Computational Treatment of Temporal Notions The CTTN-System

Hans Jürgen Ohlbach

Institut für Informatik, Universität München
email: ohlbach@lmu.de

Abstract. The CTTN-system is a computer program which provides advanced processing or 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 customized 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]

1 Introduction and Motivation

In the CTTN-project we aim at a very detailed modelling of the temporal notions which can occur in semi-structured data. The CTTN-system consists of a kernel and several modules around the kernel. The kernel itself consists of several layers. At the bottom layer there are a number of basic datatypes for elementary temporal notions. These are time points, crisp and fuzzy time intervals [4, 7] and partitionings for representing periodical temporal notions like years, months, semesters etc. [6]. The partitionings can be specified algorithmically or algebraically. The algorithmic specifications allows one to encode phenomena like leap seconds, daylight savings time regulations, the Easter date, which depends on the moon cycle etc. Partitionings can be arranged to form 'durations, e.g. '2 year + 1 month, but also '2 semester + 1 month, where *semester* is a user defined partitioning. Sets of partitionings, together with certain procedures, form a *calendar*. The Gregorian calendar in particular can be formalized with the partitionings for years, months, weeks, days, hours, minutes and seconds.

The second layer uses the functions and relations of the first layer as building blocks in the specification language GeTS ('GeoTemporal Specifications'

[1] This is an extended abstract of a talk given at the Dagstuhl Seminar 05151, 'Annotating, Extracting and Reasoning about Time and Events', April 2005.

[5]). It is essentially 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.

The third layer consists of a command interface to the CTTN-system which can be accessed via IP/TCP. Other interfaces, RMI, CORBA, SOAP etc. are in principle possible, but not yet realized.

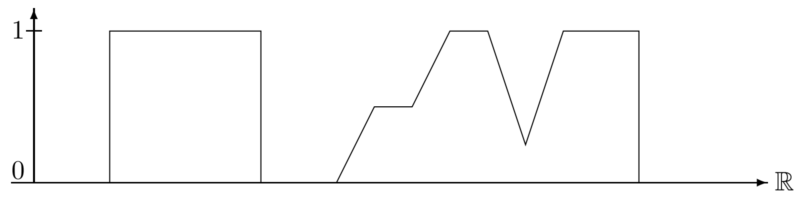
CTTN is *not* the implementation of a theoretical temporal logic, but it models the flow of time as it is perceived on our planet. It realizes the main concepts and operations underlying many temporal notions in natural language.

2 Time Points and Time Intervals

The flow of time underlying most calendar systems corresponds to a time axis which is isomorphic to the real numbers $\mathbb{R}$. Therefore CTTN takes as time points just real numbers. 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*. In the standard setting these integers count the *seconds* from the Unix epoch, which is January 1st 1970. Nothing significant changes, however, if the meaning of these integers is changed to count, for example, femtoseconds from the year 1.

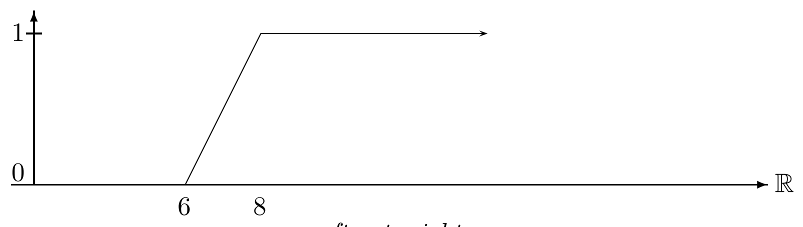
The next important datatype is that of time intervals. Time intervals can be crisp or fuzzy. With fuzzy intervals one can encode notions like ‘around noon or ‘late night etc. This is more general and more flexible than crisp intervals. Therefore the CTTN-system uses fuzzy intervals as basic interval datatype.

Fuzzy Intervals are usually defined through their membership functions [9, 3]. 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, isomorphic to the real numbers.



Crisp and Fuzzy Intervals

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



after tonight

Fuzzy time intervals are realized in the FuTI-library. Besides the pure datatype definitions (the membership function of a fuzzy interval is realized as a polygon with integer coordinates), it 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, integration functions, fuzzification functions etc. There are also very general unary and binary transformation functions which can be parameterized 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.

3 Partitionings

The CTTN-system uses the concept of *partitionings* of the real numbers to model periodical temporal notions. In particular, the basic time units years, months etc. are realized as partitionings. 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, $(..., [-100, 0[, [0, 100[, [100, 101[, [101, 500[, ...)$. 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 natural numbers (their *coordinates*). For example, we could have the following enumeration

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

Calendar Systems

A *calendar* in the CTTN-system is a set of partitionings, for example the partitionings for seconds, minutes, hours, weeks, months and years, together with some extra data and methods. Dershowitz and Reingolds 'calendrical calculations are used here [2]. The calendar systems in CTTN model all the nasty features of real calendar systems, in particular leap seconds and daylight saving time schemes.

