

Computational Treatment of Temporal Notions The CTTN-System

Hans Jürgen Ohlbach

Institut für Informatik, Universität München
E-mail: ohlbach@lmu.de

Abstract. The CTTN-system is a computer program which provides advanced processing of temporal notions. The basic data structures of the CTTN-system are time points, crisp and fuzzy time intervals, labelled partitionings of the time line, durations, and calendar systems. The labelled partitionings are used to model periodic temporal notions, quite regular ones like years, months etc., partially regular ones like timetables, but also very irregular ones like, for example, dates of a conference series. These data structures can be used in the temporal specification language GeTS (GeoTemporal Specifications). GeTS is a functional specification and programming language with a number of built-in constructs for specifying customised temporal notions.

CTTN is implemented as a Web server and as a C++ library. This paper gives a short overview over the current state of the system and its components.

1 Introduction

In the CTTN-project we aim at a very detailed modelling of the temporal notions. These are, in particular, time points, crisp and fuzzy temporal intervals together with built-in as well as user definable relations between and operations on these intervals. Furthermore, there is support for various kinds of regular and irregular periodic temporal notions, again built-in ones as well as user definable ones. The possibilities range from very simple ones like seconds or minutes up to complex ones like Easter time or solar eclipses. A special specification and programming language GeTS (GeoTemporal Specifications [10]) allows applications and users to defined their own versions of temporal notions and to do all kinds of computations with them.

CTTN is *not* the implementation of a theoretical temporal logic, but it models the flow of time as it is perceived on our planet. It realizes the main concepts and operations underlying many temporal notions in natural language.

The key components of the CTTN-system consist of the modules depicted in Figure 1. The Service module at the bottom contains a large variety of application independent functions. The FuTI module (Fuzzy Time Intervals) [9, 8] contains the data structures and operations on time time points and crisp and

fuzzy time intervals. The largest module is the PartLib module (Partitioning Library). It contains the machinery for specifying and working with periodic temporal notions. Since calendar systems consist of such periodic temporal notions, a module for representing different calendar systems is also part of PartLib.

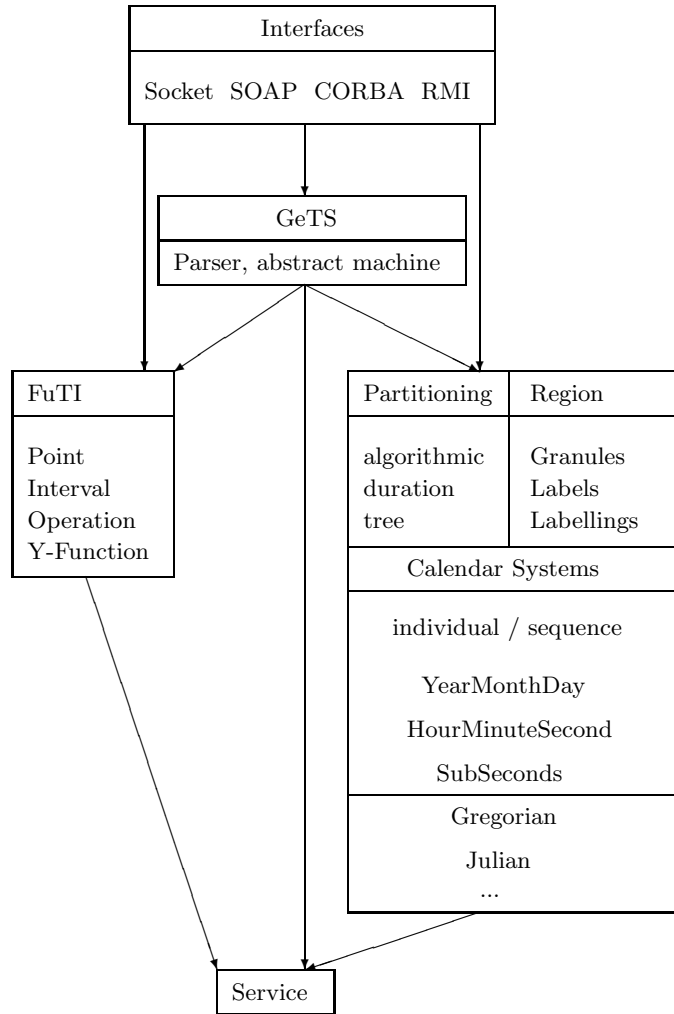


Fig. 1. The CTTN-System

The GeTS module implements a functional programming language with certain additional constructs for this application area. A flex/bison type parser and an abstract machine for GeTS has been implemented as part of the CTTN-system. GeTS is the first specification and programming language with such a

rich variety of built-in data structures and functions for GeoTemporal notions. In a first case study it has been used to define various versions of fuzzy interval-interval relations [8].

