A Reasoner for Calendric and Temporal Data – Extended Version

François Bry, Frank-André Rieß, and Stephanie Spranger

Institute for Informatics, Univ. Munich, Germany
http://www.pms.ifi.lmu.de
contact: spranger@pms.ifi.lmu.de

Abstract. Calendric and temporal data are omnipresent in countless Web and Semantic Web applications and Web services. Calendric and temporal data are probably more than any other data a subject to interpretation, in almost any case depending on some cultural, legal, professional, and/or locational context. On the current Web, calendric and temporal data can hardly be interpreted by computers. This article contributes to the Semantic Web, an endeavor aiming at enhancing the current Web with well-defined meaning and to enable computers to meaningfully process data. The contribution is a reasoner for calendric and temporal data. This reasoner is part of CaTTS, a type language for calendar definitions. The reasoner is based on a "theory reasoning" approach using constraint solving techniques. This reasoner complements general purpose "axiomatic reasoning" approaches for the Semantic Web as widely used with ontology languages like OWL or RDF.

1 Introduction

Calendric and temporal data are omnipresent in countless Web and Semantic Web applications and Web services, e.g. to schedule appointments, to book flights and hotels, to plan journeys, and to organize web-based commerce. Most existing or foreseen mobile computing applications refer to not only locations but also time. E.g. a mobile application listing pharmacies in the surrounding of a user will preferably only mention those pharmacies that are currently open. The calendric and temporal data involved in such applications are probably more than any other data a subject to interpretation, in almost any case depending on some cultural, legal, professional, and/or locational context [1]. On the current Web, such data can hardly be interpreted by computers.

The vision of the Semantic Web is to enrich the current Web with well-defined meaning and to enable computers to meaningfully process such data. This article contributes to the Semantic Web vision with a reasoner for calendric and temporal data. This reasoner is part of CaTTS, a type language for calendar definitions. CaTTS provides with language constructs to conveniently model calendars like the Gregorian and Hebrew calendars or some professional calendar (e.g. of a university) and calendric types like "month", "week", or "teaching term" as well as constraints on calendric data referring to such types, e.g. "exams

are within the last week of a teaching term". CaTTS is presented in [2, 3], two articles that focus CaTTS' data and constraint modeling aspects.

The subject of this article is CaTTS' reasoner. This reasoner is intended to answer queries over a wide range of temporal and calendric constraints over calendric and temporal data like "what are the possibilities to schedule some student's lectures and courses in a teaching term". Such queries can be formulated in CaTTS' constraint language CaTTS-CL. CaTTS' reasoner is based on a "theory reasoning" approach [4, 2, 5] using constraint programming techniques for problems expressed in CaTTS-CL. This reasoner complements general purpose "axiomatic reasoning" approaches for the Semantic Web, widely used with ontology languages like RDF [6] or OWL [7]. CaTTS' reasoner refers to and relies on user-defined calendric types of calendars specified in the definition language CaTTS-DL. This makes search space restrictions possible that would not be possible if calendars and temporal notions would be specified in a generic formalism such as first-order logic and processed with generic reasoning methods such as first-order logic theorem provers.

2 CaTTS in a Nutshell

CaTTS consists of two languages, CaTTS-DL, a type definition language and CaTTS-CL, a constraint language.

2.1 The Definition Language CaTTS-DL

CaTTS-DL, CaTTS' *definition language*, consists of two language parts, CaTTS-TDL (for type definition language) and CaTTS-FDL (for format definition language). CaTTS-DL provides means to declaratively define calendric types and calendars, themselves "typed" after calendar signatures as well as date formats for calendric types to user-friendly rendering and parsing calendric data.

In CaTTS-TDL, any calendric type is defined by a (user-defined) *predicate,* i.e. the elements belonging to a calendric type must fulfill the conditions defined by such a predicate. E.g. a predicate defining a calendric type "working day" may restrict days to those which are between each week's Monday and Friday. Each such type is (directly or indirectly) related to any other type defined in a CaTTS-TDL calendar specification. E.g. the Gregorian calendar can be modeled in CaTTS-TDL as follows:

```
calendar Gregorian:GREGORIAN =
 cal
   type second;
   type minute = aggregate 60 second @ second(1);
   ...
   type month = aggregate
     31 day named january,
     alternate month(i)
      | (i div 12) mod 4 == 0 &&
        ((i div 12) mod 100 != 0 || (i div 12) mod 400 == 0) -> 29 day
      | otherwise                                             -> 28 day
     end named february, ..., 31 day named december @ day(1);
   type year = aggregate 12 month @ month(1);
```

```
  type working_day = select day(i) where
          relative i in week >= 1 && relative i in week <= 5;
  type weekend_day = day\working_day;
end
```

The above CaTTS-DL calendar specification[1] consists of a set of type definitions (each identified by the keyword `type` followed by an identifier). The first type defined is `second`. It has no further properties, i.e. it is a user-defined parameterless type constructor. The type `minute` is defined from the type `second` by specifying a predicate stating that a minute consists of 60 seconds[2] (denoted `aggregate 60 second`) and such that the minute that has index 1, i.e. `minute(1)` comprises all seconds between `second(1)` in `second(60)` (denoted `@ second(1)`). In CaTTS-DL, the type `minute` is an *aggregation subtype* of the type `second` (written `minute` $\preceq$ `second`), because each week can be defined as an interval of days. Any further type definition follows the same pattern. The definitions are straightforward following the rules of the Gregorian calendar [8]. The type `month` is defined by a repeating pattern of the twelve Gregorian months. February which is one day longer in each Gregorian leap year is defined by an additional pattern which specifies the leap year rule for the Gregorian calendar using the CaTTS language construct `alternate...end`. The type definition of the type `working_day` is derived from that of the type `day` by specifying a predicate stating that a working day is one of the first five days (i.e. Monday to Friday) in a week (denoted `relative i in week >= 1 && relative i in week <= 5`). In CaTTS-DL, the type `working_day` is an *inclusion subtype* (in the common set-theoretic sense) of the type `day` (written `working_day` $\subseteq$ `day`), because the set of working days is a subset of the set of days. The type `weekend_day` is also an inclusion subtype of type `day`, selecting those days which are not working days (denoted `day\working_day`, where "\" is a CaTTS-DL type constructor for exception types) The syntax of CaTTS is given in Appendix A.

The above exemplified CaTTS-DL calendar specification defines a calendar as a "type" that, in principle, can be used with *any* Web language (e.g. XQuery or XSLT), using calendar data enriched with type annotations after this CaTTS-DL calendar. CaTTS' type checker [9] is used to check the calendar data typed after a CaTTS-DL calendar in such programs or specifications, thus, providing a means to annotate and interpret such data. Particularities like time zones can be easily expressed in a CaTTS-DL as shown in [2]. Further CaTTS-DL calendar specifications, in particular the Islamic and Hebrew calendars and variations of the Gregorian calendar like the Japanese calendar are given in [2, 1].

With most applications, one would appreciate not to specify dates and times using indices of the elements of CaTTS types like `day(23)` or `second(-123)`, but instead *date formats* like "5.10.2004" (common in Germany for 5th October 2004), "10/05/2004" (common in the US for 5th October 2004), or "Tue Oct 5 16:39:36 CEST 2004". CaTTS-FDL provides a means for defining date formats.

[1] This calendar specification is bound to the identifier `Gregorian` and must match the calendar signature `GREGORIAN`.

[2] In CaTTS-DL, it is also possible to define a type `minute` including leap seconds as shown in [2].

Similar to calendric types, date formats are specified by a (user-defined) predicate. E.g. the possible values for day, month, and year numbers in a Gregorian date (according to the internal indices of the elements of type `day` defined in the CaTTS-DL calendar specification given above) can be specified as follows in CaTTS-FDL:

```
format date:day = d "." m "." y where
     date within year(y),
     date within M is 1 month,
     m == relative index M    in year,
     d == relative index date in month;
end
```

This CaTTS-FDL specification binds the identifier `date` of type `day` to a date format of a (relative) day value, followed by a dot, followed by a (relative) month value, followed by a dot, followed by a (absolute) year value, e.g. `"11.2.2005"`. The variable `date` must satisfy the following predicates for `d`, `m`, and `y`. The predicate `date within year(y)` restricts date to be within the year indexed by `y` (e.g. `"11.2.2005" within 2005`). Similarly, the predicate `date within M is 1 month` restricts date to be within a month `M`. The (relative) month value `M` must be the m^{th} in a year (denoted `m == relative index M in year`), and the (relative) day value `date` must be the d^{th} day in a month.

