# External Representation of Spatial and Spatio-Temporal Values

Jose Antonio Cotelo Lema

extended by Thomas Behr and Christian Duentgen

April 2004

Last change: 21.6.2010

## 1 Introduction

For supporting the diffusion and interchange of any kind of information between systems, a well defined and unambiguous specification of an external representation for such information is required. Therefore, we need to specify an external representation for the spatial and spatio-temporal information used by the *STDB* package. The main goal in such a format must be to allow the exchange of this information between completely different platforms.

### 1.1 Notation

In the definitions of the external representation of spatial and spatio-temporal data types in the *STDB* package, the following nomenclature will be used:

**<x>** a value $x$, represented using the string representation corresponding to the type of such a value.

**<x$_1$> <x$_2$>** $\cdots$ **<x$_n$>** a list of $n$ values. If nothing explicit is said, $n \geq 0$. Note that $n = 0$ means an empty list.

## 2 Numerical Representation

For the representation of spatial and spatio-temporal data, a relevant issue is the representation of their numeric values (used for representing coordinates, instants in time or functions from the time domain to the space domain). A requirement is that the representation used for such numeric values must be able to represent their values exactly, so the process of exporting and importing data will be safe and will allow (for example) to restore the data that we have previously exported.

For ensuring the robustness and closure of spatial and spatio-temporal operations, the use of rational numbers for representing coordinates and time is required, and the precision used in such numbers could be arbitrarily large. Therefore, we need to specify how the numeric values of spatial and spatio-temporal types must be represented in such cases. For this purpose we will define the external representation of an auxiliary type *Numeric*, to which belong the coordinates in space, time instants and the coefficients of spatio-temporal functions. When representing a spatial or spatio-temporal value, it is allowed to mix the different representation formats for the *Numeric* values in it, using for each *Numeric* value the more appropiate one.

**Integer numbers:** An integer *Numeric* value $I$ can be represented in two different ways, depending on the size of the integer value:

- **Signed integers of 32 or less bits:** in this case it can be represented just as the string representation of the integer number, optionally with sign (if no sign appears, positive is assumed).

**NOTE:** such a representation of an integer number does NOT enclose the integer value in brackets.

- **Signed integers of more than 32 bits:** this category includes arbitrary precision integers. In this case the integer number is represented as an integer in base $2^{32}$, in the form:

  ( largeint <sign> <size> <m$_{size-1}$> $\cdots$ <m$_0$> )

  where:

  - <sign> can be either the symbol "+" (positive number), "-" (negative number) or nothing (interpreted as positive).
  - <size> is an integer number (its string representation) defining the number of elements $m_i$ appearing after it. <*size*> must be bigger than 0.
  - <$m_i$> is the string representation of a 32 bits unsigned integer.
    The value *I* represented is:

    $I = sign\_value * (m_{size-1} * \left(2^{32}\right)^{size-1} + \cdots + m_1 * \left(2^{32}\right)^1 + m_0)$

    where <*sign_value*> is 1 if <*sign*> is positive ("+" or nothing) and -1 if <*sign*> is negative ("-").

  Integer numbers of 32 or less bits can (optionally) be represented using the format for integers of more than 32 bits.

**Real numbers:** A real number is represented as the string representation of the real number. The string is formed in the familar way used in the most programming languages like C or Pascal. Because no cast function exists, a dot or an exponent is required to distinguish a real number from an integer.

**Rational numbers:** A rational *Numeric* value *R* is represented as:

( rat <sign> <intPart> <numDecimal> / <denomDecimal> )

where

- <*sign*> can be either the symbol "+" (positive number), "-" (negative number) or nothing (interpreted as positive).
- <*intPart*>, <*numDecimal*> and <*denomDecimal*> are non negative ($\geq 0$) integer *Numeric* values (each of them in any of the formats defined above), <*numDecimal*> $<$ <*denomDecimal*> and g.c.d.(<*numDecimal*>, <*denomDecimal*>) $= 1$ (if <*numDecimal*> $= 0$ then <*denomDecimal*> $= 1$).

