Dates & Times: An Annotated VDM Specification

by

Paul Chisholm

Document History

Version	Date	Description of Changes
1	May 2016	First release
2	July 2016	Remove module listings; refer to Overture web site Incorporate ISO 8601 supplement module
3	July 2017	Revised specification to incorporate **eq** and **ord** clauses

Table of Contents

1 Introduction

A large proportion of automated systems have a fundamental dependence on dates, times, intervals and durations. Air traffic management for example could not be conducted effectively without addressing crucial temporal dependencies that allows prediction of where an aircraft is expected to be at any point in time (the trajectory). The concept of date, time, interval and duration is vital for the precise and accurate delivery of many kinds of services.

In many ways, dates and times are so ubiquitous they are taken for granted, often captured simply with the statement (in the aviation domain) that a system employs Co-ordinated Universal Time (UTC). The programming languages employed to implement systems provide libraries for the manipulation of dates and times. These libraries are not standardised, providing varied application programming interfaces (API), and informal documentation.

This paper presents a formal specification of dates, times, intervals and durations based on International Organization for Standardisation (ISO) 8601 [1], and Request for Comments (RFC) 3339 [2]. The formalism employed is the Vienna Development Method Specification Language (VDM-SL) [3][4][1]. Temporal concepts are not core to VDM-SL. The intent is to provide a standard library that can be used in the specification of any system that requires dates, times, intervals and durations.

2 Overview

In general, we would like specifications to be as high level and abstract as possible; the purpose is to specify the 'what' with no concern for the 'how'. The aim of this specification is rather more fundamental than a typical business focussed problem [5], such as the management of aircraft flight plans. Consequently, the specification is rather low level, and in fact can be viewed as a functional program, but the goal is not to implement, it is to communicate the semantics of dates, times, intervals and durations. The focus of the 'programs' is to present the meaning as clearly as possible; no concern is given to efficiency or concrete representation.

If a specification is limited to a subset of the full VDM-SL language, the specification can be 'simulated' or 'animated'; that is, the functions being specified can be provided with input, and the simulation delivers the result as per the specification. The simulation does not merely compute the result, it verifies all pre-conditions, post-conditions and invariants during simulation. This provides a capability to 'test' the specification and increase confidence in its correctness [6] (also known as model checking).

Due to the low level nature of the date/time specification, it is restricted to the subset of VDM-SL that allows simulation. It is expected this specification will be employed by others that have a dependence on dates and times, and the authors of those models may want to carry out model checking. To enable such an approach, it is essential this specification allows simulation.

We employ VDM-SL with modules. The specification focusses on purely functional descriptions; state variables and operations are not employed. The presentation is limited to explicit definitions with decidable constraints[2]; where undecidable constraints add value, they are included as comments.

The VDM specification is implemented in the Overture Tool [7], which provides syntax and type checking of the specification, and supports simulation and testing of specifications.

[1] Often the term VDM on its own is used when VDM-SL is intended.
[2] A constraint is decidable if its truth or falsity can be computed automatically in finite time.

3 ISO 8601/RFC 3339

ISO 8601 [1] is the International Organization for Standardization's recommendation for the numeric representation of dates and times. ISO 8601 is a wide ranging and complex specification. RFC 3339 [2] is a profile of ISO 8601 aimed at internet protocols that restricts attention to those aspects of ISO 8601 that are commonly used. This specification is aligned with RFC 3339 for dates and times with the following exceptions:

- A seconds value of 60 (leap second) is not supported;
- RFC 3339 (and ISO 8601) impose no limit on the resolution of times; the finest granularity of this specification is milliseconds.

RFC 3339 does not support time intervals and durations. This specification adopts the following subset of ISO 8601:

- All intervals are represented as a pair of timestamps denoting the start and end of the interval;
- A duration is an abstract entity that is denoted by a number of milliseconds.

4 Commentary

The complete VDM specification as implemented in Overture is available from the Overture web site [8]. It is packaged with other supporting core libraries:

- ISO8601Supp – supplement to the ISO8601 module (see section 4.5);
- Ord – generic order related functions;
- Numeric - specification of numeric functionality;
- Seq - specification of functionality on sequences (lists);
- Set - specification of functionality on sets.

This section presents a commentary of some aspects of the specifications. The commentary describes a number of features of VDM and motivates their use, serving as a tutorial introduction to some aspects of VDM-SL. Refer to the Wikipedia entry [1] for a good introduction to VDM in a wider context.

4.1 Header

The module header defines:

- The name of the module;
- The modules it imports;
- The types, values and functions exported by the module.

