

About Real Time, Calendar Systems and Temporal Notions

Hans Jürgen Ohlbach (h.ohlbach@doc.ic.ac.uk)*

*Department of Computing, Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ*

(Received:; Accepted:)

Abstract. A specification language is presented for describing every day temporal notions, like weekends, holidays, office hours etc. The time model underlying this language exploits in a particular way the well investigated algorithms for calendar systems currently in use. All peculiarities and irregularities of the calendar systems, like time zones, leap years, daylight savings time are respected. The specification language and the underlying temporal model allows us to convert between arbitrary calendar systems and to decide a number of questions, for example whether a certain point in time is within the time intervals specified by a term in this language. Besides neglecting the phenomena of relativity theory, there are no idealizing assumptions. The language is the basis for a temporal logic with quantifiers over real time temporal notions.

1. Introduction

It would not be the first time that I tried to phone a colleague in another country, maybe on another continent, I look up into my database, get his or her office-phone number, dial it, and get no answer. Maybe I forgot the time shift between different time zones, and it is actually in the middle of the night over there, or in this country there is just a public holiday, or it is lunch time, or they moved to daylight savings time, and everything is shifted by one hour compared to last week, when I phoned him or her at the same time of the day. A clever database would know about all this, and if I asked for the phone number for Mr. X, it would give me the actually valid number, or at least warn me that it is not very likely to get somebody at the phone at this time of the day or the year.

One can implement of course a notion like ‘office-hour’ with a special algorithm that takes into account all the phenomena of real calendar systems (c.f. Dershowitz and Reingold’s excellent book on calendrical calculations [1]). Many commercial software products offer some kind of calendar manipulations, but they are in general limited to one calendar system and don’t convert between different systems. Much more convenient and flexible, however, would be a simple abstract specification

* This work was supported by EPSRC Research Grant GR/K57282.

language for these kind of temporal notions, which reduces the actual computation tasks to some standard algorithms. Such a specification language must be based on a mathematical model of calendar systems with all their specialties.

Let's start with the western calendar system, the Gregorian calendar. We have years, months, weeks, days, hours, minutes, seconds, milliseconds etc. Although at first glance, the system looks quite simple, there are a lot of phenomena which make a formal model difficult. First of all, there are different time zones. A year in Europe is not the same as for example a year in America. Not all years have the same length. In leap years a year is a day longer. Not all days have the same length. In many countries one day in the spring is an hour shorter and one day in the autumn is an hour longer than usual. Even not all minutes might have the same length if some 'time authorities' decide to insert some seconds into a minute to re-calibrate their reference clocks, for example to compensate for earth's decreasing angular velocity.

Some time measures are exactly in phase with each other. For example a new year starts exactly at the same time point when a new month, day, hour, minute, second starts. But it does not start at the same time point as a new week. Nevertheless, weeks are quite often counted as week 1 in a year, week 2 etc. Many businesses allocate their tasks to particular week numbers in a year. If you go into a furniture shop and buy a new sofa, they might well tell you that the sofa will be delivered in week 25, and you have to look into your calendar to find out, when the hell, is week 25.

Weeks are not in phase with months and years, but they are in phase with hours and minutes and seconds, which themselves are in phase with months and years. Quite confusing, isn't it? Things would be much easier if calendar systems would be decimal systems, with one fixed unit of time as basis, and all other units as fractions or multiples of this basis. Since this is not the case, we need to model the time units in a different way. Therefore in the next section we begin with the definition of time units as independent partitionings of a universal time axis. Based on this, a specification language for temporal notions denoting sets of time points will be presented. Its semantics is such that we obtain algorithms for deciding whether a given point in time is within the set of time points denoted by a given time term or not (for example whether it is currently an office hour, say, in Recife, Brazil).

2. Reference Time and Time Units

There is no simple way of choosing a popular unit such as a year as the basis for a mathematical model, and define everything else relative to this unit. Years have different length, and they depend on the time zone. Fortunately there is a well established time standard, which is available on most computer systems and in most programming languages. The origin of time in this standard is 1.1.1970, Greenwich Mean Time (GMT). There are functions, which give you the number of seconds (or even milliseconds) elapsed from the origin of time until right now. Other functions can map this number back to a common date format (year, month, day, hour, minute, second), either in GMT or in local time.

In our reference time standard we therefore assume a linear time axis and a time structure which is isomorphic to the set of real numbers. In a particular implementation we are free to move point 0 in this time axis to some point in history, 1.1.1970 in Unix systems, or 1.1.0 relative to the Gregorian calendar, or even at the time of the Big Bang. This is only a linear transposition by a fixed number.

Definition 2.1. (Reference Time Line) Let \mathcal{T} be an isomorphic copy of the set of real numbers. \mathcal{T} is the reference time line. The time point 0 is called the *reference origin of time*. ◁

Unfortunately it is not the case that the common time units, minutes, hours, days, weeks, months, years can be defined as multiples of a smallest unit. Nevertheless, all of them define a partitioning of the time axis into sequences of time intervals. This means, we can count for example years as ‘year 0 after the origin, year 1 after origin’ etc. The only difficulty is that different years may have different length (measured in the reference system). To specify a time unit U , for example ‘GMTyear’ as a sequence of intervals of different length, we need three things. First of all, these time units have their own coordinate system which is isomorphic to the integers. That means we can identify each time interval measured by the given unit by a particular integer. Moreover, we know, if n is the coordinate of a given interval, then $n + 1$ is the coordinate of the next interval in this sequence.

Secondly we need for each time unit U a function $U_{\mathcal{N}}$ that maps reference time points to the time unit’s own coordinates. For example if the reference time axis is the GMT time measured in seconds from 1.1.1970, then for $U = \text{GMTday}$, $U_{\mathcal{N}}$ might map all the points in the half open interval¹ $[0, 86400[$ to the GMTday coordinate 0 (if we count

¹ The notation $[b, e[$ for half open intervals denotes the set of all points t with $b \leq t < e$. We could have chosen half open intervals $]b, e]$. This does not matter.

the 1.1.1970 as day 0). In another time zone Z with, say, 1 hour difference to GMT (earlier), we would then have a unit $U' = Z\text{day}$ and a mapping $U'_{\mathcal{N}}$ which maps the interval $[-3600, 82800[$ to day 0.