2.2 The Constraint Language CaTTS-CL

CaTTS-CL, CaTTS' *constraint language,* is typed after CaTTS-DL type definitions. CaTTS-CL is a language for declaratively expressing a wide range of temporal and calendric problems over domains of different calendric type defined in CaTTS-DL. Such problems are then solved by CaTTS-CL's constraint solver. Given a CaTTS-DL specification of the Gregorian calendar (with types "day", "working day", and "month") and CaTTS-FDL format specifications for types "day" and "month", one can specify in CaTTS-CL a problem like planning a meeting of three consecutive working days after 22nd April 2005 that is finished before May 2005. This problem can be expressed in CaTTS-CL as follows:

```
Meeting is 3 working_day &&
Meeting after "22.04.2005" &&  Meeting before "05.2005"
```

The variable `Meeting` represents the domain of three-day long working day intervals (denoted `Meeting is 3 working_day`). The constraint `Meeting after "22.04.2005"` restricts the domain of the variable `Meeting` to those three-day long working day intervals starting after the day `"22.04.2005"`. Finally, the constraint `Meeting before "05.2005"` restricts the domain of `Meeting` to those intervals ending before the month `"05.2005"`.

An *answer* to a problem specified in CaTTS-CL is itself a CaTTS-CL constraint, that can no longer be simplified. Such an answer corresponds to the result computed by CaTTS' constraint propagation algorithm (cf. Section 3). The answer to the above problem in CaTTS-CL is given by the following:

```
Meeting after "22.04.2005" &&  Meeting before "05.2005" &&
Meeting is 3 working_day && start Meeting is "25.04.2005"..."27.04.2005"
```

CaTTS' constraint solver adds a new constraint on the domain of the variable `Meeting`: `start Meeting is "25.04.2005".."27.04.2005"`. This constraint specifies that the possible starting point for the three-working-day long meeting must be one working day out of the finite domain (represented by an interval) between working day `"25.04.2005"` and `"27.04.2005"`. This constraint is inferred by propagating the constraints `Meeting after "22.04.2005"` and `Meeting before "05.2005"`. Since `Meeting` must be a working day, it can not start before working day `"25.04.2005"` (according to the constraint `Meeting after "22.04.2005"`, where `"22.04.2005"` is a day). According to the constraint `Meeting before "05.2005"`, `Meeting` must end not later than working day `"29.04.2005"`, i.e. the latest working day that is before `"05.2005"`. One *solution* to the above given problem (for the 3 working day long meeting) is `"27.04.2005"` to `"29.04.2005"`. Note that answers and solutions are closely related: an answer is a compact, constraint-based, representation of several solutions (e.g. the above given answer contains 3 possible solutions). In some cases, an answer might contain unsatisfiable parts. As a consequence, solutions are necessary. Solutions are computed by searching (using back tracking) the answer. Note further that CaTTS-CL provides the user with the possibility to ask the system to compute one (or more) possible solutions from an answer.

A CaTTS-CL *program* is a finite conjunction (expressed by the keyword `&&`) of CaTTS-CL constraints. CaTTS' complete syntax, including the syntax of CaTTS-CL is given in Appendix A.

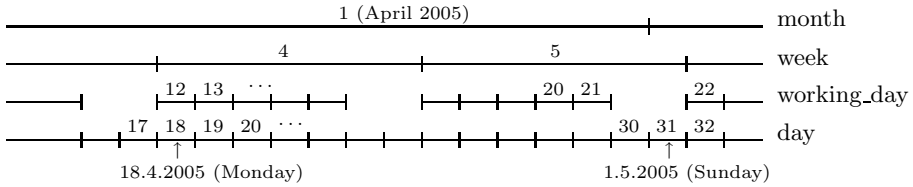

Fig. 1. Illustration of the calendric types used in Example 1 with internal indexing.

3 CaTTS' Constraint Solver

Consider the following (simple) appointment scheduling problem.

Example 1. A person wants to plan a 3 working day long meeting after 22nd April 2005 and before May 2005. A colleague's visit of 5 days within the last two weeks in April 2005 must overlap with the planned meeting.

The problem described in Example 1 is illustrated in Figure 1.

To properly analyze and solve such a problem, we are led to an abstract analyze of *activities*[3] that take time, such as "meeting" and "visit". Such activities

[3] The notion "activity" is frequently used in Constraint Programming for objects having a temporal extent.

may have different *calendric types*, e.g. "meeting" has type "working day" (i.e. Monday to Friday) while "visit" has type "day". Those activities are related in time – either to (metric) temporal information, e.g. the meeting must be "before" 22nd April 2005 or (relatively) to each other, e.g. the visit must "overlap" the meeting.

Intuitively, a solution to the problem given in Example 1 must fulfill each of the temporal relationships stated on the activities. Formalizing this simple appointment scheduling problem as a *Constraint Satisfaction Problem (CSP)* [10], i.e. as a finite set of *variables*, a finite set of values that each variable can take (i.e. the *domains* of the variables), and a finite set of *constraints* that specify which values the variables can take simultaneously, a *solution* to the problem given in Example 1 is an assignment of values taken from the domains of the variables, one to each variable, such that each of the specified constraints is *satisfied*. A constraint is satisfied if none of its domains is empty. In a conventional CSP, the domains are all taken from the same set, e.g. integers or reals. However, in CaTTS-CL, the domains of activities may be taken from different calendric types (i.e. from different sets) which rely on and refer to types defined in CaTTS-DL calendar specifications. Fortunately, using CaTTS-DL's language constructs, the values of such calendric types are defined using integer indices relative to the index set of some other type defined in CaTTS-DL. Thus, in CaTTS-CL, CSPs can be modeled over integer domains (representing elements of different calendric types), involving CaTTS-specific constraints. Those specific constraints (called *conversion constraints*, introduced in Section 3.1) represent the relationships between different calendric types defined in CaTTS-DL.

The main aspect of constraint solving is to transform a given CSP into an equivalent CSP, i.e. the constraints of the CSP have the same set of solutions but a (considerably) smaller search space. Let us consider what this means for the problem described in Example 1: each variable represents a domain typed after a calendric type defined in a CaTTS-DL calendar specification. "Meeting" represents three working day long intervals. "Visit" represents 5 day long intervals. CaTTS-CL time constraints (e.g. "after"), stated in the problem illustrated in Example 1 are applied as long as it is possible to reduce the domains of "meeting" and "visit". This process of domain reduction also takes advantage of the different calendric types of the variables and constants used to describe the problem given in Example 1 by applying *conversion constraints* (cf. Section 3.1), characteristic to CaTTS. This process reduces the domain of "meeting" such that its possible starting working days must be between "25.4.2005" and "27.4.2005" and the domain of "visit" such that its possible starting days must be between "22.4.2005" and "24.4.2005". One solution to the problem given in Example 1 (computed by searching the reduced domains of "meeting" and "visit" such that each of the constraints is satisfied) is: the colleague arrives at 22nd April 2005 and leaves (5 days later) at 26th April 2005, and the meeting starts at 25th April 2005 and ends (3 working days later) at 27th April 2005.

3.1 Calendric Types and Conversion Constraints

Recall that in Constraint Programming variables are used to represent domains. A domain is a (finite) set of values of a type like integers. However, in the case of CaTTS-CL, variables represent domains which are finite sets of values of *any* calendric type defined in CaTTS-DL. Therefore, we need a means to "compare" such variables. Fortunately, this information is already provided with each CaTTS-DL type definition. Recall that a CaTTS-DL type is declared by a (user-defined) predicate that either defines an aggregation subtype, e.g.

```
type week = aggregate 7 day @ day(1);
```

i.e. `week` $\preceq$ `day` (read "`week` is an aggregation subtype of `day`") or an inclusion subtype, e.g.

```
type working_day = select day(i) where
      relative i in week >= 1 && relative i in week <= 5;
```

i.e. `working_day` $\subseteq$ `day` (read `working_day` is an inclusion subtype of `day`). The predicate `7 day @ day(1)` of type `week` specifies which day intervals define weeks; and the predicate `relative i in week => 1 && relative i in week <= 5` of type `day` specifies which days are also working days.

Joins. To ensure that not only the elements of a calendric type and its immediate supertype (e.g. according to Figure 2, `day` is the immediate supertype of `week`) can be compared, but also *any* pair of calendric types defined in one (or more) CaTTS-DL calendar specifications and used in a CaTTS-CL program, a generalized subtype relation for calendric types, in fact the union of the aggregation and the inclusion relations [2] is defined.

Definition 1. *Let σ and τ be calendric types defined in CaTTS-DL.*
σ is a **subtype of** *τ, denoted $\sigma \leq \tau$, iff either $\sigma \preceq \tau$ or $\sigma \subseteq \tau$, i.e. $\sigma \leq \tau := \sigma \preceq \tau \cup \sigma \subseteq \tau$.*

An example subtype relation of calendric types defined in CaTTS-DL is given in Figure 2.