The basic interface to the CTTN system is socket based and implements the CTTN protocol. Prototypes of RMI, CORBA and SOAP interfaces have also been implemented, but not yet fully tested.

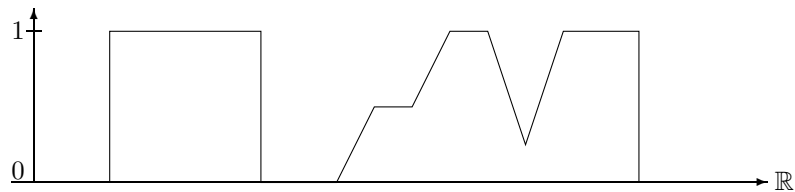
2 Time Points and Time Intervals in the FuTI-Module

The flow of time underlying most calendar systems corresponds to a time axis which is isomorphic to the real numbers \mathbb{R} . Since the most precise clocks developed so far, atomic clocks, measure the time in discrete units, it is sufficient to restrict the representation of concrete time points to *integers*. Therefore FuTI represents time points with integers, either with 64-bit integers, or with multiple precision integers (this is a compiler option). Within FuTI there is no assumption about the meaning of these integers, whether they are days, seconds, femtoseconds or not even time points¹.

Although FuTI represents time points only with integers, there is still the underlying assumption that the time axis is isomorphic to the real numbers. That means, for example, the interval between the time points 0 and 1 is not empty, but it is set of real numbers between 0 and 1.

The next important data type is that of time intervals. Time intervals can be crisp or fuzzy. With fuzzy intervals one can encode notions like ‘around noon’ or ‘late night’ etc. Since fuzzy intervals are more general and more flexible than crisp intervals, FuTI uses fuzzy intervals as basic interval data type.

Fuzzy intervals are usually defined through their membership functions [15, 4]. A membership function maps a base set to real numbers between 0 and 1. The base set for fuzzy time intervals is a linear time axis.



Crisp and Fuzzy Intervals

The fuzzy intervals can also be infinite. For example, the term ‘after tonight’ may be represented as a fuzzy distribution which rises from fuzzy value 0 at 6 pm until fuzzy value 1 at 8 pm and then remains 1 ad infinitum.

¹ A special component of FuTI, which was developed for another application allows for the representation of circular intervals like angles between 0 and 360 degrees. In this case the integers represent fractions of angular degrees.

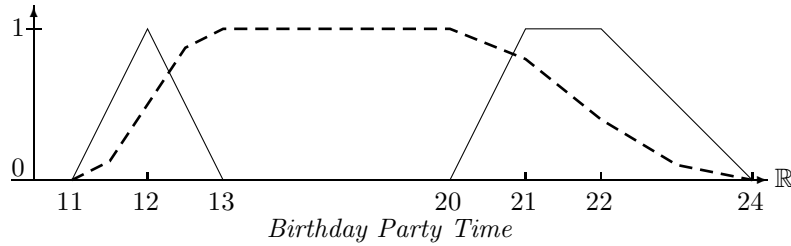


Fuzzy time intervals are realized in the FuTI-module as polygons with integer coordinates. The x -coordinates represent time points and the y -coordinates represent fuzzy values as integers between 0 and a maximum value (the default value is 1000). A normalised fuzzy value between 0 and 1 can then be obtained by dividing the integer y -coordinate by the maximum value. A y -coordinate of 500, for example, represents the normalised fuzzy value 0.5.

If the integers represent hours, one can, for example, represent the interval ‘around noon’ as the polygon $((11,0) (12,1000) (13,0))$. The membership function of the corresponding fuzzy interval starts at 11 o’clock with fuzzy value 0 and then rises linearly to fuzzy value 1 at noon. From there on it falls linearly to fuzzy value 0 at 1 pm.

FuTI provides a large collection of operations on these intervals. There are methods for accessing information about the intervals, the location of various parts of an interval, its size (which is the integral over the membership function), its components etc. There are methods for transforming the intervals, for example, hull computations, there are integration functions, fuzzification functions etc. There are also very general unary and binary transformation functions which can be parameterised with functions operating on the fuzzy values. All the set operations on fuzzy intervals, for example, are realized as transformations with functions on the fuzzy values. The transformations of the fuzzy membership functions need not be linear, i.e. they may transform straight lines into curved lines. The FuTI-module contains for these cases an approximation algorithm which approximates curved lines by polygons.