A fragment of that header is:

```
module ISO8601
imports from Numeric all,
        from Set all,
        from Seq all
exports types struct Year
              struct Month
              struct Day
              struct Hour
              struct Minute
              struct Date
              struct Time
```

```
            Duration
      values MILLIS_PER_SECOND, SECONDS_PER_MINUTE, MINUTES_PER_HOUR, HOURS_PER_DAY: nat
             DAYS_PER_MONTH, DAYS_PER_MONTH_LEAP: map nat1 to nat1
      functions isLeap: Year +> bool
                daysInMonth: Year * Month +> nat1
                daysInYear: Year +> nat1
                dtgInRange: DTG * DTG * DTG +> bool
                inInterval: DTG * Interval +> bool
```

Only those types, values and functions defined in the module that are listed in the header are visible to other modules that import this specification. In the case of types, the **struct** qualifier indicates the internal structure of the type is also visible. In modular and object oriented programming languages the internal structure of a type is generally hidden, and manipulation always via functions or methods. However, since types in VDM are typically expressed in an abstract manner, and have associated invariants to preserve integrity of the data, exposing the internal structure of a type generally does not reveal to the client unnecessary implementation detail that could compromise data integrity.

4.2 Data Types

A key part of any specification is the data model that represents the information of interest. VDM allows the expression of data types in a high-level and abstract manner, independent of physical implementation. The notation allows the simple construction of hierarchical and inductive entities based on constituent building blocks. The following is a fragment from the data type section:

```
-- A year: 0 = 0AD (or 1BC).
Year = nat
inv year == FIRST_YEAR <= year and year <= LAST_YEAR;

-- A month in a year (January is numbered 1).
Month = nat1
inv month == month <= MONTHS_PER_YEAR;

-- A day in a month.
Day = nat1
inv day == day <= MAX_DAYS_PER_MONTH;
```

The token `--` indicates a comment; any subsequent text on the same line is ignored. VDM also supports block comments with delimiters `/*` and `*/`.

Year, month and day are each specified as an appropriate subset of natural numbers (**nat**, non-negative integers); those entities that do not allow 0 have type **nat1** (non-zero natural numbers). Invariants can be attached to types, specifying constraints that values of the type must always satisfy. In the case of `Year` the value must fall between `FIRST_YEAR` and `LAST_YEAR` (discussed in section 4.3), and in the case of `Month` and `Day` the value must not exceed `MONTHS_PER_YEAR` and `MAX_DAYS_PER_MONTH` respectively. The invariants act as a filter identifying those elements from the base type that are admitted to the type being defined.

More complex is the definition of `Date`:

```
-- A date is a triple (year/month/day).
-- Day of month must be consistent with respect to year.
Date :: year : Year
        month: Month
        day  : Day
inv mk_Date(y,m,d) == d <= daysInMonth(y,m)
ord mk_Date(y1,m1,d1) < mk_Date(y2,m2,d2) ==
      Seq`lexicographic[nat]([y1,m1,d1], [y2,m2,d2]);
```

A `Date` is a triple consisting of a year, a month and a day. The invariant states the day value must be no more than the number of days in the month, with respect to the year; `daysInMonth` is a function defined later in the specification (see section 4.4). This is a good example of how one element of a data type can depend on another element (a so-called dependent type).

The general form of an invariant is a pattern preceded by `inv` and separated from a truth valued expression by `==`. The pattern (which may simply be an identifier, as in `Year`, or a complex data element, as in `Date`) has the form of an element of the type. The expression is a constraint that all elements of the type must satisfy (and may include the free variables in the pattern).

Associated with every type definition in VDM is an implicit equality relation. In the case of simple type definitions, such as `Year`, the equality is that of the underlying type (`nat` in this case) restricted to values that satisfy the invariant. For composite types the implicit equality relation is structural equality over the fields of the record. Thus two `Date` values are equal if their corresponding `year`, `month` and `day` fields are equal. The operators for equality and inequality are `=` and `<>` respectively.[3]

An order (strict less than) relation is defined, which is the usual temporal order over dates: lexicographic order on the (year,month,day) triple. When an order relation is defined for a type, the infix operators `<`, `<=`, `>` and `>=` are available for use in subsequent specifications involving values of that type. The operators `<=` and `>=` are dependent on the equality relation over the type.

The general form of an order clause is

```
ord pat1 < pat2 == expression
```

where `pat1` and `pat2` are patterns denoting values of the type being compared, and `expression` is truth valued expression that is true exactly when `pat1` is strictly less than `pat2`. In the `Date` example the order relation depends on the `lexicographic` function in the `Seq` module (see section 4.6).