Finally we need for each unit U a mapping U_{\parallel} which maps the unit's own coordinates back to the reference time axis. $U_{\parallel}(n)$ actually computes the beginning and the end of U 's time interval n . For $U = \text{GMTday}$, we would for example have $U_{\parallel}(0) = [0, 86400[$ and for U' we would have $U'_{\parallel}(0) = [-3600, 82800[$.

It is important to notice, that at this stage of our model, we assume a separate definition of U_{\parallel} and $U_{\mathcal{N}}$ for each time unit we are interested in. For the standard time units like years, months, days, hours, minutes, seconds, these functions are usually available in one form or another in a programming language, both for the GMT time zone, and for the local time zone, and, via Internet, for all other time zones. All the information about the particular calendar system, leap years, leap seconds, daylight savings time etc. must be encoded in U_{\parallel} and $U_{\mathcal{N}}$. Therefore these algorithms are usually quite complex.

The correlations between different time units so far are indirect. From a coordinate n in U 's system, we can use U_{\parallel} to go back to the reference system, and from there with $U'_{\mathcal{N}}$ to the coordinate system of some other unit U' . But this way, we can correlate all units with each other, regardless of its calendar system, its time zone, or its granularity. For example we can map the Gregorian calendar GMT-minute with coordinate n to, say the corresponding hour in the Chinese calendar system, provided the two functions $\text{GMT-minute}_{\parallel}$ and $\text{Chinese-hour}_{\mathcal{N}}$ for the two time units are available.

Definition 2.2. (Time Units) We define \mathcal{U} to be a set of *time unit symbols*.

If $U \in \mathcal{U}$ is a time unit symbol, we define an *interpretation* $\mathfrak{S}_T(U)$ to be the triple $(\mathcal{N}_U, U_{\mathcal{N}}, U_{\parallel})$ where

- \mathcal{N}_U is a U -coloured isomorphic copy of the set \mathbb{Z} of integers.²
 \mathcal{N}_U is the time unit's own coordinate system.
- $U_{\mathcal{N}} : \mathcal{T} \mapsto \mathcal{N}_U$ is a function mapping the reference time axis to the U -coordinates.
- $U_{\parallel} : \mathcal{N}_U \mapsto \text{Int}(\mathcal{T})$ is a function mapping the elements n of the U -coordinates to the half open interval $[b, e[$ in the reference time line which corresponds to the U -coordinate n .

²We can't choose closed intervals because in this case subsequent intervals are not disjoint.

²A formal definition of \mathcal{N}_U would be $\mathcal{N}_U = \{(i, U) \mid i \in \mathbb{Z}\}$ with the usual structure of integer numbers imposed on \mathcal{N}_U .

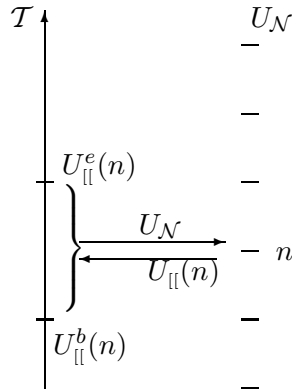


Figure 1. Relation between the reference time system and a time unit U

If $U_{||}(n) = [b, e[$ we denote with $U_{||}^b(n) = b$ the beginning of the interval and with $U_{||}^e(n) = e$ the end of the interval.

We require:

- $U_{||}^e(n) = U_{||}^b(n + 1)$ for all $n \in \mathcal{N}_U$
(there are no gaps between U -coordinates);
- $\forall t \in \mathcal{T} : t \in U_{||}(U_{\mathcal{N}}(t))$ and $\forall n \in \mathcal{N}_U : \forall t \in U_{||}(n) : U_{\mathcal{N}}(t) = n$
($U_{\mathcal{N}}$ and $U_{||}$ are inverse to each other). \triangleleft

It is important to notice that sets of half open intervals which are open all at the same side, in our case the upper side, are closed under union and intersection.

3. Temporal Notions

With the functions $U_{\mathcal{N}}$ and $U_{||}$ we can convert back and forth between a reference time system and an arbitrary time unit U . Unfortunately, time units are not the only irregular structures in a calendar where a fully fledged programming language is necessary to compute the details. Some of the Christian holidays also have a quite complex temporal cycle. The date for Easter, for example, is defined to be the first weekend after the first full moon in spring. Since in our context it makes no sense to define a language powerful enough to compute the moon cycle (this should be done in C or any other suitable programming language), we assume that certain functions, for example *easter-time* are available. *easter-time*, for example, would be a function that takes a year-coordinate and gives back a day-coordinate (Easter Sunday).

Given $U_{\mathcal{N}}$ and U_{\parallel} and special functions like *easter-time*, we need only a few further constructs to specify quite complex temporal notions and to do the corresponding computations.

3.1. THE U_within_V CONSTRUCTOR

Let us start with something very familiar, the notion of the month ‘February’. ‘February’ denotes a set of time points. There are, however, different possibilities. If I say ‘February’, do I mean a particular February, or maybe even the set of all Februaries in the history of mankind, or more sophisticated: a function, that gives me for a particular year the particular February in this year? The last interpretation is the most general one, because with this we can reconstruct the other two interpretations. Moreover, it is a quite natural one, because if you ask somebody to define ‘February’, he would say something like ‘February is the second month in a year’. What he has in mind is therefore a notion of ‘ i^{th} month in a year’, which is a function taking a year y and an integer i and yielding a month. If $i = 2$, we get a February, if $i = 3$, we get a March etc.

Both ‘month’ and ‘year’ are time units. Therefore we can generalize the notion ‘ i^{th} month in a year’ from the two time units ‘month’ and ‘year’ to two arbitrary time units U and V . For each pair of time units U and V we introduce a function U_within_V with two arguments, a V -coordinate and an integer. The value of $U_within_V(v, i)$ would be a U -coordinate.