The value of *R* represented is:

$$R = sign\_value * (intPart + \frac{\text{numDecimal}}{\text{denomDecimal}})$$

where <*sign_value*> is 1 if <*sign*> is positive ("+" or nothing) and -1 if <*sign*> is negative ("-").

If the value of *R* is in fact an integer value, it can (optionally) be represented as an integer *Numeric* value.

## 3 Spatial Values

### 3.1 Point

(<*xCoord*> <*yCoord*>) | undef

where <*xCoord*> and <*yCoord*> are the *X* and *Y* coordinates of the point respectively. Both are *Numeric* values. As an alternative, the undefined point is represented by the string *undef*.

### 3.2 Points

((<x$_1$> <y$_1$>) (<x$_2$> <y$_2$>) $\cdots$ (<x$_n$> <y$_n$>)) | undef

where (<$x_i$> <$y_i$>) represents a defned point value and $n \geq 0$.

### 3.3 Line

(<seg$_1$> <seg$_2$> $\cdots$ <seg$_n$>) | undef

where <*seg$_i$*> is a segment value and $n \geq 0$.

A segment value is represented as:

(<x$_1$> <y$_1$> <x$_2$> <y$_2$>)

where <$x_i$> and <$y_i$> are the coordinates of the end point *i*. All of them are *Numeric* values.

### 3.4 Region

(<face$_1$> <face$_2$> <face$_n$>) | undef

where <*face$_i$*> represents a face in the space, as a polygon with (optionally) holes, and $n \geq 0$.

A face is represented as:

(<outer_cycle> <hole_cycle$_1$> <hole_cycle$_2$> $\cdots$ <hole_cycle$_n$>)

where the *outer_cycle* and all the *hole_cycles* are cycles (simple polygons) and $n \geq 0$ (this is, it can be a face without holes).

A cycle is represented as:

(<vertex$_1$> <vertex$_2$> $\cdots$ <vertex$_n$>)

being <*vertex$_i$*> and <*vertex*$_{(i \, mod \, n)+1}$> consecutive vertices (and *hence* <*vertex*$_1$> $\neq$ <*vertex$_n$*>) and $n \geq 3$.

A vertex is a *Point* value.

## 4 Representation of Time

They are two types of time. The first one is `instant`, and the second one is `duration`. If a time value is used internal (e.g. in moving objects), the list will be:
`( type <timevalue> )`
where `type` is a Symbol atom which can holds one of the values `{ instant, duration}`. For the reason of compatibility we allow also the value `datetime` as an alias for an instant type.

The representations of `<timevalue>` are described in the next subsections.

### 4.1 String Representation

A <timevalue> of type `instant` can be represented as string in the following format:
`year-month-day[-hour:minute[:second[.millisecond]]] | undef`

The squared brackets indicate optionally parts, the vertical line separates exclusive alternatives. All contained values are integers which must be in the appropriate interval.

### 4.2 Real Representation

A <timevalue> can just be represented as an real value. The conversion beween the formats is described in section 4.5. This format is accepted by `instant` and `duration` types.

## 4.3 Julian Representation

A <timevalue> in the Julian representation is defined for `duration` types as follow:
`(day millisecond) | undef`
`day` can be any integer and millisecond is an integer in the interval [0,86400000[.

## 4.4 Gregorian Representation

In the Gregorian Representation (for `instants` only) a <timevalue> is written down as:
`(day month year [ hour minute [ second [ millisecond]]]) | undef`
Like in the string representation (see section 4.1), the squared brackets are not part of the representation but indicates optionally information. The values of the omitted parts are assumed to be zero. The triple (day, month, year) must be a valid date. The remaining values must be in the appropriate interval.