Even in this fairly simple case the value of an expressive specification language is evident; the constraints on a type are highlighted by being associated directly with the type, as is the order relation. Only valid dates can be constructed; dates such as 29/02/2015 are excluded explicitly, because the expression

```
29 <= daysInMonth(2015,2)
```

is false. Furthermore, due to the order clause, the ordering of elements is explicit and co-located with the definition of the type.

A more traditional specification would define the hierarchical data that models information of interest, then add the relevant constraints to a list of business rules, presented separately from the model and expressed in an informal manner. An order relation might be defined for the data, but in a function separate from the type definition.

To denote a date in VDM with this definition, we require an expression of the form

```
mk_Date(2016,2,29)
```

where each of the individual values satisfies its constraint (in particular, 29 is a valid day of February in year 2016). The values correspond, respectively, to the three fields `year`, `month` and `day`, and must be listed in the same order as they appear in the definition. The expression

```
mk_Date(2015,2,29)
```

on the other hand, while syntactically correct, does not satisfy the invariant. The environment for simulation and testing of a specification automatically checks all invariants to ensure the integrity of the model is maintained.

The field names can be used to reference the value in an element of the type. If we have

```
leap_day = mk_Date(2016,2,29)
```

then

```
leap_day.year  = 2016
leap_day.month = 2
leap_day.day   = 29
```

[3] The implicit equality relation can be over-ridden by an explicitly defined relation, described in section ????

We can compare dates for (in)equality as in

```
mk_Date(2015,2,29) =  mk_Date(2015,2,29)
mk_Date(2015,2,29) <> mk_Date(2016,2,29)
```

or for order as in

```
mk_Date(2015,2,28) <  mk_Date(2016,2,29)
mk_Date(2015,2,28) <= mk_Date(2015,2,28)
mk_Date(1962,2,20) >  mk_Date(1961,2,12)
mk_Date(1912,1,17) >= mk_Date(1911,12,14)
```

`Time` is defined similarly to `Date`; the finest granularity that can be represented in the model is one millisecond.

```
-- An hour in a day.
Hour = nat
inv hour == hour < HOURS_PER_DAY;

-- A minute in an hour.
Minute = nat
inv minute == minute < MINUTES_PER_HOUR;

-- A second in a minute.
Second = nat
inv second == second < SECONDS_PER_MINUTE;

-- A millisecond in a second.
Millisecond = nat
inv milli == milli < MILLIS_PER_SECOND;

-- A Time consists of four elements (hours/minutes/seconds/milliseconds).
Time :: hour  : Hour
        minute: Minute
        second: Second
        milli : Millisecond
ord mk_Time(h1,m1,s1,l1) < mk_Time(h2,m2,s2,l2) ==
      Seq`lexicographic[nat]([h1,m1,s1,l1], [h2,m2,s2,l2]);
```

No invariant is defined on `Time` because, unlike `Date`, any valid values of the field elements of `Time`, when combined, result in a valid element of `Time`. Like `Date`, an order relation is defined for `Time`, being the usual temporal ordering and again employing the `lexicographic` function in the `Seq` module.

The situation becomes a little more complex when we introduce time zones. The same instant in time can be represented by different values in different time zones. For example, the following values all represent the same point in time:

19:18:04-01:00

20:18:04Z (i.e. UTC)

21:48:04+01:30

Equality over times with respect to time zones much take account of any time zone offset.

```
-- A time in a zone consists of a time and a time zone offset.
TimeInZone :: time  : Time
              offset: Offset
eq t1 = t2 == normaliseTime(t1).#1 = normaliseTime(t2).#1
ord t1 < t2 == normaliseTime(t1).#1 < normaliseTime(t2).#1;

-- The timezone offset
Offset :: delta: Duration
          pm   : PlusOrMinus
inv os == -- Offset must be less than one day and an integral number of minutes.
          os.delta < ONE_DAY and durModMinutes(os.delta) = NO_DURATION
eq mk_Offset(d1,o1) = mk_Offset(d2,o2) ==
     d1 = d2 and o1 = o2 or d1 = NO_DURATION and d2 = NO_DURATION;

-- The direction of a time zone offset.
```

```
PlusOrMinus = <PLUS> | <MINUS>;
```

A time with respect to a zone consists of the time (in the zone) plus the offset with respect to UTC. The offset consists of a duration and a direction of deviation (ahead of or behind UTC). The implicit structural equality provided by VDM would treat the three earlier time examples as distinct. The same absolute point in time, though expressed differently in different time zones, must be identified in the model to capture the intended semantics. As noted earlier, the implicit VDM equality relation is structural equality. When such an equality is inadequate, an explicit relation can be defined that is employed in preference to the implicit equality.