Let’s go back to the ‘February’ example. We have $U = \text{month}$ and $V = \text{year}$. Stating

$$February(y) \stackrel{\text{def}}{=} \text{month_within_year}(y, 2) \quad (1)$$

we get a definition of a function *February* that takes a year-coordinate y and yields a month-coordinate (as the second month in a year), provided we have a suitable definition of the function `month_within_year`. In this case, we can use the *February*-function to determine whether a given point t in the temporal reference system is in a February. We compute $y = \text{year}_{\mathcal{N}}(t)$ and get the year-coordinate. $m = February(y)$ gives us the corresponding month-coordinate of the February in the year y . Now we check whether t lies in the interval `month_{\parallel}(m)`. If this is so, then t is in a February, if not, it is not in a February. Since all the regular and irregular things about the time units ‘month’ and ‘year’ are built into the functions $\text{year}_{\mathcal{N}}$ and `month_{\parallel}`, and, by the definition of U_within_V below, also into this function, *February*(y) gives us a precise definition of the notion of ‘February’.

Before we come to a general definition of the U_within_V function, let's see whether there are other useful examples of the application of this function. The date 'February, 20th' denotes a particular day in a year. Following the same ideas as above, we would like to encode this as a function that maps the coordinate of a year to the coordinate of a day, the 20th of February. We have got the function *February* which maps year-coordinates to month-coordinates. In a very similar fashion, 20th-of denotes a function which maps a month-coordinate m to a day-coordinate. Therefore we can define something like:

$$20^{th}\text{-in-month}(m) \stackrel{\text{def}}{=} \text{day_within_month}(m, 20).$$

Composing this with the *February*-function, we get:

$$20^{th}\text{-in-February}(y) = \text{day_within_month}(\text{February}(y), 20)$$

which maps a year-coordinate y to a day-coordinate. The usual date notation year:month:day:hour:minute:second can in fact be encoded by using the U_within_V function for the corresponding time units, and function composition.

As another example,

$$\text{Monday}(w) \stackrel{\text{def}}{=} \text{day_within_week}(w, 1)$$

defines Monday to be the first day in a week, by mapping a week-coordinate w to a day-coordinate. Although we have not yet given a formal definition of the U_within_V -function, its meaning seems to be quite obvious. In all cases, where the two units U and V are in phase, i.e. the beginning of a V unit (i.e. year) coincides with the beginning of a U unit (i.e. month), it is in fact obvious.

But sometimes there is a subtlety, we have to consider. For example the beginning of a year does not coincide with the beginning of a week (years and weeks are not in phase). It is therefore not clear whether *week_within_year* should count the first week overlapping with a new year as week number 1 (inclusive interpretation), or only the first week which lies completely within the new year (exclusive interpretation). In this paper we avoid making a choice on this, by simply providing both versions: $U_within^e V$ is the exclusive reading and $U_within^i V$ is the inclusive reading. For all those pairs of time units which are in phase with each other, both versions coincide.

Let us consider the definition of U_within_V for these cases first. *month_within_year*(y, i) is an example. In order to get the month coordinate for the year y and integer i , we first need to compute with $k \stackrel{\text{def}}{=} \text{year}_{\parallel}^b(y)$ the reference time of the beginning of the year. Since years and months are in phase, *month* _{\mathcal{N}} (k) yields the month-coordinate of

the corresponding January in the year y . If this is counted as the first month, then we need to add $i - 1$ to get the month-coordinate of the i -th month in the year y . The definition of $U_within_V(n, i)$ in the general case is therefore just

$$U_within_V(v, i) \stackrel{\text{def}}{=} U_{\mathcal{N}}(V_{\parallel}^b(v)) + i - 1.$$

If the two time units are not in phase, we either add $i - 1$ or i , in the exclusive reading, depending whether there is an overlap at the beginning of the interval or not. These definitions compute answers even for terms like $month_within_year(1980, 100)$ (the 100th month in the year 1980) or $year_within_month(200000, 100)$ (the 100th year in the month 200000.) It depends on the application, whether this might be useful or not.

3.2. THE *begin_U* AND *end_U* CONSTRUCTORS

Suppose we have a built-in function $springbegin(y)$ which maps a year-coordinate to a day-coordinate (giving the first day in spring). If we want to define ‘the Monday in the first week in spring’, we cannot use the ‘Monday’-function above, because this needs as an input a week coordinate. But given a day-coordinate d , we can of course get the week coordinate w of the week containing the given day. This is just $week_{\mathcal{N}}(day_{\parallel}^b(d))$, i.e. from d we compute the reference time of the beginning of the day d , and from there the week-coordinate corresponding to this reference time.

In this example, we went from the finer day-coordinate to the coarser week-coordinate. We might also want to go the other way round, say from a coarser year-coordinate to the finer day-coordinate, either of the beginning of the year (which could be achieved with the U_within_V constructor), or to the end of the year, which could only be achieved with the U_within_V constructor if the number of days within a year were fixed.

To do this kind of type casting, we introduce the $begin_U$ and end_U constructors. For a given time unit U and a given V -coordinate v of another time unit V , $begin_U(v)$ yields the U -coordinate of the reference time corresponding to the *beginning* of the v -interval, and $end_U(v)$ yields the U -coordinate of the reference time corresponding to the *end* of the v -interval. Both yield the same result if V is coarser than U . A useful example for the application of the end_U constructor is

$$new-years-eve(y) \stackrel{\text{def}}{=} end_day(y).$$

This function maps a year coordinate y to the day-coordinate of its last day, new year’s eve.

3.3. SET CONSTRUCTORS

With the constructors, we have got so far, we can select single coordinate units within bigger time units. This is still too limited for many purposes. For example, a weekend usually consists of two days, which can only be specified by joining the two days into a set. Therefore we introduce the constructors for manipulating sets of coordinates.

First of all, we introduce the familiar set constructors $\{\dots\}, \cup, \cap, \setminus$. $\{n\}$ turns the single element n into a singleton set $\{n\}$. \cup, \cap, \setminus denote union, intersection and difference of sets. Now we can define for example

$$weekend(w) \stackrel{\text{def}}{=} \{Saturday(w)\} \cup \{Sunday(w)\}$$

as a function mapping week-coordinates to the union of the two day coordinates corresponding to the Saturday and Sunday in a week. In order to check whether a given reference time t is actually in a weekend, we compute $\{d_1, d_2\} = weekend(week_{\mathcal{N}}(t))$ and check whether t is within the interval $week_{\llbracket}(d_1)$ or $week_{\llbracket}(d_2)$.