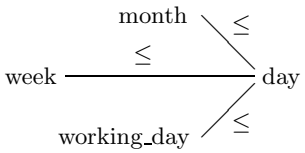

Fig. 2. Subtype relation of some calendric types defined in CaTTS-DL.

CaTTS' subtype relation "$\leq$" induces a partial order on the calendric types defined in a CaTTS-DL calendar specification. A formalization of calendars that can be expressed in CaTTS-DL is given in Definition 2.

Definition 2. *A* **calendar** *$C = \{\tau_1, \ldots \tau_n\}$ is a finite set of calendric types such that there exists a $\tau_i \in C$ and for all $\tau_j \in C$, $i, j \in \{1...n\}$ and $i \neq j$, $\tau_j \leq \tau_i$.*

Note that a finite set $\mathcal{S}_\mathcal{C}$ of CaTTS-DL calendars is also a calendar according to Definition 2, if either the τ_i (according to Definition 2) of the CaTTS-DL calendars in $\mathcal{S}_\mathcal{C}$ are *aligned*, i.e. identical except for the numbering of their indices or there exists a type τ_0 which is a supertype of the τ_i (according to Definition 2) of the calendars belonging to $\mathcal{S}_\mathcal{C}$.

To compare the domains of two variables of different types τ_s and τ_t during constraint solving in CaTTS, the *join* of τ_s and τ_t is computed and the domains are converted to the equivalent domains in the join type. A join in CaTTS is slightly weaker than the ordinary lattice join, which only allows for the first condition of Proposition 1. The second of Proposition 1 is an extension to deal with CaTTS-DL calendars which are not (always) lattices. This condition forces the join to be the smallest possible *unique*[4] upper bound. E.g. according to Figure 2, the join of types `working_day` and `month` is `day`. For each pair of calendric types defined in a CaTTS-DL calendar specification (according to Definition 2) such a join exists:

Proposition 1. *Let $(C, \leq)$ be a calendar.*
For any pair of calendric types τ_t and τ_s of C, there exists a join $\chi \in C$ such that $\tau_s \vee \tau_t = \chi$, i.e. $\tau_s \leq \chi$, $\tau_t \leq \chi$, and for all $\sigma_i \in C$ with $\tau_t \leq \sigma_i$ and $\tau_s \leq \sigma_i$, either

1. *$\chi \leq \sigma_i$ or*
2. *$\sigma_i < \chi$, and there exists another $\sigma_k \in C$ with $\tau_s, \tau_t \leq \sigma_k$, $\sigma_k < \chi$ and σ_i, σ_k being incomparable.*

Proof. For a pair of types τ and σ of calendar C, consider the set of upper bounds $U(\tau, \sigma) = \{\upsilon | \tau \leq \upsilon, \sigma \leq \upsilon\}$.
(Existence) If $\tau_s = \tau_t = \alpha$, with α being the top element of C, then $U(\tau_s, \tau_t) = \{\alpha\}$, and our proposition is satisfied through *(i)*. So, if $\tau_s \vee \tau_t$ exists, so does $\tau'_s \vee \tau_t$, with τ'_s direct subtype of τ_s: If $\tau_s \leq \tau_t$ so is $\tau'_s \leq \tau_t$ and in this case τ_t is the join, as $\tau_t \in U(\tau'_s, \tau_t)$ satisfies *(i)*. In case of $\tau_t < \tau_s$, either $\tau_t \leq \tau'_s$ and thus $\tau'_s \in U(\tau'_s, \tau_t)$ satisfies *(i)*, or else τ_t and τ'_s are incomparable and thus $\tau_s \in U(\tau'_s, \tau_t)$ satisfies *(i)*. Finally, if τ_t and τ_s are incomparable, τ'_s cannot be greater than or equal to τ_t, because then τ_t would have to be less than of equal τ_s; either τ'_s, too, is incomparable to τ_t and thus $\tau'_s \vee \tau_t = \tau_s \vee \tau_t \in U(\tau'_s, \tau_t)$ satisfies *(ii)*, or else $\tau'_s \leq \tau_t$ and thus $\tau_t \in U(\tau'_s, \tau_t)$ satisfies *(i)*.
(Uniqueness) Be $\chi = \tau_s \vee \tau_t$. Let's assume χ' would also qualify as a join of τ_s and τ_t. If χ' and χ were incomparable, then neither $\chi' \leq \chi$ nor $\chi < \chi'$ and thus χ' violates *(i)* and *(ii)*. If $\chi' < \chi$, then χ must have satisfied *(ii)*, thus exist an upper bound σ_k incomparable to χ'; however, all upper bounds are comparable to χ' if it is a join *(i,ii)*. Finally, if $\chi < \chi'$, then χ' must satisfy *(ii)*, thus exist an upper bound σ_k incomparable to χ, failing analogously. □

Informally, an algorithm to find a join (according to Proposition 1) consists in finding all join-candidates for a pair of types. If more than one such a candidate

[4] in terms of equality

exists, find the join for all those candidates. A reference implementation of the algorithm is given in Appendix C.

With Proposition 1, conversions, and, thus, constraint solving over arbitrary calendric domains (as long as their types are properly defined in a CaTTS-DL calendar specification) is ensured in CaTTS.

Conversions. CaTTS' constraint solver treats time constraints like "after" as ordinary finite domain constraints. To deal with domains of different calendric types, the domains need to be converted. For this purpose, CaTTS provides with *conversion constraints*. The basic idea of those conversion constraints is the following: assume that the variables X_{week} and Y_{day} with domains D^X_{week} (of type `week`) and D^Y_{day} (of type `day`) participate in the CaTTS-CL constraint $after$. To propagate this CaTTS-CL constraint, CaTTS' constraint solver provides with a *preprocessor*. During preprocessing, the join (according to Proposition 1) of types `week` and `day` (i.e. `day`) is inferred and conversion constraints for X_{week} and Y_{day} according to the join are added. Preprocessing refers to and relies on the typing and subtyping derivations inferred during type checking CaTTS-CL programs [9].

The conversion constraints propagate new variables with domains in the same type equivalent to the initial domains (i.e. the converted domains specify the same set of solutions). The conversion constraints are implemented by applying *translation functions* between CaTTS-DL subtypes. CaTTS' translation functions are given in Appendix B. In the following, the functioning of CaTTS' conversion constraints is exemplified.

Example 2. The domains in this example refer to the calendric types illustrated in Figure 1.

(i) Assume that the variable X_{week} represents the domain of one week long intervals starting in week 4 or 5 (denoted $X_{week} : 4..5+1$). Applying the conversion constraint for weeks and days (that uses the translation function generated from the type predicate of type `week`, i.e. from `7 day @ day(1)`) to X_{week}, X_{week} in days represents the domain $X_{day} : 18..25+7$, i.e. 7 day long intervals that must start between the first day in week 4 (i.e. 18) and the first day in week 5 (i.e. 25).

(ii) Assume that X_{day} represents the domain of 2 day long intervals starting between day 23 and 28 (denoted $X_{day} : 23..28+2$). Applying the conversion constraint for working days and days (that uses the translation function generated from the type predicate of type `working_day`, i.e. from `relative i in week >= 1 && relative i in week <= 5`) to X_{day}, X_{day} in working days represents the following domain $X_{working\_day} : 17..19+2$, i.e. 2 working day long intervals that must start between that working day succeeding day 23 (i.e. 17) and working day 19 such that $X_{working\_day}$ end on that working day preceeding day 30, i.e. the maximal ending day of X_{day}.

(iii) Applying the conversion constraint for weeks and days to $X_{day} : 23..28+2$ fails (i.e. the constraint is inconsistent); no two day long interval may represent a week.

3.2 Activities and Time Constraints

In CaTTS-CL, we only take into account the fact that *activities* like those of Example 1 take a finite continuous period of time over the reals, expressed in a calendric type defined in CaTTS-DL. Note that constraint reasoning on possibly non-continuous activities is a further (more complex) reasoning problem not considered in CaTTS. Thus, in CaTTS-CL activities have a duration and can be identified with closed, non-empty intervals of (integer) indices of a calendric type. This reflects a widespread common-sense understanding of time having a duration. However, CaTTS can deal with time point-like data like "22.4.2005". For this purpose, CaTTS' activities can be represented either as *events* or as *tasks*.[5] Events represent single values of a calendric type like "22.4.2005" of type "day" or "05.2005" of type "month". Events are modeled by finite domain variables with a calendric type. E.g. in CaTTS-CL `X is 1 day` specifies that the variable `X` represents an event of type `day`. Tasks represent intervals that have a starting point and a duration of a calendric type like "form `"22.4.2005"` to `"24.4.2005"`" of type `day` or "the last two weeks in February 2005" of type `week`. A starting point is modeled by a finite domain with a calendric type and the duration by an integer interval[6]. E.g. in CaTTS-CL `X is 5 day` specifies that the variable `X` represents tasks (with duration 5 days) of type `day`.