Example 1 (Birthday Party Time). This example illustrates some of the operations which are possible with the FuTI-module. Consider the statement “the birthday party for took place *from around noon until early evening* of 20/7/2003”. The corresponding fuzzy interval could be generated by integrating the fuzzy interval for ‘around noon’ in positive direction, integrating the fuzzy interval for ‘early evening’ in negative direction and then intersecting the two integrals. The resulting fuzzy set is:



A GeTS specification of this example is given in Example 9. ■

3 Periodic Temporal Notions in the PartLib–Module

The PartLib module offers powerful machinery for specifying and working with periodic temporal notions. The basic concept is the concept of the *partitionings of the time axis*. Since most periodic temporal notions, for example, days, yield infinite partitionings of the time axis, PartLib offers different versions of finite representations of these infinite structures. The operations on the infinite structures are turned into operations on the corresponding finite representations.

Partitions can be *labelled*, e.g., with ‘Monday’, ‘Tuesday’ etc. Partitionings with labels can be comprised in different ways to different structures. For example, from the day–partitioning and the corresponding labelling one can derive the structure which corresponds to ‘all Mondays’ or to ‘all non-Mondays’. If the labels are organised in a hierarchy, for example, Monday,...,Friday are all ‘Workdays’ and Saturday and Sunday are ‘Weekenddays’ one can derive the notion of ‘all Workdays’. Since there are a number of further ways to derive new substructures of the time axis from labelled partitionings, all these ways are comprised into the concept of *region structure* (see Sec. 3.5). A region structure is essentially a subset of a partitioning of the time axis. Many operations in the CTTN system work with the more general region structures instead with partitionings.

3.1 Partitionings of the Time Axis

Most basic time units of calendar systems, years, months etc., are essentially partitionings of the time axis. Other periodical temporal notions, for example, semesters, school holidays, sunsets and sunrises etc., can also be modelled as partitionings.

A partitioning of the real numbers \mathbb{R} may be, for example, $(\dots, [-100, 0[, [0, 100[, [100, 101[, [101, 500[, \dots)$. The intervals in the partitionings need not be of the same length (because time units like years are not of the same length either). The intervals can, however, be enumerated by integers (their *coordinates*). For example, we could have the following enumeration

$$\begin{array}{ccccccc} \dots & [-100 & 0[& [0 & 100[& [100 & 101[& [101 & 500[& \dots \\ \dots & -1 & & 0 & & 1 & & 2 & & \dots \end{array}$$

The enumeration of partitions, i.e. their coordinates, are a very useful means for concrete computations. It turned out, however, that in some cases instead of integer coordinates, certain other structures which are isomorphic to integers are more useful. An example for a structure which is isomorphic to the integers are the paths in an infinite tree. Therefore PartLib has introduced the concept of *Partition Access Specifier (PASp)* as a generalisation of the integer coordinates.

Definition 1 (Partitioning). A partitioning P of the time axis in PartLib is a sequence

$$\dots [t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \dots$$

of non-empty half open intervals in \mathbb{R} with integer boundaries such that $t_i < t_{i+1}$ for all i .

The partitioning may be finite at one or both sides, i.e. $]-\infty, t_0[, \dots, [t_n, +\infty[$ is allowed.

A Partition Access Specifier Structure is a set of objects which is isomorphic to the integers.

A coordinate mapping c is a bijective mapping between a partitioning and a Partition Access Specifier Structure (or a part of it if the partitioning is finite) such that if partition p is before partition q then $c(p) < c(q)$. ■

The choice of half open intervals of the kind $[t_i, t_{i+1}[$ as partitions was arbitrary. It means that, for example, Midnight always belongs to the next day.

3.2 Labelled Partitionings

The partitions in CTTN can be *labelled*. The labels are just names for the partitions like in the following example.

Example 2 (The Labelling of Days). We count the time in seconds beginning with January 1st 1970. This was a Thursday. Therefore we choose as labelling for the day partitioning

$$L \stackrel{\text{def}}{=} Th, Fr, Sa, Su, Mo, Tu, We.$$

The following correspondences are obtained:

$$\begin{array}{llll} \text{time :} & \dots [-86400, 0[& [0, 86400[& [86400, 172800[\dots \\ \text{coordinate :} & \dots & -1 & 0 & 1 & \dots \\ \text{label :} & \dots & We & Th & Fr & \dots \end{array}$$

This means, for example, $L(-1) = We$, i.e. December 31 1969 was a Wednesday. ■

Labels are different to coordinates because different partitions can have the same label (e.g., all Mondays). Labellings can be used for three purposes. The first purpose is to get access to the partitions via their names (labels). One can use these names in various GeTS functions. The second purpose is to associate

partitions with further attributes. The labels can, for example, serve as keys into databases. The third purpose is to use the labels for grouping partitions together into *regions*. An example is the set of all Mondays. This is no longer a partitioning of the time axis because there are gaps between the Mondays.