Some precautions must be taken with the binary set constructors in order not to mix coordinates of different time units. Although something like $weekend(w) \cup hour_within_week(w, 36)$ which computes the union of days and hours makes sense set theoretically, it does not make sense in our setting. Therefore in the actual language we need a type mechanism to prevent the mixing of coordinates of different time units.

3.4. THE DECOMPOSITION CONSTRUCTOR $U_{\mathcal{S}}$

Besides the singleton set constructor $\{\dots\}$, there is another constructor, which is quite useful for turning individual coordinates into sets. This is the $U_{\mathcal{S}}$ constructor. If, for example, w is a week-coordinate, then $day_{\mathcal{S}}(w)$ denotes the set of all day-coordinates within the week w . Now, for example, we can define

$$working_days(w) \stackrel{\text{def}}{=} day_{\mathcal{S}}(w) \setminus weekend(w)$$

as a function mapping week coordinates w to the set of day coordinates corresponding to the complement of the weekend coordinates.

For time units which are not in phase with each other, for example weeks and years, we need a refined $U_{\mathcal{S}}$ constructor. For computing all the weeks within a year, we have to decide whether to include the weeks which overlap partially with the year (inclusive reading) or only the weeks which are completely included in the year (exclusive reading). Therefore here again we introduce two versions of the $U_{\mathcal{S}}$ constructor, the $U^e_{\mathcal{S}}$ (exclusive) and the $U^i_{\mathcal{S}}$ (inclusive) version. The $U_{\mathcal{S}}$ constructor is particularly useful for converting between different time zones or

calendar systems. If we have for example a time unit A-day (day in time zone A) and B-hour (hour in time zone B) and a A-day coordinate d then $\text{B-hour}_s^e(d)$ yields the set of B-hours contained completely within the A-day d , and $\text{B-hour}_s^i(d)$ yields the set of B-hours overlapping with the A-day d .

3.5. INDIVIDUALS AND SETS

Each function ϕ producing a set of U -coordinates for a time unit U can become in a canonical way an argument of a function $\psi(n)$ accepting a single U -coordinate. We just define $\psi(\phi) \stackrel{\text{def}}{=} \{\psi(n) \mid n \in \phi\}$ For example

$$\text{all-Mondays}(y) \stackrel{\text{def}}{=} \text{Monday}(\text{weeks}(y))$$

denotes the set of all Mondays in the year y . $\text{weeks}(y)$ yields the set of all week coordinates for the year y , and the definition of *Monday* is applied to all of them separately. The resulting day coordinates are collected in a set.

We finish this section with a more complicated example. We want a realistic definition of the notion of ‘office-hour’. There are different ways to define this. We might start with the definition of lunch-time per day:

$$\text{lunch-time}(d) \stackrel{\text{def}}{=} \text{hour_within_day}(d, 12)$$

(Lunch time is between 12 and 1 o’clock.) Next we define $\text{ohd}(d)$ (office-hours per day):

$$\text{ohd}(d) \stackrel{\text{def}}{=} \text{hour_within_day}(d, [9..17]) \setminus \text{lunch-time}(d)$$

$(\text{hour_within_day}(d, [9..17]))$. abbreviates $\{\text{hour_within_day}(d, 9)\} \cup \dots \cup \{\text{hour_within_day}(d, 17)\}$. In the next step we define office-hours per week $\text{ohw}(w)$ and take the weekends out.

$$\text{ohw}(w) \stackrel{\text{def}}{=} \text{ohd}(\text{days}(w)) \setminus \text{hours}(\text{weekend}(w))$$

$\text{days}(w)$ gives us all day-coordinates of the week w . $\text{ohd}(\text{days}(w))$ is applied to each day-coordinate and the union of the resulting sets is formed. $\text{weekend}(w)$ yields the two day-coordinates for Saturday and Sunday. The corresponding hours are $\text{hours}(\text{weekend}(w))$, and these are subtracted from the office hours. Let’s further assume we have a definition of $\text{holidays}(y)$ which computes the set of all holidays in the year y . Now we can finally define office-hours per year

$$\text{ohy}(y) \stackrel{\text{def}}{=} \text{ohw}(\text{weeks}(y)) \setminus \text{hours}(\text{holiday}(y)).$$

In an actual implementation, one would not of course compute these coordinates as sets, but maybe represent them as sets of intervals, and do the corresponding set operations on the intervals.

3.6. PERIODS OF TIME

Time units are not only coordinates for fixed intervals in the reference time line. They are also used for specifying other periods of time. If somebody says for example ‘for the next two years ...’, he usually means ‘for a period of time beginning now and lasting for two years’. Unfortunately notions like ‘for two years’ are not as precise as they seem to be. If next year is a leap year, does ‘for two years’ mean $365 + 365$ days or $366 + 365$ days? When does this period start and end? Does it start exactly at this moment and end at some point at some day in two years time, or does it include today as a whole day and the last day after two years time as a whole day?

I propose a flexible encoding of time periods where the user can choose the precise meaning. The constructor is *U_period_V*. For example *day_period_year(d, 2)* denotes the set of day coordinates from the day *d* up to a day after 2 years time. This includes the begin and the end day. If the 2 years time period are to be located more precisely, say, at a resolution of hours, one can choose *hour_period_year(h, 2)* which yields the set of hour coordinates from hour *h* up to an hour after 2 years time. (One can of course also define an exclusive reading where the coordinates at the beginning and at the end are not included.)

The problem that a period of 2 years time itself is also not precisely defined, can be solved, or at least be approximated in the following way: Consider again *day_period_year(d, 2)*. Day *d* defines an interval $[d^b, d^e[$ in the reference time line. We compute the middle $t_d = (d^b + d^e)/2$ of this interval, and from this the current year coordinate *y*. *y* itself corresponds to an interval $[y^b, y^e[$, and $t_d \in [y^b, y^e[$. t_d therefore divides $[y^b, y^e[$ into a fraction $f = (t_d - y_b)/(y^e - y^b)$. *f* is the fraction of the year *y* elapsed since y_b . Now we go 2 years further to $y' = y + 2$ and get the time point $t'_d = y'^b + f \cdot (y'^e - y'^b)$ which corresponds to the same fraction *f*. From t'_d we get the day coordinate of the corresponding day in two years time.