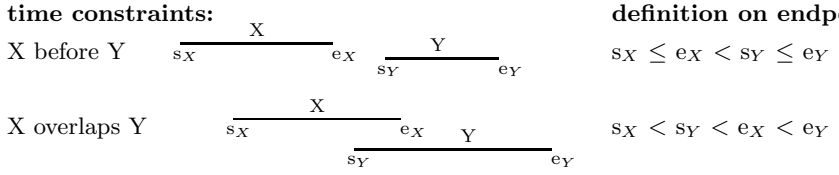

Fig. 3. Illustration of some of CaTTS-CL's time constraints used in Example 1.

In CaTTS-CL, *time constraints* are used to model conditions that must hold between activities like "before" and "overlaps". The time constraints provided with CaTTS-CL are, in particular, the thirteen interval relations introduced by Allen [11] (some are illustrated in Figure 3). Additionally, metric relations like "shift" (e.g. to shift a day forward by 3 days in time) are supported. CaTTS' complete syntax including the time constraints is given in Appendix A.

All CaTTS-CL time constraints are implemented by binary finite domain constraints on the ending points between activities of the same calendric type. Intuitively, the starting point and the ending point of an event are the same, e.g. the interval "22.4.2005" starts and ends at 22nd April 2005. Thus, the duration of an event is 1. Since each task is modeled by its starting point and its duration, the ending point can be computed by adding the duration to the starting point.

[5] The notions "event" and "task" are taken from research on "planning" and "scheduling", well-known kinds of CSPs over finite domains [10].

[6] Durations need to be represent by integer intervals to meet conversion constraints. E.g. converting months to days yields in a varying duration of day intervals.

E.g. if the starting point of task `X` (of type `day`) is represented by the finite domain `"18.04.2005".."27.04.2005"` and its duration is 5 (days), then its possible ending points can be commuted by adding the duration to the possible starting points, i.e. `"22.04.2005".."01.05.2005"`.

3.3 CaTTS' Constraint Propagation Algorithm

The main idea of constraint propagation is to reduce a given CSP to an equivalent CSP, i.e. the constraints of the CSP have the same set of solutions, but they are easier to solve. Algorithms that achieve such a reduction usually aim at reaching some form of *local consistency*. Local consistency means that some subparts of the considered CSP are consistent (i.e. have a solution). Achieving local consistency consists either in reducing the domains of the considered variables or in reducing the considered constraints. The notion of local consistency chosen for CaTTS is *arc-consistency* [10]:

Definition 3. *A constraint $C \subseteq D_1 \times \ldots \times D_n$ on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ is* **arc-consistent***, if for all $i \in \{1, \ldots, n\}$ and for all possible values of x_i in D_i, there exists values for all variables x_j $(j \neq i)$ in the respective domains D_j such that C is satisfied.*

A CSP is **arc-consistent** *if all its constraints are arc-consistent.*

Intuitively, a constraint C is arc-consistent if for every involved domain each value of it participates in a solution to C. The algorithm implementing arc-consistency in CaTTS is based on the logical formulation of Definition 3:

If $x_1 \in D_1 \wedge \ldots \wedge x_i \in D_i \wedge \ldots \wedge x_n \in D_n \wedge C(x_1, \ldots, x_n) \rightarrow x_i \in D'_i$, then $D_i \cap D'_i$ is the new domain of x_i. This notion of arc-consistency formulates the terminates criterion for CaTTS' constraint propagation algorithm, i.e. the algorithm terminates (and returns an equivalent but smaller CSP) if the CSP is arc-consistent and fails otherwise. Note that since constraint propagation results in a locally consistent CSP, an answer (i.e. the result of constraint propagation) to a CSP specified in CaTTS-CL might be inconsistent. To ensure (global) consistency, solutions must be computed by searching (using back tracking) the reduced domains.

In what follows, we discuss a few rules that allow us to manipulate time constraints and conversion constraints over activities (that represent finite domains of different calendric types). These rules, together with the above given formalization of a local consistency notion, define CaTTS' *constraint propagation algorithm*.

Reduction Rules for Time and Conversion Constraints – Constraint Propagation. CaTTS' time constraints are implemented by finite domain constraints on activity ending points with activities of the same calendric type (which is defined in CaTTS-DL). To relate activities of different calendric types in a CaTTS-CL program, conversion constraints are used. Conversion constraints are implemented by applying translation functions on activity domains. Those

translation functions rely on and refer to typing and subtyping derivations inferred from type checking CaTTS-CL programs [9]. CaTTS' translation functions are given in Appendix B.

The time constraints provided with CaTTS-CL are Allen's thirteen interval relations [11] as well as some metric constraints like shifts, both implemented on ending points. Since all those time constraints are implemented analogously (cf. Appendix B), we only discuss the domain reduction rule for the constraint *before:*

$$\frac{x, y \in \tau,\ \ x\ before\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ before\ y;\ \ x \in l_{sx}..(min(h_{sx} + d_x^+ - 1, h_{sy} - 1) - d_x^+ + 1) + (d_x^- : d_x^+), \\ y \in max(l_{sy}, l_{ex} + 1)..h_{sy} + (d_y^- : d_y^+) \end{array}}$$

The activities x and y, both of type τ, represent finite domain constraints of possible starting points and an integer (interval) of their durations (denoted $x \in l_{sx}..h_{sx} + (d_x^- : d_x^+)$, $y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)$). The ending points can be computed by adding the duration to the starting points (i.e. $l_{ex} := l_{sx} + d_x^- - 1$ and $h_{ex} := h_{sx} + d_x^+ - 1$). The constraint $x\ before\ y$ is propagated on the activities ending points (cf. Figure 3), i.e. the ending point of x must be less than the starting point of y. E.g. applying the domain reduction rule for *before* to the constraints $x\ before\ y$, $x \in$ "20.4.2005".."25.4.2005" + 1, and $y \in$ "18.4.2005".."23.4.2005" + 1 (both of type `day` according to Figure 1) yields in reduced domains for x and y such that $x \in$ "20.4.2005".."22.4.2005" + 1 and $y \in$ "21.4.2005".."23.4.2005" + 1.

CaTTS implements two different reduction rules for conversion constraints, one for aggregation subtypes and one for inclusion subtypes. Both kinds of conversion constraints depend on the corresponding aggregation (resp. inclusion) subtype definitions given in some CaTTS-DL calendar specification. For each subtype definition CaTTS automatically generates translation functions that compute the starting and ending points (for $\sigma \leq \tau$ denoted $p^s_{\sigma\to\tau}$ and $p^s_{\tau\to\sigma}$ and $p^e_{\sigma\to\tau}$ and $p^e_{\tau\to\sigma}$, reps.) and durations (denoted $p^d_{\sigma\to\tau}$ and $p^d_{\tau\to\sigma}$) in the corresponding subtype (resp. supertype). The generation rules for those translations are given in Appendix B.

The domain reduction rule for aggregation subtypes is given in the following:

$$\frac{x \in \sigma,\ y \in \tau,\ \sigma \preceq \tau\ \ convert(x, \sigma, y, \tau),\ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} convert(x, \sigma, y, \tau),\ x \in p^s_{\tau\to\sigma}(l_{sy})..(p^e_{\tau\to\sigma}(h_{ey}) - d_x^+ + 1) + p^d_{\tau\to\sigma}(d_y^- : d_y^+), \\ y \in p^s_{\sigma\to\tau}(l_{sx})..p^s_{\sigma\to\tau}(h_{sx}) + p^d_{\sigma\to\tau}(d_x^- : d_x^+) \end{array}}$$

E.g. $x \in$ "21.4.2005".."25.4.2005" + 7 of type `day` and $y \in$ "$W4$ 2005".."$W5$ 2005" + 1 of type `week` (according to Figure 1). Applying the previously given reduction rule to x and y yields in a reduction of the domain of x such that its starting points start 7 day long intervals corresponding to weeks of y, i.e. $x \in$ "25.4.2005".."25.4.2005" + 7 and y such that the weeks correspond to the 7 day long intervals of x, i.e. $y \in$ "$W5$ 2005".."$W5$ 2005" + 1.