Definition 2 (Labels). *A set of labels in PartLib is just an arbitrary finite or infinite set²*

A label hierarchy is a binary relation \sqsubseteq which orders the labels in a tree.

A labelling of a PartLib partitioning is a possibly partial mapping from the partitions into the set of labels. ■

Since a labelling can be partial, not all partitions need to have labels. As an example, where this makes sense, consider the partitioning of hours and the labelling which associates the label ‘working hour’ with all hours between 8 am and noon and all hours between 1 pm and 5 pm. The other hours don’t have labels. This labelling specifies implicitly the concept of ‘working day’, the concept of ‘lunch time’, and the concept of ‘after work’. These implicit definitions can be made explicit in PartLib by turning them into *region structures* (see below).

3.3 Specification of Partitionings

Partitionings have a finite representation in PartLib. There are the following representations for partitionings.

Algorithmic Partitionings

This type of partitionings is mainly used for modelling the basic time units of calendar systems, years, months etc. The specification consists of an offset against time point 0, an average length of the partitions, and a correction function which corrects the average length to the actual length.

Example 3 (Basic Time Units for the Gregorian Calendar).

The specification of the basic time units as algorithmic partitionings for the Gregorian Calendar are:

second: average length: 1, offset: 0, correction function: $\lambda(n)0$.

minute: average length: 60, offset: 0, correction function: $\lambda(n)0$.

hour: average length: 3600, offset: 0, correction function: $\lambda(n)0$.

day: average length: 86400, offset: 0, correction function: $\lambda(n) - 3600 \cdot h$ if the day n is during the daylight saving time period, 0 otherwise.

The number h is usually 1 (for 1 hour). Exceptions are, for example, the year 1947 in Germany, where in the night of 1947/5/11 the clock was set forward a second time by 1 hour such that the offset against standard time was 2 hours.

week: average length: 604800, offset -259200³, correction function: again, this function has to return an offset of $-3600 \cdot h$ for the weeks during the daylight saving time periods.

² Labels are in fact instances of subclasses of a class *Label*.

³ This is because the first of January 1970 is Thursday.

month: average length: 2592000 (30 days), offset 0, correction function: this function has to deal with the different length of the months and the daylight saving time regulations.

year: average length: 31536000 (365 days), offset 0, correction function: this function has to deal with leap years only. The effects of daylight saving time regulations are averaged out over the year. ■

Duration Partitionings

They are specified by an anchor time and a sequence of ‘*durations*’.

For example, I could define ‘my weekend’ as a *duration partitioning* with anchor time 2004/7/23, 4 pm (Friday July, 23rd, 2004, 4 pm) and durations: (‘8 hour + 2 day’, ‘4 day + 16 hour’). The first interval would be labelled ‘weekend’.

A simpler example is the notion of a semester at a university. In the Munich case, the dates could be: anchor time: October 2000. The durations are: 6 months (with label ‘winter semester’) and 6 months (with label ‘summer semester’). This defines a partitioning with partition 0 starting at the anchor time, and then extending into the past and the future. The first partition in this example is the winter semester 2000/2001.

The units for the duration are in fact region structures, and not just partitionings. Thus, one can, for example, define durations in terms of *granules*. An example is ‘3.5 working_days + 1.5 weekends’.

smallskipDate Partitionings

In this version we provide the boundaries of the partitions by concrete dates. Therefore the partitioning can only cover a finite part of the time line.

An example could be the dates of the Time conferences: 1994/5/4 Time94 1994/5/4 gap 1995/4/26 Time95 1995/4/26 ... 2004/7/1 Time04 2004/7/3.

Since the intervals between two adjacent dates determine durations, date partitionings are in fact special cases of duration partitionings, and this is how they are treated in PartLib.

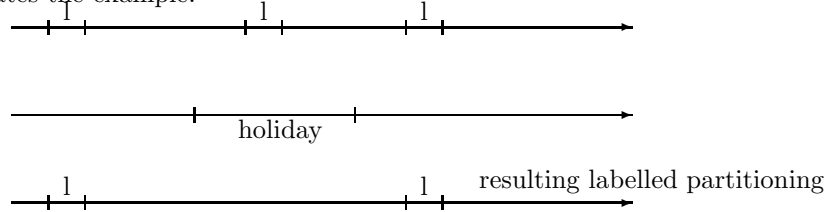
Intersection Partitionings

They combine two previously defined partitionings by intersecting their partitions. If the two original partitionings are labelled then a new labelling can be computed by means of *mapping rules* for labels.