3.7. ARITHMETIC

The results of all these functions, we defined so far, are either single coordinates or sets of coordinates, and a coordinate is nothing but an integer. Therefore nothing stops us from embedding the time language into an integer arithmetic language. This way, we can for example define

$$\text{tomorrow}(d) \stackrel{\text{def}}{=} d + 1$$

as a function that returns for a given day coordinate *d* the coordinate of the following day. Even conditional terms with arithmetic comparisons

are possible. Borrowing the $c?a : b$ construct from the programming language C (if c holds then a , otherwise b) we can for example define the ‘first Monday after the beginning of springtime’ by

$$\begin{aligned} & \text{Monday}(\text{begin_week}(\text{springbegin}(y))) < \text{springbegin}(y) ? \\ & \text{Monday}(\text{begin_week}(\text{springbegin}(y)) + 1) : \\ & \text{Monday}(\text{begin_week}(\text{springbegin}(y))) \end{aligned}$$

(if the beginning of springtime lies within a week such that the Monday is actually before the beginning of springtime, then we choose the Monday of the next week, otherwise we take this Monday.)

4. The Time Term Language

We now turn these informal ideas into a formal specification language. Its semantics is purely functional and can be defined in terms of a kind of interpreter which can actually execute the computations indicated above. A specification S of temporal notions is a list of equations $f(u_1, \dots, u_n) \stackrel{\text{def}}{=} \varphi[u_1, \dots, u_n]$ where on the left hand side a new function symbol f is introduced, and defined by φ . S should be such that all defined function symbols can be eliminated in a finite sequence of rewrite steps.

In order not to mix up coordinates of different time units (all of them are essentially integers), we need a sorted (typed) language where for each function the sort of the arguments and the sort of the resulting values are specified [3]. The sort structure consists of two parts. The ‘single value part’ contains the sort symbol \mathbb{Z} (for the integers) and the time unit symbols \mathcal{U} used as further sort symbols denoting the time units’ own coordinates. In the ‘set value part’ we have for each time unit sort symbol $U \in \mathcal{U}$ a symbol U^* denoting sets of U -coordinates. In addition there is a sort $Bool$ for arithmetic comparisons.

Most of the function symbols can have ‘overloaded’ sort declarations where the sort of the function value depends on the sort of its arguments.

Definition 4.1. (The Time Term Language)

We define a first-order (many-sorted) term language \mathcal{L} as follows:

- The set of (element) sort symbols consists of the (finite) set \mathcal{U} , the special symbol \mathbb{Z} for integers, and the sort $Bool$ for Boolean values. In addition we have for each $U \in \mathcal{U}$ a (set) sort symbol U^* (denoting sets of U -elements).

- For each $U \in \mathcal{U}$, each $u \in \mathcal{N}_U$ is a *constant symbol* of sort U . Each integer is a constant of sort \mathbb{Z} ³.
- For each sort $U \in \mathcal{U}$ we have an unlimited supply of variable symbols of that sort (there are no ‘set variables’, no integer variables and no Boolean variables).
- We can have an arbitrary number of built-in functions of sort $U_1 \times \dots \times U_n \rightarrow U$. (*eastertime(year)* would be a typical built-in function.)⁴
- Depending on the set \mathcal{U} of time units, there is a certain set of derived function symbols.
 - For $U, V \in \mathcal{U}$ we have function symbols $U_within^e V$ and $U_within^i V$, both of sort $V \times \mathbb{Z} \rightarrow U$ and $V^* \times \mathbb{Z} \rightarrow U^*$ (a single V -coordinate as input yields a single U -coordinate as output, whereas a set of V -coordinates yields a set of U -coordinates as output.)
 - For each $U \in \mathcal{U}$ we have the ‘begin’ and ‘end’ constructors $begin_U$ and end_U , both of sort $V \rightarrow U$ and $V^* \rightarrow U^*$ for all sorts $V \in \mathcal{U}$.
 - For each $U \in \mathcal{U}$ we have the set constructor U_s of sort $V \rightarrow U^*$ and $V^* \rightarrow U^*$ for all sorts $V \in \mathcal{U}$.
 - For $U, V \in \mathcal{U}$ we have function symbols U_period_V and both of sort $U \times V^*$ and $U^* \times V^*$.
 - The ‘singleton set’ operator $\{\dots\}$ of sort $U \rightarrow U^*$ for each sort $U \in \mathcal{U}$ maps single elements to singleton sets.
 - The set connectives \cap, \cup, \setminus , all of sort $U^* \times U^* \rightarrow U^*$, for all $U \in \mathcal{U}$ denote the usual set operations intersection, union and set difference.
 - For each time unit $U \in \mathcal{U}$ there are two ‘decomposition function’ symbols U^e_s and U^i_s with sort declarations $V \rightarrow U^*$ for each sort (time unit) $V \in \mathcal{U}$.
- We include the standard integer arithmetic symbols $+, -, *$, all of sort $U \times U \rightarrow U$, $U^* \times U \rightarrow U$ and $U \times U^* \rightarrow U$ for all $U \in \mathcal{U}$.
- There is the special Boolean conditional operator $\dots? \dots : \dots$ of sort $Bool \times U \times U \rightarrow U$ for all $U \in \mathcal{U}$.

³ With these constants we have a name for each coordinate. For example the constant 1997 of sort ‘year’ (actually the pair (1997,year)) denotes the year 1997.

⁴ In the sort declarations $U_1 \times \dots \times U_n \rightarrow U$ for function symbols, the U_k are the *domain-sorts*, and the U is the *range-sort*.