The domain reduction rule for inclusion subtypes is given in the following:

$$\frac{x \in \sigma,\ y \in \tau,\ \sigma \subseteq \tau\ \ convert(x,\sigma,y,\tau);\ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} convert(x,\sigma,y,\tau);\ x \in p^s_{\tau\to\sigma}(l_{sy})..p^s_{\tau\to\sigma}(h_{sy}) + (p^e_{\tau\to\sigma}(l_{ey}) - p^s_{\tau\to\sigma}(l_{sy}) + 1 : \\ p^e_{\tau\to\sigma}(h_{ey}) - p^s_{\tau\to\sigma}(h_{sy}) + 1), \\ y \in p_{\sigma\to\tau}(l_{sx})..p_{\sigma\to\tau}(h_{sx}) + (p_{\sigma\to\tau}(l_{ex}) - p_{\sigma\to\tau}(l_{sx}) + 1 : p_{\sigma\to\tau}(h_{ex}) - p_{\sigma\to\tau}(h_{sx}) + 1) \end{array}}$$

E.g. $x \in$ "23.4.2005".."25.4.2005" $+2$ of type `day` (according to Figure 1) and y should represent corresponding values in `working_day`. Applying the previously given reduction rule to x and y would yield in a reduction of the domain of x such that it corresponds to working days, i.e. $x \in$ "25.4.2005".."25.4.2005" $+ 2$ and y would be reduced to the same set, i.e. $y \in$ "25.4.2005".."25.4.2005" $+ 2$ (y, of course, of type `working_day`).

Note that not considering domains of different calendric types would (i) not allow for as much search space reduction as possible with conversion constraints, hence, would be less efficient, and (ii) end up in loss of semantics (e.g. we won't no longer know which days are also working days). The two conversion constraints given above are corner stones of the CaTTS reasoner that, to the best of the knowledge of the authors, have not been proposed elsewhere. As the given examples show, they are very useful in reasoning on calendric and temporal data. Arguably, to efficiently solve problems like the one given in Example 1, one has to deal with a specific theory (that of calendars and time) which requires a specific treatment to make both search space restrictions and semantic support of calendric types, thus gain in efficiency, possible. Using constraint programming techniques to efficiently solve problems over calendar domains has similar advantages compared to those of "paramodulation" [12] used to efficiently reason with equality in first-order languages.

4 Related Work

CaTTS complements data type definition languages and data modeling and reasoning methods for the Semantic such as XML Schema [13], RDF [6], and OWL [7]: XML Schema provides a considerably large set of predefined time and date data types dedicated to the Gregorian calendar whereas CaTTS enables user-defined data types dedicated to any calendar. RDF and OWL are designed for *generic* Semantic Web applications. In contrast, CaTTS provides with methods *specific* to particular application domains, that of calendars and time. Thus, CaTTS' reasoner which is based on constraint solving techniques and dedicated to CaTTS-DL calendar specifications is specific to calendar and time reasoning.

CaTTS departs from time ontologies such as the DAML Ontology of Time [14] or time in OWL-S [15]: CaTTS' constraint solver is dedicated to calendric types defined in CaTTS-DL; this dedication makes both considerable search space restrictions, hence gains in efficiency, and support of calendric and temporal data with different calendric types possible. While (time) ontologies follow the (automated reasoning) approach of "axiomatic reasoning", CaTTS is based on a (specific) form of "theory reasoning" [4, 2, 5], a well-known example of which is

paramodulation [12]. Like paramodulation ensures efficient processing of equality in resolution theorem proving, CaTTS provides the user with convenient constructs for calendric types and efficient processing of data and constraints over those types.

CaTTS' constraint solver is intended to be, in principle, used with *any* (Semantic) Web language (e.g. XQuery, XSLT, OWL) as far as the calendric and temporal data used is typed after calendric types defined in a CaTTS-DL calendar specification. CaTTS-DL calendar specifications provide, similar to XML Schemas, schemas for the data used, however, specific to time and calendars.

5 Conclusion

This article has introduced a reasoner based on constraint programming techniques for calendric and temporal data. Such a reasoner is necessary for Semantic Web applications and Web services like appointment and travel scheduling that involved often complex temporal and calendric data.

The reasoner provides with novel constraints, the *conversion constraints*, to convert domains over different calendric types without loss of semantics. Those conversion constraints provide a natural way to obtain temporal constraint reasoning with domains of different calendric types. Note that temporal reasoning with domains of different types is frequently considered in current research. The authors believe that CaTTS offers a particularly convenient and intuitive manner to solve temporal problems involving arbitrary calendric domains.

The proposed reasoner is part of the type language CaTTS: calendric types defined in CaTTS' definition language CaTTS-DL and referred to in CaTTS' constraint language CaTTS-CL are used to automatically generate translation functions which are applied when propagating conversion constraints on temporal and calendric data.

The proposed reasoner offers Semantic Web applications a means to benefit from the advantages of constraint programming techniques when dealing with specific theories like time and calendars. The reasoner can be, in principle, used with any (Semantic) Web language.

Acknowledgment

This research has been funded in part by the PhD Program Logics in Computer Science (GKLI) and the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Program project REWERSE number 506779 (cf. http://rewerse.net).

References

1. Bry, F., Haußer, J., Rieß, F.A., Spranger, S.: Cultural Calendars for Programming and Querying. In: Proc. 1^{st} Forum on the Promotion of European and Japanese Culture and Traditions in Cyber Society and Virtual Reality, France. (2005)

2. Bry, F., Rieß, F.A., Spranger, S.: CaTTS: Calendar Types and Constraints for Web Applications. In: Proc. 14^{th} Int. World Wide Web Conference, Japan. (2005)
3. Bry, F., Spranger, S.: Towards a Multi-calendar Temporal Type System for (Semantic) Web Query Languages. In: Proc. 2^{nd} Int. Workshop Principles and Practice in Semantic Web Reasoning. LNCS 3208, Springer-Verlag (2004)
4. Stickel, M.E.: Automated Deduction by Theory Reasoning. Journal of Automated Reasoning **1** (1985) 333–355
5. Bry, F., Marchiori, M.: Ten Theses on Logic Languages for the Semantic Web. In: Proc. W3C Workshop on Rule Languages for Interoperability, USA. (2005)
6. W3C, World Wide Web Consortium: RDF Primer. (2004)
7. W3C, World Wide Web Consortium: OWL Web Ontology Language. (2004)
8. Dershowitz, N., Reingold, E.: Calendrical Calculations: The Millennium Edition. Cambridge University Press (2001)
9. Bry, F., Rieß, F.A., Spranger, S.: A Type Language for Calendars. submitted to publication (2005)
10. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (2003)
11. Allen, J.F.: Maintaining Knowledge about temporal Intervals. Com. of the ACM **26** (1983) 832–843
12. Robinson, G., Wos, L.: Paramodulation and Theorem Proving in First Order Theories. Machine Intelligence **4** (1969) 135–150
13. W3C, World Wide Web Consortium: XML Schema Part 2: Datatypes. (2001)
14. DARPA Agent Markup Language: A DAML Ontology of Time. (2002)
15. Pan, F., Hobbs, J.R.: Time in OWL-S. In: Semantic Web Services, AAAI Spring Symposium Series. (2004)
16. Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)

A The Syntax of CaTTS

c ::=	*constraints:*
true	
false	
X is 1 τ	*event*
X is τ	*task*
X is n τ	*task + duration* $n \in \mathbb{N}$
X intervalC Y	*interval constraint*
e intervalC Z	
X metricC Y	*metric constraint*
e metricC Z	
c && c	*conjunction*

e ::=	*expressions:*
X	*variable*
d	*CaTTS-FDL date*
τ(i)	*part,* $i \in \mathbb{Z}$
n τ	*duration,* $n \in \mathbb{N}$
[e..e]	*endpoint interval*
e upto e	*duration interval*
e downto e	*duration interval*
binOp e e	*binary operation*
unOp e	*unary operation*

binOp ::= + | − | ∗ | mod | div | shift forward | shift backward | min | max | avg | extend by | shorten by | relative to | relative in
unOp ::= duration | begin | end | index |
intervalC ::= equals | before | after | during | contains | starts | started_by | finishes | finished_by | meets | met_by | overlaps | overlapped_by | within | on_or_before | on_or_after
metricC ::= == | <= | < | > | >= | ! =

τ ::=	*type expressions:*
reference	*(user-def. or predef.) reference type*
refinement n @ e	*refinement,* $n \in \mathbb{N}$
aggregate e {,e} @ e	*(abs. anchored) aggregation*
aggregate e {,e} ˜@ z	*(rel. anchored) aggregation,* $z \in \mathbb{Z}$
select e where c	*selection*
τ^n	*duration*
τ^*	*time interval*
$\tau \& \tau$	*conjunction*
$\tau \mid \tau$	*disjunction*
$\tau \setminus \tau$	*exception*
$\tau \# < \tau$	*restriction*

B CaTTS' Constraint Propagation Algorithm

B.1 Domain Reduction Rules for Time Constraints

Time Constraints over Activities The following notations are used with the subsequently given domain reduction rules for time constraints over activities.