## 4.5 Conversions between Representations of Time

Both types of time can be represented by a tuple `(day, milliseconds)`. The Meaning for a duration type is clear. For an instant type this tuple is the difference to a fixed `NULL_DATE`. Because this `NULL_DATE` is given in the Julian calender, we denote this representation as Julian Representation. The conversion between `real` value and Julian representation is straightforward:
$JulToReal : int \times int \rightarrow \mathbb{R}$
$JulToReal(day, millisecond) = day + \frac{millisecond}{86400000}$
The used operators are working on real numbers.

The another direction is computed as follow. Note a possible lost of precision.
$RealToJul : \mathbb{R} \rightarrow (int, int)$
$RealToJul(instant) = (day, milli)$
where:

$$day = \begin{cases} int(instant) & \text{if } instant > 0 \text{ or } int(instant) = instant \\ int(instant) - 1 & otherwise \end{cases}$$

$$milli = \begin{cases} tmp & \text{if } instant > 0 \text{ or } int(instant) = instant \\ tmp + 86400000 & \text{otherwise} \end{cases}$$

where : tmp = int ((instant-int(instant))*86400000+0.5)

A quadrupel $(hour, min, sec, millisec)$ can be converted into an single integer representing all milliseconds using the following formula:
$millis : int \times int \times int \times int \rightarrow int$
$millis(h, min, sec, msec) = (((h \cdot 60) + min) \cdot 60 + sec) \cdot 1000 + msec$

The inverse direction can be computed as follow:
$splitmillis : int \rightarrow (int, int, int, int)$
$splitmillis(ms) = (h, min, sec, msec)$
where:

$$h = ms/3600000 \mod 24$$

$$min = ms/60000 \mod 60$$

$$sec = ms/1000 \mod 60$$

$$msec = ms \mod 1000$$

The functions for converting a Julian Date into a Gregorian date and vice versa are given in the appendix.

# 5 Spatio-temporal Values

Basically any spatio-temporal value of type <*moving_type*> is represented as a string:

(<unit$_1$> <unit$_2$> $\cdots$ <unit$_n$>) | undef

where <*unit$_i$*> is a unit value of the unit type corresponding to <*moving_type*> and $n \geq 0$.

A unit value of a type <*moving_type*> is represented as:

(<interval> <map_type_value>)

An <*interval*> value is represented as:

(<start> <end> <leftclosed> <rightclosed>)

where <*start*> and <*end*> are defined instances of type *instant* representing the start and end instant of the time interval, respectively, and <*leftclosed*> and <*rightclosed*> are defined boolean values defining if the interval is open (*false*) or closed (*true*) at the start or end time instant, respectively.

The representation of a <*map_type_value*> depends on the specific *type*.

## 5.1 moving(point)

The representation of a <*map_point_value*> is:

(<x$_1$> <y$_1$> <x$_2$> <y$_2$>)

where x$_i$ and y$_i$ are *Numeric* values. In a given Unit
((<start> <end> <leftclosed> <rightclosed>) (<x$_1$> <y$_1$> <x$_2$> <y$_2$>)) the point moves in the given interval from (x$_1$,y$_1$) to (x$_2$,y$_2$). The position of a moving point can be computed for a single unit using the following function:

$pos : unit(point) \times instant \rightarrow point$

$$pos(((s\ e\ l\ r)(x_1 y_1 x_2 y_2))), I) := \begin{cases} \text{undefined} & \text{if } I \notin (s\ e\ l\ r) \\ (x_1, y_1) & \text{if } s = e \textbf{ and } I \in (s\ e\ l\ r) \\ (x_I, y_I) & \text{otherwise} \end{cases}$$

where:

$$x_I = x_1 + \frac{I-s}{e-s} \cdot (x_2 - x_1)$$

$$y_I = y_1 + \frac{I-s}{e-s} \cdot (y_2 - y_1)$$

## 5.2 moving(points)

The representation of a <*map_points_value*> is:

(<map_point_value$_1$> <map_point_value$_2$> $\cdots$ <map_point_value$_n$>)

with $n > 0$. If the *movingpoints* value is empty for a given time interval, no unit referring to such a time interval appears in its representation.[1]

## 5.3 moving(line)

The representation of a <*map_line_value*> is:

(<map_seg_value$_1$> <map_seg_value$_2$> $\cdots$ <map_seg_value$_n$>)

where <*map_seg_value$_i$*> is the mapping value of a segment and n $>$ 0. If the *movingline* value is empty for a given time interval, no unit referring to such a time interval appears in its representation.