- The standard integer comparison operators $=, <, >, \leq, \geq$ of sort $U \times U \rightarrow Bool$ for all $U \in \mathcal{U}$ can be used in the conditional part of the conditional operator.
- Other function symbols can only be used as abbreviations if they have suitable non-recursive definitions of the form $f(u_1, \dots, u_n) \stackrel{\text{def}}{=} \psi[u_1, \dots, u_n]$.⁵ This way one can always get rid of these function symbols by using their defining equation as rewrite rule (this is actually the main requirement on these definitions). \triangleleft

The sort declarations for the function symbols guarantee that they don't compute coordinates of mixed time units (sorts).

Proposition 4.2. (Unique Range-Sort) Since the terms of our language \mathcal{L} are standard first-order terms with overloaded sort declarations, given the sorts of the free variables in the term they always have a unique range-sort. \triangleleft

We define the semantics of \mathcal{L} terms by giving the definitions of all the built-in functions. From a logical point of view, the semantics of the language \mathcal{L} is based on a single model only.

Definition 4.3. (Semantics of the Time Term Language) Given the interpretation \mathfrak{S}_T for the time unit symbols \mathcal{U} (Def. 2.2) we define an interpretation $\mathfrak{S} = (\mathfrak{S}_T, \mathcal{S}, \mathcal{V}, \mathcal{F})$ for the time term language \mathcal{L} :

- \mathcal{S} maps (element) sort symbols U to \mathcal{N}_U and (set) sort symbols U^* to the set of all subsets of \mathcal{N}_U .
- \mathcal{V} is the variable assignment. For each variable u of sort U we have $\mathcal{V}(u) \in \mathcal{N}_U$.
- The function interpretation \mathcal{F} maps constant symbols to themselves. It also contains the interpretation of the built-in functions (such as *eastertime*).
- The other functions are interpreted by \mathcal{F} as follows:
 - if $v \in \mathcal{N}_V$ and $i \in \mathbb{Z}$ then $\mathcal{F}(U_within^i V)(v, i) = U_{\mathcal{N}}(V_{\parallel}^b(v)) + i - 1$.
 - $\mathcal{F}(U_within^e V)(v, i) = \begin{cases} U_{\mathcal{N}}(V_{\parallel}^b(v)) + i - 1 & \text{if } U_{\parallel}^b(U_{\mathcal{N}}(V_{\parallel}^b(v))) < V_{\parallel}^b(v) \\ U_{\mathcal{N}}(V_{\parallel}^b(v)) + i & \text{otherwise} \end{cases}$.

⁵ The notation $\psi[u_1, \dots, u_n]$ means that ψ is a term containing the variables u_1, \dots, u_n at some places. u_1, \dots, u_n are ψ 's free variables.

- if $v \in \mathcal{N}_V$ then
 $\mathcal{F}(U^i_{\text{S}})(v) = \{u \in \mathcal{N}_U \mid V_{\mathcal{N}}(U_{\parallel}^b(u)) = v \text{ or } V_{\mathcal{N}}(U_{\parallel}^e(u)) = v\}$
 $\mathcal{F}(U^e_{\text{S}})(v) = \{u \in \mathcal{N}_U \mid V_{\mathcal{N}}(U_{\parallel}^b(u)) = v \text{ and } V_{\mathcal{N}}(U_{\parallel}^e(u)) = v\}$
- $\mathcal{F}(\text{begin}_{\text{U}})(v) = U_{\mathcal{N}}(V_{\parallel}^b(v))$ if v is of sort $V \in \mathcal{U}$.
 $\mathcal{F}(\text{end}_{\text{U}})(v) = U_{\mathcal{N}}(V_{\parallel}^e(v))$ if v is of sort $V \in \mathcal{U}$.
- $\mathcal{F}(U_{\text{period}}V)(u, n) = \{u, \dots, w\}$ where w is determined as follows:

Let $t \stackrel{\text{def}}{=} (U_{\parallel}^e(u) + U_{\parallel}^b(u))/2$ the middle of the u interval.
Let $v \stackrel{\text{def}}{=} V_{\mathcal{N}}(t)$ the V -coordinate of the middle.
Let $f \stackrel{\text{def}}{=} \frac{t - V_{\parallel}^b(v)}{V_{\parallel}^e(v) - V_{\parallel}^b(v)}$ the fraction elapsed so far.
Let $v' \stackrel{\text{def}}{=} v + n$ the new V -coordinate.
Let $t' \stackrel{\text{def}}{=} V_{\parallel}^b(v') + f \cdot (V_{\parallel}^e(v') - V_{\parallel}^b(v'))$ the end time point.
Let $w \stackrel{\text{def}}{=} U_{\mathcal{N}}(t')$ the end U -coordinate.

- if the argument to one of the functions is a set instead of a single coordinate then the function is applied to each set and the results are collected in a set.
- the conditional $\dots? \dots : \dots$, the set operators and the arithmetic terms are interpreted in the standard way. If one argument to one of the arithmetic functions is a set, then the function is applied to each element and the results are collected in a set.

– Arbitrary time terms $\psi[u_1, \dots, u_m]$ with free variables $\{u_1, \dots, u_n\}$ with $\text{sort}(u_i) = U_i$ for $1 \leq i \leq n$, and range-sort U can now be interpreted in two ways. The first way is the standard homomorphic extension of the interpretation \mathfrak{S} . If $\psi = u$ is a variable then $\mathfrak{S}(u) \stackrel{\text{def}}{=} \mathcal{V}(u)$. If $\psi = f(t_1 \dots, t_n)$ is a complex term then $\mathfrak{S}(\psi) \stackrel{\text{def}}{=} \mathcal{F}(f)(\mathfrak{S}(t_1), \dots, \mathfrak{S}(t_n))$. The binding of the variables u_i to some U_i -coordinates is contained in the variable assignment \mathcal{V} . Therefore $\mathfrak{S}(\psi)$ is well defined and yields a set of U -coordinates.

– Much more interesting and useful is the interpretation of time terms as functions mapping a reference time point t to some U -coordinates. If for a variable v of sort V , and a V -coordinate n we define $\mathfrak{S}[v/n]$ to be the interpretation which is like \mathfrak{S} , but the variable assignment maps v to n , then we can define an $\mathfrak{S}(\psi)(t)$ as a function mapping a reference time point t to some U -coordinates.