As an example, suppose there is a partitioning *p1* representing the lecture course *l*, say every Wednesday from 10 am until 12 am. There is a second partitioning *p2* which represents public holidays. *p2* is labelled with the holiday names (Easter, Christmas etc.) The holiday name labels are all sub-labels in a label hierarchy with top element ‘holiday’. The partitioning which represents the lecture time without the public holidays can be generated by intersecting *p1* and *p2* with the following mapping rules

$$\begin{aligned} l * holiday &\mapsto gap; \\ l * gap &\mapsto l; \\ gap * holiday &\mapsto gap \end{aligned}$$

with the extra provision that adjacent partitions without labels are comprised into a single partition. ‘gap’ stands for the empty label. The following picture illustrates the example.

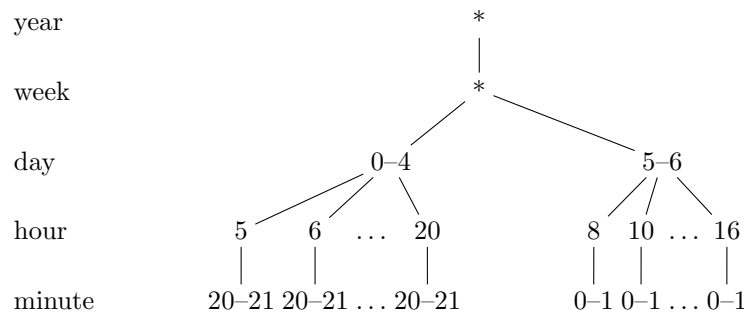


Tree Partitionings

This type of specification for partitionings can be used when concrete dates are involved. Typical examples are bus timetables. A tree partitioning is given by a *Partition Access Format* (PAF) and a *Partition Access Tree* (PAT). The PAF determines a kind of calendar to be used for interpreting the nodes in the PAT [11].

Example 4 (for a Tree Partitioning Specification). A typical PAF is the standard date format year/week/day/hour/minute/second.

The following PAT may define a bus schedule.



It specifies the following bus schedule: every year, every week, every work day (0-4), there is a bus at 5:20 – 5:21 (2 minutes stay at the bus stop), 6:20 – 6:21 until 20:20 – 20:21, and at the weekends (days 5,6) there is a bus every hour from 8 until 16 hours.

The nodes in the PAT determine an offset from the start of the region given by the corresponding position in the PAF. There are four different node types:

NumberRange nodes. They specify concrete number sets, for example, 4-6,10-12 specifies the set {4, 5, 6, 10, 11, 12}

NumberIterator nodes. They specify iterators like, for example, in a ‘for loop’. The iterator is given by a start value, a step value and a number of iterations. For example, start = 1, step = 2, iterations = 5 specifies the set {1, 3, 5, 7, 9}

LabelRange nodes. They specify concrete label sets, for example, March-May, August specifies the set {2, 3, 4, 7} (January is month 0).

LabelIterator nodes. They specify labels by giving a label together with a number iterator. For example, startLabel = 'L' start = 2, steps = 10, iterations = 5 The loop starts with the second occurrence of L and then continues 5 times in steps of 10 partitions with this label, 5 iterations.

In all four cases it is also possible to interpret the numbers as distances from the end of a partition. For example, if the day partition is right below the month partition in the corresponding Partition Access Format, and the backwards flag is set to true, then the number 0 at the day level is interpreted as the very last day in the given month.

The specification of a partitioning can be quite complex and require a lot of data. Therefore for each partitioning type, except for algorithmic partitionings, there is a corresponding XML document type for specifying a partitioning. After the CTTN interface has read and parsed such an XML specification one can use them in the same way as the built-in partitionings for calendar systems.

3.4 Leap Seconds

To compensate for the slowing down of the earth's rotation, since 1971 every few years a leap second has been introduced. The last minute in the year where a leap second has been inserted has 61 seconds instead of 60 seconds. This has an effect on all partitionings above the level of seconds. It would be very complicated and error prone to integrate the effect of leap seconds in all these partitionings. As an alternative, this phenomenon is taken care of by separating the reference time into a *global reference time* and a *local reference time*. The global reference time counts the seconds as they are. It knows nothing about leap seconds. The local reference time shrinks the leap seconds to 0 length. That means the last minute in the years where a leap second has been inserted has still 60 seconds in the local reference time. The extra second occurs only in transition to the global reference time. This way the leap second calculations have been concentrated in a single place, the transition between local and global reference time. All other partitioning dependent calculations can ignore leap seconds.