- $x \in D_x$, x represents the domain D_x where D_x is represented by
  - its starting point, represented by a finite domain constraint, i.e. $l_{sx}..h_{sx}$ (abrv. S_x), and
  - its (possibly impricise duration), i.e. $(d_x^- : d_x^+)$ (abrv. d_x)

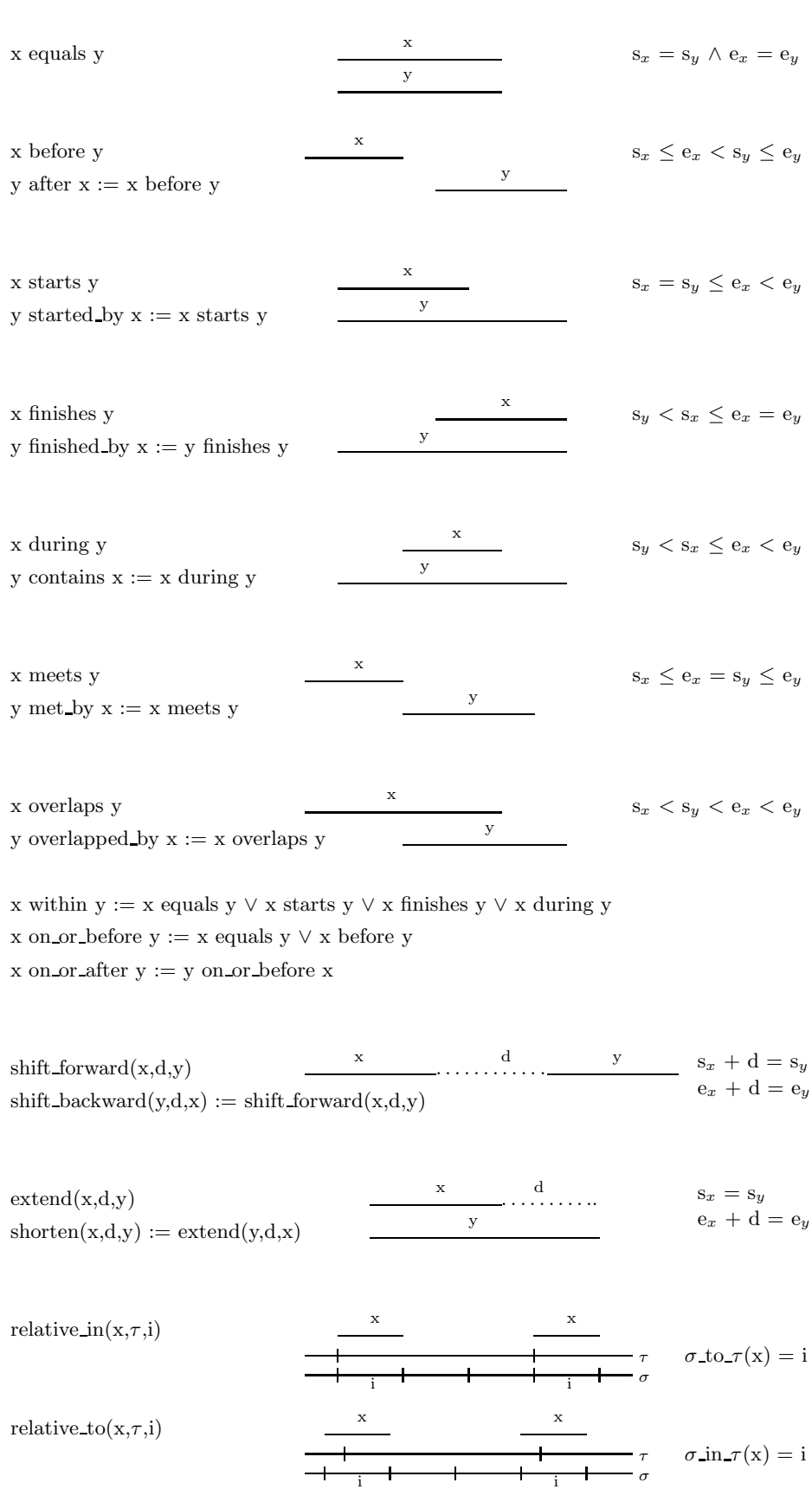

Fig. 4. Illustrative overview of CaTTS' binary time constraints.

- the ending point (denoted $l_{ex}..h_{ex}$, abrv. E_x) of x can be computed from its starting point and its duration, i.e. $l_{ex} := l_{sx} + d_x^- - 1$ and $h_{ex} := h_{sx} + d_x^+ - 1$ (abrv. $E_x := S_x + d_x - 1$)
- $[i_1, \ldots, i_k]$ denotes the list representation of the domain D_x
- $min(a, b)$ denotes the minimum of elements a and b
- $max(a, b)$ denotes the maximum of elements a and b
- $D_x \cap D_y$ denotes the intersection of the domains D_x and D_y
- the symbol $\in$ is overloaded:
  - $x \in \tau$ is read, the variable x is of type τ
  - $x \in D_x$ is read, the variable x represents the domain (constraint) D_x, i.e. x may be assigned to each of the values represented by D_x
- $\sigma\_\text{in}\_\tau(\text{x})$ denotes the function that computes all the elements of type τ that contain the elements of x (of type σ) and $\sigma\_\text{to}\_\tau(\text{x})$ those elements of τ overlapping with the elements of x; note that those functions are generated from user-defined predicates of calendric types defined in CaTTS-DL

Note that all constraints over activities are equally defined for events and tasks since an event $x \in l_{sx}..h_{sx}$ may be represented by a task $x' \in l_{sx}..h_{sx} + (1:1)$, i.e. events are represented by intervals with duration 1 and tasks are intervals with duration greater or equal to 1.

Representation of CaTTS-CL's unary operations as unary constraints:

$$\frac{duration\ x;\quad , x \in S_x + d_x}{duration\ x;\quad y \in d_x}$$

$$\frac{begin\ x;\quad , x \in l_{sx}..h_{sx} + d_x}{begin\ x;\quad y \in l_{sx}..h_{sx}}$$

$$\frac{end\ x;\quad , x \in l_{sx}..h_{sx} + d_x}{begin\ x;\quad y \in l_{ex}..h_{ex}}$$

$$\frac{index\ x;\quad , x \in l_{sx}..h_{sx} + d_x}{begin\ x;\quad y \in [i_{l_{sx}}, \ldots, i_{h_{ex}}]}$$

Some basic simplifications for time constraints over activities:

$$\frac{x\ equals\ x;\quad x \in D_x}{;\quad x \in D_x}$$

$$\frac{x\ before\ x;\quad x \in D_x}{;\quad x \in \emptyset}$$

$$\frac{x\ starts\ x;\quad x \in D_x}{;\quad x \in \emptyset}$$

$$\frac{x\ finishes\ x;\ \ x \in D_x}{;\ \ x \in \emptyset}$$

$$\frac{x\ during\ x;\ \ x \in D_x}{;\ \ x \in \emptyset}$$

$$\frac{x\ meets\ x;\ \ x \in D_x}{;\ \ x \in \emptyset}$$

$$\frac{x\ overlaps\ x;\ \ x \in D_x}{;\ \ x \in \emptyset}$$

$$\frac{x, y \in \tau,\ \ x\ starts\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y \mid d_x \geq d_y}{;\ \ \bot}$$

$$\frac{x, y \in \tau,\ \ x\ finishes\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y \mid d_x \geq d_y}{;\ \ \bot}$$

$$\frac{x, y \in \tau,\ \ x\ during\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y \mid d_x \geq d_y}{;\ \ \bot}$$

$$\frac{x, y \in \tau,\ \ x\ overlaps\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y \mid d_x == 1}{;\ \ \bot}$$

$$\frac{x, y \in \tau,\ \ x\ overlaps\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y \mid d_y == 1}{;\ \ \bot}$$