If $\psi = u$ is a variable, then

$$\mathfrak{S}(u)(t) \stackrel{\text{def}}{=} \mathfrak{S}[u/U_{\mathcal{N}}(t)](u).$$

If $\psi[u_1, \dots, u_m]$ is a complex term, then

$$\mathfrak{S}(\psi)(t) \stackrel{\text{def}}{=} \mathfrak{S}[u_1/U_{1\mathcal{N}}(t), \dots, u_m/U_{m\mathcal{N}}(t)](\psi).$$

Furthermore let

$$\mathfrak{S}'(\psi)(t) \stackrel{\text{def}}{=} \bigcup_{u \in \mathfrak{S}(\psi(t))} U_{\parallel}(u).$$

◁

$\mathfrak{S}(\psi[u_1, \dots, u_m])(t)$ computes the value of $\psi[u_1, \dots, u_m]$ at a given reference time point t by first computing the corresponding U_i coordinates $U_{i\mathcal{N}}(t)$, binding the variables u_i to these coordinates and then evaluating them in the usual way. The result is either a single U -coordinate or a set of U -coordinates.

The \mathfrak{S}' function goes one step further and turns the computed U -coordinates back into intervals in the reference time system.

Using the interpreter \mathfrak{S} we can now check the *instance* relation between a given reference time point and a \mathcal{L} term (for example whether the current moment in time lies in the office hours of my colleague in Brazil).

Definition 4.4. (Instance Relation) A time point t in the reference system is an instance of the term ψ with range-sort U (we write $t \in \psi$), iff $t \in \mathfrak{S}'(\psi[u_1, \dots, u_m])(t)$ or, which is equivalent, iff $U_{\mathcal{N}}(t) \in \mathfrak{S}(\psi[u_1, \dots, u_m])(t)$. ◁

According to this definition, we compute the U -coordinate corresponding to t and check whether this is one of the coordinates computed from ψ . In general this will still be quite inefficient, and better algorithms have to be worked out.

A more complicated relation is the subsumption relation. An example might be the problem to figure out whether my colleague's morning office hours in Brazil lie always within my local afternoon. (Then I could always phone him in the afternoon without first checking it in the database.) Unfortunately this general subsumption relation is usually not decidable. This is due to the fact that we treat the functions $U_{\mathcal{N}}$ and U_{\parallel} as a kind of black boxes where the most peculiar things may be encoded, for example a transition to a completely new calendar system in the year 2100.

For temporal relationships from everyday life, this general subsumption relationship, however, is really too general. I certainly won't be interested to know whether even in the year 2100 my colleague's morning office hours in Brazil lie within my local afternoon. For me it is completely sufficient to know this until his or my own time of retirement.

Definition 4.5. (Subsumption Relation) A term ψ is subsumed by a term φ ($\psi \subseteq \varphi$) iff for all time points t in the reference system $\mathfrak{S}'(\psi)(t) \subseteq \mathfrak{S}'(\varphi)(t)$.

A *restricted subsumption* relation ($\psi \subseteq_i \varphi$) for the time interval $i = [t_1, t_2[$ holds between ψ and φ iff for all time points $t_1 \leq t < t_2$ $\mathfrak{S}'(\psi)(t) \subseteq \mathfrak{S}'(\varphi)(t)$. \triangleleft

Since the restricted subsumption relation still quantifies over an infinite number of time points, this definition is not yet a suitable basis for an algorithm. In order to decide restricted subsumption, we exploit that the time terms depend on the coordinates of the time units, and they are therefore constant over the period in the reference time line corresponding to the same coordinates of the time unit. For example the time term *February*(y) (Def. (1)) depends on the year coordinate. It is therefore constant over a whole year, i.e. over all the points t in the reference time line with $\text{year}_{\mathcal{N}}(t) = y$.

Exploiting this observation, each time term $\psi[u_1, \dots, u_n]$ can be used to partition each finite interval in the reference time line into the finitely many sub-intervals where $\psi[u_1, \dots, u_n]$ represents a constant function. This partitioning can then be used to trigger finite case analysis for checking whether ψ subsumes φ over an interval i . i is partitioned into the set $\{i_1, \dots, i_n\}$ of sub-intervals over which $\psi \cup \varphi$ is constant, and then for $1 \leq k \leq n$, $\mathfrak{S}(\psi)(t_k) \subseteq \mathfrak{S}(\varphi)(t_k)$ is checked for some arbitrary chosen $t_k \in i_k$

Definition 4.6. (Partitioning Algorithm) For a non-empty interval $i = [i^b, i^e[\subseteq \mathcal{T}$ and a time term $\psi[u_1, \dots, u_n]$ with $U_k \stackrel{\text{def}}{=} \text{sort}(u_k)$, $k = 1, \dots, n$, we define the *partitioning of i with respect to ψ* as

$$\delta(i, \psi) = \{[i^b, t]\} \cup \delta([t, i^e[, \psi)$$

where

$$t \stackrel{\text{def}}{=} \begin{cases} \min(\{U_{k|l}(U_{k\mathcal{N}}(i^b) + 1) \in i \mid 1 \leq k \leq n\}) \\ i^e \text{ if this set is empty.} \end{cases}$$

\triangleleft

The partitioning algorithm decomposes i by computing the U_k -coordinate $U_{k\mathcal{N}}(i^b)$ corresponding to the beginning of the interval i and then checking whether the beginning of the next interval corresponding to the next U_k -coordinate still lies in i or not. If it still lies in i , this is a candidate for the lower border of the next sub-interval. The real border is determined by the time unit U_k which gives the smallest such border. (A properly implemented algorithm would of course try

the most fine grained time unit U_k first, e.g. seconds before minutes before hours etc.)

Notice that the partitioning function only depends on the time units corresponding to ψ 's free variables, not on ψ itself.

Proposition 4.7. (Soundness of the Partitioning Algorithm)

For each interval $j \in \delta(i, \psi)$ (Def. 4.6) and for each $\{t_1, t_2\} \subseteq j$: $\mathfrak{S}'(\psi)(t_1) = \mathfrak{S}'(\psi)(t_2)$.