3.5 Region Structures

The labels which can be attached to the partitionings generate a variety of new substructures of the time axis which are no longer partitionings because there can be gaps between the corresponding time intervals. Since periodic temporal notions with gaps are much more frequent than partitionings, the new concept of *region structures* has been introduced.

Region structures are like partitionings, but there are two essential differences

- there are gaps allowed between two neighbouring regions
- there are gaps allowed even within a region. An example is 'working day' from 8 am until 5 pm with a lunch break from 12 am until 1 pm.

CTTN distinguishes the following types of region structures:

PartitioningRegion: each partition is a region. Labels are ignored.

LabelRegion: are determined by a label (possibly within a label hierarchy). For example, the LabelRegion with label ‘weekendday’ of a day partitioning (with sub-labels Saturday and Sunday below weekendday) would join the days of the weekends into a region. A Saturday is a region, followed by the following Sunday, followed by the following Saturday etc.

LabelBlock: is similar to a LabelRegion. The difference is that neighbouring partitions with the given label form one region. A LabelBlock with labels ‘weekendday’ (see above) would join Saturdays and Sundays into one single region.

LabellingRegion: declares a whole label sequence as a region. For example, the labelling ‘Monday’, ‘Tuesday’, ... ‘Sunday’ of the day partitioning comprises a whole week into a single region.

GapBlock: A GapBlock comprises all adjacent partitions without labels into one region.

Granule: A granule is a sequence of partitions with the same label possibly interrupted by partitions without label. As an example, consider the hour partitioning where the hours between 8 and 12 and between 13 and 18 hours are labelled ‘working hour’. The corresponding granule comprises the working hours into one, in this case non-convex, region. This concept is very much like the concept of granules found in the literature [1].

As soon as a labelling has been attached to a partitioning, all these types of region structures are available as concrete data types, and a common API is available via the superclass ‘Region’. Typical examples for the API are methods which move from a given region to the next region, methods which move from a given time point n regions forward or backward (n may be fractional), methods which measure time intervals in terms of region length etc.

3.6 Calendar Systems

A *calendar system* in the CTTN-system is a set of partitionings or region structures, for example the partitionings for seconds, minutes, hours, weeks, months and years, together with some extra data and methods. Dershowitz and Reingold’s ‘calendrical calculations’ are used here [3] for computing the details down to the level of days. In addition PartLib models all the nasty features of real calendar systems, in particular leap seconds and daylight saving time schemes (in a submodule *DLST*). Calendar systems can be arranged in sequences, for example, the sequence consisting of the Julian calendar system until 4th of October 1582 followed by the Gregorian system. Another example of a sequence of calendar systems in PartLib could be a sequence of calendars and time zones a traveller encounters when he travels around the world.

The Calendar submodule in PartLib has predefined general classes for years/months/days, for hours/minutes/seconds and for sub-seconds. Using these classes it requires very little code to add new calendar systems.

4 The GeTS Language

The PartLib module has, via the XML-interface, mechanisms for integrating user defined periodic temporal notions. Not all temporal notions and computations, however, have to do with periodicities. The GeoTemporal Specification Language GeTS has therefore been added as a general purpose language for working with temporal notions. The design of the GeTS language was influenced by the following considerations:

1. Although the GeTS language has many features of a functional programming language, it is not intended as a general purpose programming language. It is a specification language for temporal notions, however, with a concrete operational semantics.
2. The parser, compiler, and in particular the underlying GeTS abstract machine are not standalone systems. They must be embedded into a host system which provides the data structures and algorithms for time intervals, partitionings etc., and which serves as the interface to the application. GeTS provides a corresponding application programming interface (API).
3. The language should be simple, intuitive, and easy to use. It should not be cluttered with too many features which are mainly necessary for general purpose programming languages.
4. The last aspect, but even more the point before, namely that GeTS is to be integrated into a host system, were the main arguments against an easy solution where GeTS is only a particular module in a functional language like SML or Haskell. The host system was developed in C++ (it could also be Java, but multiple precision integers are more efficient in C++). Linking a C++ host system to an SML or Haskell interpreter for GeTS would be more complicated than developing GeTS in C++ directly. The drawback is that features like sophisticated type inferencing or general purpose data structures like lists or vectors are not available in the current version of GeTS.
5. Developing GeTS from scratch instead of using an existing functional language has also an advantage. One can design the syntax of the language in a way which better reflects the semantics of the language constructs. This makes it easier to understand and use. As an example, the syntax for a time interval constructor is just $[expression_1, expression_2]$.

The GeTS language is a strongly typed functional language with a few imperative constructs. Here we can give only a flavour of the language. The technical details are in [10].