Propagations for time constraints over activities:

$$\frac{x, y \in \tau,\ \ x\ equals\ y;\ \ x \in D_x,\ y \in D_y}{x\ equals\ y;\ \ , x \in D_x \cap D_y,\ y \in D_x \cap D_y}$$

$$\frac{x, y \in \tau,\ \ x\ before\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ before\ y;\ \ x \in l_{sx}..(min(h_{sx} + d_x^+ - 1, h_{sy} - 1) - d_x^+ + 1) + (d_x^- : d_x^+), \\ y \in max(l_{sy}, l_{ex} + 1)..h_{sy} + (d_y^- : d_y^+) \end{array}}$$

$$\frac{x, y \in \tau,\ \ x\ starts\ y;\ \ x \in S_x + (d_x^- : d_x^+),\ y \in S_y + (d_y^- : d_y^+)}{x\ starts\ y;\ \ x \in S_x \cap S_y + (d_x^- : d_x^+),\ y \in S_x \cap S_y + (d_y^- : d_y^+)}$$

$$\frac{x, y \in \tau,\ \ x\ starts\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ starts\ y;\ \ x \in l_{sx}..min(h_{sx}, h_{e_y} - 1 - d_x^+ + 1) + (d_x^- : d_x^+), \\ y \in max(l_{sy}, l_{ex} + 1 - d_y^- + 1)..h_{sy} + (d_y^- : d_y^+) \end{array}}$$

$$\frac{x, y \in \tau,\ \ x\ finishes\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ finishes\ y;\ \ x \in l_{sx}..min(h_{sx}, h_{sy} - 1) + (d_x^- : d_x^+), \\ y \in max(l_{sy}, l_{sx} + 1)..h_{sy} + (d_y^- : d_y^+) \end{array}}$$

$$\frac{x, y \in \tau,\ \ x\ finishes\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y}{x\ finishes\ y;\ \ x \in (E_x \cap E_y - d_x + 1) + d_x,\ y \in (E_x \cap E_y - d_y + 1) + d_y}$$

$$\frac{x, y \in \tau,\ \ x\ during\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ during\ y;\ \ x \in max(l_{sx}, l_{sy} + 1)..h_{sx} + (d_x^- : d_x^+), \\ y \in l_{sy}..min(h_{sy}, h_{sx} - 1) + (d_y^- : d_y^+) \end{array}}$$

$$\frac{x, y \in \tau,\ \ x\ during\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ during\ y;\ \ x \in l_{sx}..min(h_{sx}, h_{ey} - 1 - d_x^+ + 1) + (d_x^- : d_x^+), \\ y \in max(l_{sy}, l_{ex} + 1 - d_y^- + 1)..h_{sy} + (d_y^- : d_y^+) \end{array}}$$

$$\frac{x, y \in \tau,\ \ x\ meets\ y;\ \ x \in S_x + d_x,\ y \in S_y + d_y}{x\ meets\ y;\ \ x \in (E_x \cap S_y - d_x + 1) + d_x,\ y \in (E_x \cap S_y) + d_y}$$

$$\frac{x, y \in \tau,\ \ x\ overlaps\ y;\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} x\ overlaps\ y;\ \ x \in max(l_{sx}, l_{sy} + 1 - d_x^- + 1)..h_{sx} + (d_x^- : d_x^+), \\ y \in l_{sy}..min(h_{sy}, h_{ex} - 1) + (d_y^- : d_y^+) \end{array}}$$

$$\frac{shift\_forward(x, d, y);\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ d \in d^- : d^+}{shift\_forward(x, d, y);\ \ , x \in l_{sx} + d^-..h_{sx} + d^+ + (d_x^- : d_x^+)}$$

$$\frac{shift\_forward(x, d, y);\ \ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+),\ d \in d^- : d^+}{shift\_forward(x, d, y);\ \ , y \in l_{sy} - d^-..h_{sy} - d^+ + (d_y^- : d_y^+)}$$

$$\frac{shift\_forward(x, d, y);\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{shift\_forward(x, d, y);\ \ , d \in l_{sy} - h_{ex} : h_{sy} - l_{ex}}$$

$$\frac{extend\_by(x, d, y);\ \ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ d \in d^- : d^+}{extend\_by(x, d, y);\ \ , y \in l_{sx}..h_{sx} + (d_x^- + d^- : d_x^+ + d^+)}$$

$$\frac{extend\_by(x, d, y);\ \ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+),\ d \in d^- : d^+}{extend\_by(x, d, y);\ \ , x \in l_{sy}..h_{sy} + (d_y^- - d^- : d_y^+ - d^+)}$$

$$\frac{x \in \sigma,\ y \in \tau,\ \ relative\_in(x, \tau, i);\ \ x \in D_x}{relative\_in(x, \tau, i);\ \ i \in [\sigma\_in\_\tau(D_x)]}$$

$$\frac{x, y \in \tau,\ \ relative\_to(x, \tau, i);\ \ x \in D_x}{relative\_to(x, \tau, i);\ \ i \in [\sigma\_to\_\tau(D_x)]}$$

Time Constraints over Durations and Indices With some applications it becomes necessary not only to reason over activities but also over durations (and indices of activities). To this purpose, durations (and indices) are represented by ordinary finite domain constraints with interval representation, i.e. for durations $x_d \in l_d..h_d$ (and for indices $x_i \in l_i..h_i$). The equality, inequality, and disequality

constraints (i.e. == | <= | < | > | >= | ! =) over durations and indices are lineary constraints over integer intervals as given in [16] for example. Note that as it is the case of CaTTS' time constraints over activities, time constraints over durations (and indices) are defined for variables representing durations (and indices) from the same type (defined in some CaTTS-DL calendar specification).

B.2 Domain Reduction Rules for Conversion Constraints

The domain reduction rule for aggregation subtypes is given in the following:

$$\frac{x \in \sigma,\ y \in \tau,\ \sigma \preceq \tau\ \ convert(x,\sigma,y,\tau),\ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} convert(x,\sigma,y,\tau),\ x \in p^s_{\tau\to\sigma}(l_{sy})..(p^e_{\tau\to\sigma}(h_{ey}) - d_x^+ + 1) + p^d_{\tau\to\sigma}(d_y^- : d_y^+) \\ y \in p^s_{\sigma\to\tau}(l_{sx})..p^s_{\sigma\to\tau}(h_{sx}) + p^d_{\sigma\to\tau}(d_x^- : d_x^+) \end{array}}$$

The domain reduction rule for inclusion subtypes is given in the following:

$$\frac{x \in \sigma,\ y \in \tau,\ \sigma \subseteq \tau\ \ convert(x,\sigma,y,\tau);\ x \in l_{sx}..h_{sx} + (d_x^- : d_x^+),\ y \in l_{sy}..h_{sy} + (d_y^- : d_y^+)}{\begin{array}{c} convert(x,\sigma,y,\tau);\ x \in p^s_{\tau\to\sigma}(l_{sy})..p^s_{\tau\to\sigma}(h_{sy}) + (p^e_{\tau\to\sigma}(l_{ey}) - p^s_{\tau\to\sigma}(l_{sy}) + 1 : \\ p^e_{\tau\to\sigma}(h_{ey}) - p^s_{\tau\to\sigma}(h_{sy}) + 1) \\ y \in p_{\sigma\to\tau}(l_{sx})..p_{\sigma\to\tau}(h_{sx}) + (p_{\sigma\to\tau}(l_{ex}) - p_{\sigma\to\tau}(l_{sx}) + 1 : p_{\sigma\to\tau}(h_{ex}) - p_{\sigma\to\tau}(h_{sx}) + 1) \end{array}}$$

B.3 Translation Functions for Conversion Constraints

The translation functions $p_{\sigma\to\tau}$ and $p_{\tau\to\sigma}$ used with CaTTS' conversion constraints are generated from any pair of types σ and τ with $\sigma \leq \tau$ (i.e. σ is a (inclusion or aggregation) subtype of τ) defined in a CaTTS-DL calendar specification. That means, such functions are generated for any pair of subtypes in the reflexive and transitive closure of a CaTTS-DL calendar specification according to the $\leq$- relation.

Translation Functions for Aggregations.

Periodic Aggregations. CaTTS-DL type declaration (pattern) for periodic aggregations has the following form:

type σ = **aggregate** $\mathrm{d}_1\ \tau, \ldots \mathrm{d}_k\ \tau$ @ τ(a);

where

- d_i, $i \in \{1, \ldots k\}$ is the duration of some value of type σ in terms of values of type τ,
- k is the length of the ordered periodic pattern of k values of type σ, and
- a the anchor index oftype σ in τ

E.g. "week" defined from "day":

type week = **aggregate** 7 day @ day(1);

From such a periodic aggregation type declaration in CaTTS-DL the following translation functions $p^s_{\sigma\to\tau}$ (start), $p^e_{\sigma\to\tau}$ (end), $p^d_{\sigma\to\tau}$ (duration), $p^s_{\tau\to\sigma}$ (starting sucsessor), $p^e_{\tau\to\sigma}$ (ending predecessor), and $p^d_{\tau\to\sigma}$ (duration) which are used with CaTTS' conversion constraints are (automatically) generated from CaTTS-DL type declarations by the following definitions:

$$\begin{aligned}
p^s_{\sigma\to\tau}(i) &:= \textbf{let } i \mathit{ mod } k = m \\
&\quad \textbf{in } (d_1 + \ldots + d_k) \times ((i-1) \mathit{ div } k) + a + (d_1 + \ldots + d_{m-1}) \\
\\
p^e_{\sigma\to\tau}(i) &:= \textbf{let } i \mathit{ mod } k = m \\
&\quad \textbf{in } (d_1 + \ldots + d_k) \times ((i-1) \mathit{ div } k) + a + (d_m + \ldots + d_k - 1) \\
\\
p^d_{\sigma\to\tau}(d^-_x : d^+_x) &:= (d^-_x \times \mathit{min}(d_1, \ldots d : k) : d^+_x \times \mathit{max}(d_1, \ldots, d_k)) \\
\\
p^s_{\tau\to\sigma}(i) &:= \textbf{if } (i-a) \mathit{ mod } (d_1 + \ldots + d_k) == 0 \\
&\quad \textbf{then } (i-a) \mathit{ div } (d_1 + \ldots + d_k + 1) \\
&\quad \textbf{else } (i-a) \mathit{ div } (d_1 + \ldots + d_k + 2) \\
\\
p^e_{\tau\to\sigma}(i) &:= \textbf{if } (i-a) \mathit{ mod } (d_1 + \ldots + d_k) == d_1 + \ldots + d_k - 1 \\
&\quad \textbf{then } (i-a) \mathit{ div } (d_1 + \ldots + d_k + 1) \\
&\quad \textbf{else } (i-a) \mathit{ div } (d_1 + \ldots + d_k) \\
\\
p^d_{\tau\to\sigma}(d^-_y : d^+_y) &:= (d^-_y \mathit{ div } \mathit{min}(d_1, \ldots d_k) : d^+_y \times \mathit{ div } \mathit{max}(d_1, \ldots d_k))
\end{aligned}$$