A <*map_seg_value*> is represented as:

( <*map_point_value$_1$*> <*map_point_value$_2$*>)

representing the <*map_point_value*> values of its end points.

**Restriction:** a <*map_seg_value*> must be defined over a plane in the 3D space *X*, *Y*, *Time*.

[1] This is the reason why $n \neq 0$.

## 5.4 moving(region)

The representation of a <*map_region_value*> is:

(<map_face_value$_1$> <map_face_value$_2$> $\cdots$ <map_face_value$_n$>)

where n $> 0$.

If the *movingregion* value is empty for a given time interval, no unit referring to such a time interval appears in its representation.

The representation of a <*map_face_value*> is the same as the spatial representation of a <*face*>, but in this case each vertex of its cycles is represented as a <*map_point_value*>.

**Restriction:** in a similar way as with the representation of a <*map_line_value*>, any segment defined by two consecutive vertices <*map_point_value$_i$*> and <*map_point_value$_{(i \bmod n)+1}$*> in a cycle of a <*map_face_value*> must be defined over a plane in the 3D space *X*, *Y*, *Time*.

## A Conversion from a Gregorian Date to Julian Date and vice versa

```
#include <math.h>
static const long NULL_DAY = 2451547;
// this corresponds to 3.1.2004 and must be a multiple of 7

/*
   The function ToJulian computes the Julian day number of the given
   Gregorian date + the reference time.Positive year signifies A.D.,
   negative year B.C.Remember that the year after 1 B.C. was 1 A.D.
   Julian day 0 is a Monday.
   This algorithm is from Press et al., Numerical Recipes
   in C, 2nd ed., Cambridge University Press 1992
*/

static long ToJulian(int year, int month, int day){
  int jy = year;
  if (year < 0)
     jy++;
  int jm = month;
  if (month > 2)
     jm++;
  else{
     jy--;
     jm += 13;
  }

  int jul = (int)(floor(365.25 * jy) + floor(30.6001*jm)
                  + day + 1720995.0);
  int IGREG = 15 + 31*(10+12*1582);
  // Gregorian Calendar adopted Oct. 15, 1582
  if (day + 31 * (month + 12 * year) >= IGREG){
     // change over to Gregorian calendar
     int ja = (int)(0.01 * jy);
     jul += 2 - ja + (int)(0.25 * ja);
  }
  return jul-NULL_DAY;
}

/*
   This function converts a Julian day to a date in the Gregorian calender.

   This algorithm is from Press et al., Numerical Recipes
   in C, 2nd ed., Cambridge University Press 1992

*/
static void ToGregorian(long Julian, int &year, int &month, int &day){
   int j=(int)(Julian+NULL_DAY);
   int ja = j;
   int JGREG = 2299161;
   /* the Julian date of the adoption of the Gregorian
      calendar
   */
```

```
    if (j >= JGREG){
    /* cross-over to Gregorian Calendar produces this
       correction
    */
       int jalpha = (int)(((float)(j - 1867216) - 0.25)/36524.25);
       ja += 1 + jalpha - (int)(0.25 * jalpha);
    }
    int jb = ja + 1524;
    int jc = (int)(6680.0 + ((float)(jb-2439870) - 122.1)/365.25);
    int jd = (int)(365 * jc + (0.25 * jc));
    int je = (int)((jb - jd)/30.6001);
    day = jb - jd - (int)(30.6001 * je);
    month = je - 1;
    if (month > 12) month -= 12;
    year = jc - 4715;
    if (month > 2) --year;
    if (year <= 0) --year;
}
```