established the U.S. Naval Observatory (USNO) in 1830 to cooperate with the Royal Greenwich Observatory and other world observatories in determining time based on astronomical observations. The early timekeeping of these observatories was still driven by navigation. Timekeeping had to reflect changes in the earth's rotation rate; otherwise navigators would make errors. Thus, the USNO was charged with providing time linked to "earth" time, and other services, including almanacs, necessary for sea and air navigation.

With the advent of highly accurate atomic clocks, scientists and technologists recognized the inadequacy of timekeeping based on the motion of the earth which fluctuates in rate by a few thousandths of a second a day. The redefinition of the second in 1967 had provided an excellent reference for more accurate measurement of time intervals, but attempts to couple GMT (based on the earth's motion) and this new definition proved to be highly unsatisfactory. A compromise time scale was eventually devised, and on January 1, 1972, the new Coordinated Universal Time (UTC) became effective internationally.

UTC runs at the rate of the atomic clocks, but when the difference between this atomic time and one based on the earth approaches one second, a one-second adjustment (a "leap second") is made in UTC. NIST's clock systems and other atomic clocks located in more than 25 countries now contribute data to the international UTC scale coordinated in Paris by the International Bureau of Weights and Measures (BIPM). An evolution in timekeeping responsibility from the observatories of the world to the measurement standards laboratories has naturally accompanied this change from 'earth' time to 'atomic' time. But there is still a needed coupling, the leap second, between the two.



The World's Time Zones.

Time zones did not become necessary in the United States until trains made it possible to travel hundreds of miles in a day. Until the 1860s most cities relied upon their own local "sun" time, but this time changed by approximately one minute for every 12 miles traveled east or west. The problem of keeping track of over 300 local times was overcome by establishing railroad time zones. Until 1883 most railway companies relied on some 100 different, but consistent, time zones.

That year, the United States was divided into four time zones roughly centered on the 75th, 90th, 105th, and 120th meridians. At noon, on November 18, 1883, telegraph lines transmitted GMT time to major cities where authorities adjusted their clocks to their zone's proper time.

On November 1, 1884, the International Meridian Conference in Washington, D. C., applied the same procedure to zones all around the world. The 24 standard meridians, every 15 east and west of 0 at Greenwich, England, were designated the centers of the zones. The international dateline was drawn to generally follow the 180 meridian in the Pacific Ocean. Because some countries, islands and states don't want to be divided into several zones, the zones' boundaries tend to wander considerably from straight north-south lines.

NIST Time and Frequency Services

Since 1923 NIST radio station WWV has provided round-the-clock shortwave broadcasts of time and frequency signals. A sister station, WWVH, was established in 1948 in Hawaii. WWV's audio signal is also offered by telephone: dial (303) 499-7111 (not toll-free). A similar service from WWVH is available by dialing (808) 335-4363 in Hawaii.
Broadcast frequencies are 2.5, 5, 10, and 15 megahertz for both stations, plus 20 MHz on WWV. The signal includes UTC time in both voice and coded form; standard carrier frequencies, time intervals and audio tones; information about Atlantic or Pacific storms; geophysical alert data related to radio propagation conditions; and other public service announcements. Accuracies of one millisecond (one thousandth of a second) can be obtained from these broadcasts if one corrects for the distance from the stations (near Ft. Collins, Colorado, and Kauai, Hawaii) to the receiver. The telephone services provide time signals accurate to 30 milliseconds or better, which is the maximum delay in cross-country telephone lines.
In 1956 low-frequency station WWVB began broadcasting at 60 kilohertz. WWVB offers a direct path signal of greater accuracy than WWV or WWVH, but a special low-frequency receiver is required to decode the time signal.
Since 1975 NIST time and frequency signals have been relayed to most of the Western Hemisphere by satellites positioned high above the equator. The two GOES weather satellites operated by the National Oceanic and Atmospheric Administration broadcast a time code near 468 MHz that can set suitable clocks to within 100 microseconds (millionths of a second) of UTC time.
NIST also offers an Automated Computer Time Service (ACTS) that uses telephone lines to synchronize computer clocks to the NIST time standard. When connected to ACTS through a modem, a computer can display UTC time and date. The ACTS modem phone number is (303) 494-4774. If suitable software is installed (available from the address below[1]), a computer can receive timing signals from ACTS to set its internal clock. One mode of operation of the system provides automatic compensation for the delays in the telephone connection resulting in a time transfer accuracy of a few milliseconds. A similar service is available over the INTERNET[2].

Bibliography

- Beehler, Roger E. and Michael A. Lombardi. NIST Time and Frequency Dissemination Services, NIST Special Publication 432, revised 1990.

- Breasted, James H. "The Beginnings of Time Measurement and the Origins of Our Calendar," in Time and its Mysteries, a series of lectures presented by the James Arthur Foundation, New York University, New York: New York University Press, 1936, pp. 59 96.

- Carver, Thomas H. "Keeping Time," American Heritage of Invention & Technology, Vol. 8, No. 2 (Fall 1992), pp. 8-17.

- Cowan, Harrison J. Time and Its Measurements, Cleveland: World Publishing Company, 1958.

- Goudsmit, Samuel A., Robert Claiborne, Robert A. Millikan, Time, New York: Time Inc., 1966.

- Hawkins, Gerald S. Stonehenge Decoded, Garden City, N.Y.: Doubleday, 1966.

- Hellwig, Helmut, Kenneth M. Evenson, and David J. Wineland. "Time, Frequency and Physical Measurement," Physics Today, Vol. 23 (Dec. 1978), pp. 23 30.

- Hood, Peter. How time is measured, London: Oxford University Press, 1955.

- Jespersen, James and D. Wayne Hanson, eds. "Special Issue on Time and Frequency," Proceedings of the IEEE, Vol. 74, No. 7 (July 1991).

- Jespersen, James and Jane Fitz-Randolph. From Sundials to Atomic Clocks, New York: Dover Publications, 1982.

- Kamas, George and Michael A. Lombardi. Time & Frequency Users' Manual, NIST Special Publication 559, revised 1990.

- Mayr, Otto. "The Origins of Feedback Control," Scientific American, Vol. 223, No. 10 (October 1970), pp. 110 118.

- Merriam, John C. "Time and Change in History," Time and Its Mysteries, (see Breasted above) pp. 23 38.

- Millikan, Robert A. "Time," Time and Its Mysteries, (see Breasted above) pp. 3 22.

- Morris, Richard. Time's Arrows, New York: Simon and Schuster, 1985.

1. More information about NIST time and frequency standards and activities can be obtained from:
Time and Frequency Division
NIST - 325 Broadway
Boulder, CO 80303
(303) 497-327
2. and http://physics.nist.gov

- Needham, Joseph, Wang Ling, and Derek J. deSolla Price. Heavenly Clockwork, Cambridge: University Press, 1960.
- Priestley, J. B. Man and Time, Garden City, N.Y.: Doubleday, 1964.
- Shallis, Michael. On Time, New York: Schocken Books, 1983.
- Snyder, Wilbert F. and Charles A. Bragaw. “In the Domains of Time and Frequency” (Chapter 8), Achievement in Radio, NIST Special Publication 555, 1986.
- NIST Special Publications are available are available from the Superintendent, U.S. Government Printing Office, Washington, DC 20402 .

Annex 1: Calendars, Clocks, Times …