The partitionings in CTTN can represent infinite partitionings of the real numbers. This is suitable to model, for example, years. They can, however, also be used to represent *finite* sequences of intervals. Examples are the school holidays in Bavaria from 1970 until 2006. CTTN extrapolates these intervals in a certain way to get an infinite partitioning. This simplifies the algorithms considerably, but it may yield unwanted results for time points where the partitioning is not meant for.

Therefore one can define boundaries for the validity of the partitionings. These boundaries have no influence on the computations, but they can be checked with special functions in the GeTS language.

The CTTN-system uses *labelled partitionings.* The labels are names for the partitions. They can be used for two purposes. The first purpose is to get access to the partitions via their names (labels). For example, the labels for the ‘day partitioning can be ‘Monday, ‘Tuesday etc., and one can use these names in various GeTS functions. The second purpose is to use the labels to group partitions together to so called *granules* [1]. The concept of ‘working day, for example, can be modelled by taking an ‘hour partitioning, and attach labels ‘working_hour and ‘gap to the hour partitions. Groups of hour partitions labelled ‘working_hour yield a working day. The working days can be interrupted by ‘gap partitions, for example to take ‘lunch time out of a ‘working day.

Definition 1 (Labels and Granules). *A* labelling *L is a finite sequence of strings $l_0, \ldots, l_{n-1}$. The label* `gap` *has a special meaning.*

A labelling L can now be very easily attached to a partitioning: the partition with coordinate i gets label $L(i \text{ mod } n)$.

A granule *is a sequence $p_i, \ldots, p_{i+k}$ of partitions such that: (1) the labels of p_i and p_{i+k} are not* `gap`*; (2) the labels of $p_i, \ldots, p_{i+k}$ which are not* `gap` *are the same, and (3) $i \text{ mod } n < (i+k) \text{ mod } n$.* ■

Example 1 (The Labelling of Days). The origin of the reference time is again January 1^{st} 1970. This was a Thursday. Therefore we choose as labelling for the day partitioning

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

The following correspondences are obtained:

time :	...	$[-86400, 0[$	$[0, 86400[$	$[86400, 172800[$	...
coordinate :	...	-1	0	1	...
label :	...	We	Th	Fr	...

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

The partitionings are the mathematical model of periodic time units, such as years, months etc. This offers the possibility to define *durations.* A duration may, for example, be ‘3 months + 2 weeks. Months and weeks are represented as partitionings, and 3 and 2 denote the number of partitions in these partitionings. The numbers need not be integers, but they can be arbitrary real numbers.

A duration can be interpreted as the length of an interval. In this case the numbers should not be negative. A duration, however, can also be interpreted as a time shift. In this interpretation negative numbers make perfect sense. $d = -2\ week + 3\ month$, for example, denotes a backward shift of 2 weeks followed by a forward shift of 3 months.

4 The GeTS Language

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++. 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 [5].

Example 2 (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 3 (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 $[t_1, t_2[$.[2] 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 4 (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 parameterized fuzzy version of the interval–interval before relation.

Example 5 (Fuzzy Interval–Interval Before Relation). A fuzzy version of an interval–interval before relation could be

[2] Crisp intervals in CTTN are always half-open intervals $[\ldots,\ldots[$. 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.

```
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`.

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. The Web interface is currently being developed and not yet documented.

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.

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. Timezones 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 timezones.

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 date about the Olympic games in Rome are available. We are currently working at a link to the EFGT net, which stores named entities in a three dimensional context of thematic fields, geographic regions and time periods [8].

More details about the CTTN-system will soon be available at the CTTN homepage: http://www.pms.ifi.lmu.de/CTTN.

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. Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations*. Cambridge University Press, 1997.
3. Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.
4. 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.
5. 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.
6. Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings of the real numbers – the PartLib library. Research Report PMS-FB-2005-28, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-28.
7. Hans Jürgen Ohlbach. Relations between fuzzy time intervals. Research Report PMS-FB-2005-27, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-27.
8. Klaus U. Schulz and Felix Weigel. Systematics and architecture for a resource representing knowledge abo ut named entities. In Jan Maluszynski Francois Bry, Nicola Henze, editor, *Principles and Practice of Semantic Web Reasoning*, pages 189–208, Berlin, 2003. Springer-Verlag.
9. L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.