Proof: by induction on the number of intervals in $\delta(i, \psi)$. In the base case, $\delta(i, \psi) = i$, i.e. i itself is the only component, and for $1 \leq k \leq n$, $U_{k\llbracket}(U_{k\mathcal{N}}(i^b) + 1) > i^e$. Therefore for all $t \in i$, $U_{k\mathcal{N}}(t)$ yields the same U_k coordinate. Since $\psi[u_1, \dots, u_n]$ depends on the U_k -coordinates only, $\mathfrak{S}'(\psi)(t_1) = \mathfrak{S}'(\psi)(t_2)$ for all $\{t_1, t_2\} \subseteq i$.

In the induction step, $\delta(i, \psi)$ consists of more than one interval and $[i^b, s[$ is the left most sub-interval of the decomposition. According to the definition of δ , t is the smallest value such that $t = U_{k\llbracket}(U_{k\mathcal{N}}(i^b) + 1)$ lies still in i . That means for all $k \in \{1, \dots, n\}$: $U_{k\mathcal{N}}(t_1) = U_{k\mathcal{N}}(t_2)$ for all $t_1 \in [i^b, t[$ and $t_2 \in [i^b, t[$. Thus, $\mathfrak{S}'(\psi)(t_1) = \mathfrak{S}'(\psi)(t_2)$ for the first interval in $\delta(i, \psi)$. The induction hypothesis directly applies to the remaining intervals. \triangleleft

Proposition 4.8. (Decidability)

- The instance relation is decidable. $\mathfrak{S}'(\psi)(t)$ yields only finitely many intervals. Therefore just check $t \in \mathfrak{S}'(\psi)(t)$, which amounts to finding some $[i^b, i^e[\in \mathfrak{S}'(\psi)(t)$ with $i^b \leq t < i^i$.
- Without special assumptions about the functions $U_{\mathcal{N}}$ and U_{\llbracket} , the subsumption relation is in general not decidable.
- The restricted subsumption relation $\psi \subseteq_i \varphi$ over an interval $i = [t_1, t_2[$ is decidable. Using the partitioning algorithm of Def. 4.6, partition this interval into the finitely many sub-intervals where the two time terms are constant functions, choose for each sub-interval j some point $t \in j$, and check the subsumption relation $\mathfrak{S}'(\psi)(t) \subseteq \mathfrak{S}'(\varphi)(t)$ for this point. To do this, check for each $[i^b, i^e[\in \mathfrak{S}'(\psi)(t)$ whether there is some $[j^b, j^e[\in \mathfrak{S}'(\varphi)(t)$ with $[i^b, i^e[\subseteq [j^b, j^e[$.⁶ \triangleleft

4.1. TOP-DOWN INTERPRETATION

The interpretation $\mathfrak{S}(\psi)$ of a time term ψ works bottom-up in the usual recursive style. There is a recursive descent into the term down to the

⁶ $\mathfrak{S}'(\psi)(t)$ should be computed in such a way that the sequences of intervals without gaps are comprised into one single interval.

variable level, and then the actual computation is done on the way back to the top-level of the term. This can be quite expensive if large sets of coordinates are generated. For example the interpretation of the term $\text{seconds}(y)$ where y is a year-variable causes the generation of the set of all second-coordinates within the year y (a set with about 31 million elements).

For the instance check $t \in \text{seconds}(y)$ (Def. 4.4) this would be much too expensive. A *top-down interpretation* is much more efficient in this case. How could this work? For checking $t \in \text{seconds}(y)$, we exploit that the term $\text{seconds}(y)$ has range-sort ‘second’. Therefore one can compute the second-coordinate $s = \text{second}_{\mathcal{N}}(t)$, and then compute the set of all year coordinates y such that $s = \text{seconds}(y)$. There is only one y in this case, and this can be computed quite easily, using the semantics of the seconds -constructor. If $t \in \text{year}_{\llbracket}(y)$ the instance relation holds, otherwise not.

To do this kind of top-down interpretation, one needs for each constructor $f(u_1, \dots, u_n)$ in the language an algorithm for computing for a given u the set of all u_1, \dots, u_n -tuples such that $u = \mathcal{F}(f)(u_1, \dots, u_n)$. That means one must compute for a wanted result of a computation the set of all arguments which actually produce this result. A top-down interpreter can then descend into a time term and pass the potential arguments as wanted results to the functions at the deeper level of the term. This has not yet been investigated in detail, but for the constructors in the time term language this seems to be quite feasible.

There are, however, examples, where this strategy also causes the computation of large sets of coordinates. Therefore a mixed bottom-up, top-down strategy seems to be the most efficient way to decide the instance relation.

5. Summary

In this paper I proposed a formal representation of real time and real calendar systems. With corresponding conversion functions the calendar system’s time units, years, months, weeks, days, hours, minutes and seconds are individually linked to a reference time system. Each time unit has its own coordinate axis (as integers) which represents an interval in the reference time system. This way one can convert between arbitrary calendar systems and arbitrary time zones.

In the second part a specification language \mathcal{L} for temporal notions has been proposed. It allows us to specify complex temporal notions in a very simple and intuitive way. Since the reference time system represents a single model for this language, and we have the conversion

functions, the terms of the language \mathcal{L} can be interpreted and questions like ‘is a given point in time within the time intervals specified by a term ψ ?’ can be decided.

Possible applications of the presented system are temporal databases with automatic conversion between different time systems and user defined notions. Another potential application is the World-Wide-Web, where the author of a WWW-page can use his own temporal notions, and the WWW-browser can convert it into the reader’s favourite system.

In [2] we incorporated the time term language into *Calendar Logic*, a propositional temporal logic with operators ‘sometimes within ψ ’ and ‘always within ψ ’ where ψ is a time term. A simple statement which can be expressed in this language is ‘yesterday I worked for eight hours with one hour lunch break at noon’. Calendar Logic is decidable, and we provided a tableau based decision procedure.

References

1. Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations*. Cambridge University Press, 1997.
2. Hans Jürgen Ohlbach and Dov Gabbay. *Calendar logic*. Imperial College, 1997.
3. Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, New York, 1989.