Example 5 (tomorrow). The definition

```
tomorrow = partition(now(),day,1,1)
```

specifies ‘tomorrow’ as follows: `now()` yields the time point of the current point in time. `day` is the name of the day partitioning. Let i be the coordinate of the day-partition containing `now()`. `partition(now(),day,1,1)` computes the

interval $[t_1, t_2[$ where t_1 is the start of the partition with coordinate $i + 1$ and t_2 is the end of the partition with coordinate $i + 1$. Thus, $[t_1, t_2[$ is in fact the interval which corresponds to ‘tomorrow’.

In a similar way, we can define

```
this_week(Time t) = partition(t,week,0,0).
```

The time point t , for which the week is to be computed, is now a parameter of the function. ■

Example 6 (Christmas). The definition

```
christmas(Time t) =
  dLet year = date(t,Gregorian_month) in
    [time(year|12|25,Gregorian_month),
     time(year|12|27,Gregorian_month)]
```

specifies Christmas for the year containing the time point t . ■

`date(t, Gregorian_month)` computes a date representation for the time point t in the date format `Gregorian_month` (`year/month/day/hour/minute/second`). Only the year is needed. `dLet year = ...` therefore binds only the year to the integer variable `year`. If, for example, in addition the month is needed one can write `dLet year|month = date(...`

`time(year|12|25, Gregorian_month)` computes $t_1 =$ begin of the 25th of December of this year. `time(year|12|27, Gregorian_month)` computes $t_2 =$ begin of the 27th of December of this year. The expression `[..., ...]` denotes the half open interval $[t_1, t_2[$.⁴ The result is therefore the half open interval from the beginning of the 25th of December of this year until the end of the 26th of December of this year.

Example 7 (Point–Interval Before Relation). The function

```
PIRBefore(Time t, Interval I) =
  if (isEmpty(I) or isInfinite(I,left)) then false
  else (t < point(I,left,support))
```

specifies the standard crisp point–interval ‘before’ relation in a way which works also for fuzzy intervals. ■

If the interval I is empty or infinite at the left side then `PIRBefore(t,I)` is `false`, otherwise t must be smaller than the left boundary of the support of I . Now we define a parameterised fuzzy version of the interval–interval before relation.

⁴ Crisp intervals in CTTN are always half open intervals `[..., ...]`. Sequences of such intervals, for example, sequences of days, can therefore be used to partition a time period. The syntactic representation of these intervals in GeTS is `[..., ...]` and not `[..., ...]` because this simplifies the grammar and the parser considerably.

Example 8 (Fuzzy Interval–Interval Before Relation). A fuzzy version of an interval–interval before relation could be

```
IIRFuzzyBefore(Interval I, Interval J, Interval->Interval B) =
case
  isEmpty(I) or isEmpty(J) or
    isInfinite(I,right) or isInfinite(J,left)      : 0,
  (point(I,right,support) <= point(J,left,support)) : 1,
  isInfinite(I,left) : integrateAsymmetric(intersection(I,J),B(J))
else integrateAsymmetric(I,B(J))
```

■

The input are the two intervals I and J and a function B which maps intervals to intervals. B is used to compute for the interval J an interval $B(J)$, which represents the degree of ‘beforeness’ for the points before J .

The function first checks some trivial cases where I cannot be before J (first clause in the `case` statement), or where I definitely is before J (second clause in the `case` statement). If I is infinite at the left side then $\int (I \cap J)(x) \cdot B(J)(x) dx / |I \cap J|$ is computed to get a degree of ‘beforeness’, at least for the part where I and J intersect. If I is finite then $\int I(x) \cdot B(J)(x) dx / |I|$ is computed. This averages the degree of a point–interval ‘beforeness’, which is given by the product $I(x) \cdot B(J)(x)$, over the interval I .

The next example is a parameterised version of an ‘Until’ operator. It can be used to formalise expressions like ‘from around noon until early evening’. The parameters are operators which manipulate the front and back end of the intervals, together with a complement operator.