Periodic Aggregations with finite many Exceptions. E.g. "month" defined from "day":

```
type month = aggregate
   31 day named january,
   alternate month(i)
    | ((i div 12)+1) mod 4 == 0 &&
        (((i div 12)+1) mod 100 != 0
         || ((i div 12)+1) mod 400 == 0) -> 29 day
      | otherwise                        -> 28 day
   end named february,
   31 day named march,
   30 day named april,
   31 day named may,
   30 day named june,
   31 day named july,
   31 day named august,
   30 day named september,
   31 day named october,
   30 day named november,
   31 day named december @ day(1);
```

Instead of d_i for $i \in \{1, \ldots k\}$ we have $\mathrm{D}_{i\in\{1,\ldots k\}}$, where every D_i is the set of possible durations for a specific phase i. In the cases of "month", these would be $\mathrm{D}_{j\in\{1,3,5,7,8,10,12\}} = \{31\}$, $\mathrm{D}_{l\in\{4,6,9,11\}} = \{30\}$ and $\mathrm{D}_2 = 28, 29$. Additionally,

let D= $\bigcup D_{i\in\{1,\ldots k\}}$ and L= $\{\bar{d}|\bar{d} = \sum_{i=1}^{k} d_i, d_i \in D_i\}$ the set of possible cycle lengths (for months, L= $\{265, 266\}$).

$$\begin{aligned} p^{d}_{\sigma\to\tau}(d_x^- : d_x^+) := \textbf{let}\ & (d_x^- \ mod\ k) \times min(D) = \dot{d}_x^- \\ & (d_x^+ \ mod\ k) \times max(D) = \dot{d}_x^+ \\ & (d_x^- \ div\ k) \times min(L) = c^- \\ & (d_x^+ \ div\ k) \times max(L) = c^+ \\ \textbf{in}\ & (\dot{d}_x^- + c^- : \dot{d}_x^+ + c^+) \end{aligned}$$

$p^{d}_{\tau\to\sigma}(d_y^- : d_y^+) := ((d_x^- \times k)\ div\ max(L) : (d_x^+ \times k)\ div\ min(L))$ (???)

Other conversions need additional facilities of constraint solving and memoization. This makes them not fit into the above mentioned pattern.

Translation Functions for Inclusions. CaTTS' generates translation functions for each kind of predicate that can be specified for the `select` type constructor.

Time Constraint "relative i in τ". The following notations are used for the subsequently given translation functions:

- $\sigma \preceq \tau$
- d denotes the duration of σ in τ (according to the aggregation subtype definition of τ from σ)
- a denotes the anchor of σ in τ (according to the aggregation subtype definition of τ from σ)
- $length([l..k])$ denotes the length of the list $[l..k]$ (from element l to element k)

Case 1:

```
type ρ = select σ(i) where relative i in τ == k;
```

Defining ρ as an inclusion subtype of σ, i.e. $\rho \subseteq \sigma$.

From such an inclusion type declaration in CaTTS-DL the following translation functions which are used with CaTTS' conversion constraints are (automatically) generated from CaTTS-DL type declarations by the following definitions:

$$p_{\rho\to\sigma}(i) := d \times (i-1) + a + k - 1$$

$$\begin{aligned} p^{succ}_{\sigma\to\rho}(i) := \textbf{let}\ & \tau\_of\_\sigma = ((i-1)\ div\ d) + i\ \textbf{in} \\ & \textbf{if}\ (((i-1)\ mod\ d) + 1) <= k \\ & \textbf{then}\ \tau\_of\_\sigma \\ & \textbf{else}\ \tau\_of\_\sigma + 1 \end{aligned}$$

$$\begin{aligned} p^{pred}_{\sigma\to\rho}(i) := \textbf{let}\ & \tau\_of\_\sigma = ((i-1)\ div\ d) + i\ \textbf{in} \\ & \textbf{if}\ (((i-1)\ mod\ d) + 1) < k \\ & \textbf{then}\ \tau\_of\_\sigma - 1 \\ & \textbf{else}\ \tau\_of\_\sigma \end{aligned}$$

Case 2:

```
type ρ = select σ(i) where
           relative i in τ >= k && relative i in τ <= l;
```

Defining ρ as an inclusion subtype of σ, i.e. $\rho \subseteq \sigma$.

From such an inclusion type declaration in CaTTS-DL the following translation functions which are used with CaTTS' conversion constraints are (automatically) generated from CaTTS-DL type declarations by the following definitions:

$$p_{\rho\to\sigma}(i) := d \times ((i-1)\ div\ length([l..k])) + a + ((i-1)\ mod\ length([l..k]))$$

$$\begin{aligned} p^{succ}_{\sigma\to\rho}(i) := \ & \textbf{let}\ \ \tau\_of\_\sigma = ((i-1)\ div\ d) + i\ \ \textbf{in} \\ & \textbf{if}\ \ (((i-1)\ mod\ d) + 1) >= l\ \ \textbf{and}\ \ (((i-1)\ mod\ d) + 1) <= k \\ & \textbf{then}\ \ (length([l..k]) \times \tau\_of\_\sigma) - length([l..k]) + (i\ mod\ d) \\ & \textbf{elseif}\ \ (((i-1)\ mod\ d) + 1) < l\ \ \textbf{then}\ \ (length([l..k]) \times (\tau\_of\_\sigma) - 1)) + 1 \\ & \textbf{else}\ \ (length([l..k]) \times \tau\_of\_\sigma) + 1 \end{aligned}$$

$$\begin{aligned} p^{pred}_{\sigma\to\rho}(i) := \ & \textbf{let}\ \ \tau\_of\_\sigma = ((i-1)\ div\ d) + i\ \ \textbf{in} \\ & \textbf{if}\ \ (((i-1)\ mod\ d) + 1) >= l\ \ \textbf{and}\ \ (((i-1)\ mod\ d) + 1) <= k \\ & \textbf{then}\ \ (length([l..k]) \times \tau\_of\_\sigma) - length([l..k]) + (i\ mod\ d) \\ & \textbf{elseif}\ \ (((i-1)\ mod\ d) + 1) < l\ \ \textbf{then}\ \ length([l..k]) \times (\tau\_of\_\sigma) - 1) \\ & \textbf{else}\ \ length([l..k]) \times \tau\_of\_\sigma \end{aligned}$$

The specifications of the translation functions for further inclusion subtypes will be given in an later version of this article.

C CaTTS-Joins of Calendric Types

The following algorithm computes the join for a pair of calendric types defined in a CaTTS-Dl calendar specification according to Proposition 1. The algorithm is given in Prolog.

```
upperbound(X,X,X) :- !.
upperbound(X,Y,U) :- direct_supertype(X,SuperX), upperbound(SuperX,Y,U).
upperbound(X,Y,U) :- direct_supertype(Y,SuperY), upperbound(X,SuperY,U).

allupperbounds(X,Y,Us) :- bagof(U,upperbound(X,Y,U),Us).

multi_allupperbounds([X,Y|Rest],[Us|Uss]) :- allupperbounds(X,Y,Us),
  multi_allupperbounds(Rest,Uss).
multi_allupperbounds([Last],[Last]).
multi_allupperbounds([],[]).

concat([A,B|Rest],Concatenation) :- append(A,B,AB),
  concat([AB|Rest],Concatenation).
concat([Result],Result).
concat([],[]).

union(Set,Union) :- concat(Set,MUnion), remove_duplicates(MUnion,Union).

multi_join(Items,J) :- multi_allupperbounds(Items,Uss),
  (Us = [[R]] -> J = R; union(Uss,Us), multi_join(Us,J))

join(X,Y,J) :- multi_join([X,Y],J).
```