Example 9 (Until). an ‘Until’ operator can be defined in GeTS:

```
Until(Interval I, Interval J, Side s1, Side s2,
      (Interval*Interval)->Interval Ints,
      Interval->Interval Ep, Interval->Interval En,
      Interval->Interval C) =
  if (s1 == left) then
    (if (s2 == left) then Ints(Ep(I),C(Ep(J)))
     else Ints(Ep(I),En(J)))
  else
    (if (s2 == left) then Ints(C(En(I)),C(Ep(J)))
     else Ints(C(En(I)),En(J)));
```

The birthday party example (Example 1) could be specified using this function:

```
Birthdayparty(I,J)
= Until(I, J, left, right,
  lambda(Interval K, Interval L) intersection(K,L),
  lambda(Interval K) integrate(K,positive),
  lambda(Interval K) integrate(K,negative),
  lambda(Interval K) complement(K)).
```

■

5 The Web-Interface

CTTN is a collection of C++ classes and methods which can be used in any other C++ program. There is, however, also a command interface which is realized as a web server. It communicates with a client through a socket. There is a group of commands for uploading application specific definitions of temporal notions in the GeTS language and in the specification language for labelled partitionings. There are also commands for working with instances of these temporal notions, particular time intervals, particular partitionings, particular calendar systems etc.

6 Extensions of the CTTN-System

A number of extensions of the CTTN-system are on the agenda. The most important one is the inclusion of constraint reasoning for ‘floating’ time intervals. The expression ‘two weeks between Christmas and Easter’, for example, cannot be represented so far, because the precise location of these two weeks are not known. Here we need to invoke constraints and constraint reasoning. Since the basic intervals are fuzzy intervals, the constraint calculus must also be able to deal with fuzziness. There are some approaches in the direction of fuzzy temporal reasoning [5, 14, 6] and fuzzy constraint networks [13, 7] which might be usable for the CTTN-system. Temporal constraint reasoning without taking fuzziness into account is certainly also very useful and should be integrated into the system [2].

Another extension is a context module. A simple example for context information which is useful for an application of the CTTN-system are the specification of time zones. Time zones are submitted to the current CTTN-system as offsets to GMT time. It would, however, be much more user friendly, if there would be an automatic mapping of countries or regions to time zones.

A third extension is a link to a system which represents *named entities*. The phrase ‘after the Olympic games in Rome’, for example, can only be analysed if some dates about the Olympic games in Rome are available. We are currently working on a link to the EFGT net, which stores named entities in a three dimensional context of thematic fields, geographic regions and time periods [12].

More details about the CTTN-system are available at the CTTN homepage: <http://www.pms.ifl.lmu.de/CTTN>.

Acknowledgements

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REVERSE number 506779 (cf. <http://reverse.net>).

References

1. C. Bettini and R.D.Sibi. Symbolic representation of user-defined time granularities. *Annals of Mathematics and Artificial Intelligence*, 30:53–92, 2000. Kluwer Academic Publishers.
2. François Bry, Frank-André Rieß, and Stephanie Spranger. A Reasoner for Calendric and Temporal Data. Forschungsbericht/research report PMS-FB-2005-18, Institute for Informatics, University of Munich, 2005.
3. Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations*. Cambridge University Press, 1997.
4. Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.
5. L. Godo and L. Vila. Possibilistic temporal reasoning based on fuzzy temporal constraints. In Chris S. Mellish, editor, *IJCAI'95: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1916–1922. IJCAI, 1995.
6. I. Navarette M.A. Cardenas and R. Marin. Efficient resolution mechanism for fuzzy temporal constraint logic. In *TIME'2000: Proc. of the Seventh International Workshop on Temporal Representation and Reasoning*, pages 39–46. IEEE Press, 2000.
7. Roque Marín, M. A. Cárdenas Viedma, M. Balsa, and J. L. Sanchez. Obtaining solutions in fuzzy constraint networks. *Int. J. Approx. Reasoning*, 16(3-4):261–288, 1997.
8. Hans Jürgen Ohlbach. Relations between fuzzy time intervals. In *Proceedings of 11th International Symposium on Temporal Representation and Reasoning, Tati-houi, Normandie, France (1st–3rd July 2004)*, pages 44–51. IEEE Computer Society, 2004. See also <http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-33>.
9. Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-26>.
10. Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-29>.
11. Hans Jürgen Ohlbach. Periodic temporal notions as ‘tree partitionings’. Forschungsbericht/research report PMS-FB-2006-11, Institute for Informatics, University of Munich, 2006.
12. Klaus U. Schulz and Felix Weigel. Systematics and architecture for a resource representing knowledge about named entities. In Jan Maluszynski François Bry, Nicola Henze, editor, *Principles and Practice of Semantic Web Reasoning*, pages 189–208, Berlin, 2003. Springer-Verlag.
13. L. Vila and L. Godo. On fuzzy temporal constraint networks. *Mathware and Soft Computing*, 3:315–334, 1994.
14. L. Vila and L. Godo. Query-answering in fuzzy temporal constraint networks. In Chris S. Mellish, editor, *FUZZ-IEEE'95: IEEE International Conference on Fuzzy Systems Yokohama*, volume 1, pages 43–48, 1995.
15. L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.