The function `normaliseTime` (section 4.4) takes a time with an offset (type `TimeInZone`) and converts it to the equivalent UTC value, i.e. time without offset (type `Time`). We can then employ the implicit structural equality on time to compare the values. Similarly, the order clause defines an order relation on `TimeInZone` by comparing for order the normalised values using the order relation on `Time`.

The definition of `PlusOrMinus` demonstrates VDM union types, that in this case is equivalent to an enumeration type in programming languages. The values, bounded by angle brackets, are elements of the VDM Quote type (constants that can be compared for equality but are otherwise indivisible). However, VDM allows the union of arbitrary types so is far more general than simple enumerations.

A combined date and time, referred to as a Date/Time Group (DTG), is defined as:

```
-- A DTG (date/time group) is a combined date and time.
DTG :: date: Date
       time: Time
ord mk_DTG(d1,t1) < mk_DTG(d2,t2) == d1 < d2 or d1 = d2 and t1 < t2;
```

An explicit order relation is defined, based on the order relations of the constituent date and time fields. No equality is defined, hence two DTGs are equal if they have the same date and time, as per the implicit default equality relation.

A DTG with respect to a time zone is a combined date and time with respect to a time zone offset:

```
-- A date and time with time zone offset.
DTGInZone :: date: Date
             time: TimeInZone
inv mk_DTGInZone(date,time) ==
      let changeDay = normaliseTime(time).#2
      in -- Adjusted time must not be earlier than 0000-01-01T00:00:00Z.
         not (date = FIRST_DATE and changeDay = <PLUS>) and
         -- Adjusted time must not be later than 9999-12-31T23:59:59,999Z
         not (date = LAST_DATE and changeDay = <MINUS>)
eq dtg1 = dtg2 == normalise(dtg1) = normalise(dtg2)
ord dtg1 < dtg2 == normalise(dtg1) < normalise(dtg2);
```

The model has a minimum and maximum representable date. A combination of date and time with offset could result in a DTG outside the allowed bounds. When adjusted for the offset, the following values

```
0000-00-00T00:01:00+01:00
9999-12-31T23:59:59-01:00
```

are outside the allowed range of values. In the former case the offset is '+' so the time zone is one hour ahead of UTC; if we subtract the offset (one hour) from the time (one minute after the 'start of time'), the result is not valid. Similarly, a DTG value greater than the maximum allowed value could be constructed with an offset and time zone that is behind UTC. The `DTG` invariant ensures any value that can be constructed is in the UTC range

```
0000-00-00T00:00:00,000Z/9999-12-31T23:59:59,999Z
```

This again demonstrates the value of invariants. There are many functions that create and manipulate DTG values. By creating the invariant, and presenting it along with the data type to which it belongs, then:

- The constraint on the type is immediate and obvious;
- The constraint need only be stated once throughout the model;
- Constraint checking guarantees invalid values are identified immediately any attempt is made to construct a DTG, without needing to attach external 'validation code'.

As with `TimeInZone`, equality and order relation are defined for `DTGInZone` in terms of normalised values.

An interval is an ordered pair of DTGs:

```
-- An interval is a pair of DTGs representing all time instants between the start (inclusive)
-- and end (exclusive) values.
Interval :: begins: DTG
            ends  : DTG
inv ival == -- The start of the interval must be earlier than the end.
            ival.begins < ival.ends;
```

The invariant captures that the start of the interval must be earlier than the end. The start of the range is inclusive, and the end of the range is exclusive.

A duration is a period of time specified as a number of milliseconds:

```
-- Duration: a period of time in milliseconds.
Duration :: dur: nat
ord d1 < d2 == d1.dur < d2.dur;
```

The definition follows the same pattern as that of other types. In this case a duration is a number of milliseconds, so `mk_Duration(2000)` denotes 2 seconds. However, there is a difference in the way the types are exported in the module header:

```
struct Date
Duration
```

The keyword `struct` prior to `Date` indicates the fields of `Date` – `year`, `month`, `day` – and its structure are visible outside the module. The lack of a `struct` qualifier associated with `Duration` means this is not true of `Duration`. The expression `mk_Duration(2000)` is invalid outside the module. We do not want clients to need to be aware the value is in milliseconds, which could easily be overlooked. Instead, we provide a collection of functions that create durations (from seconds, minutes, hours, etc.) and that manipulate duration values.

4.3 Values

The values section of the specification defines constants that would commonly be used in specifications and computations. Simple examples are:

```
MINUTES_PER_HOUR:nat = 60;
HOURS_PER_DAY:nat = 24;
FIRST_YEAR:nat = 0;
LAST_YEAR:nat = 9999;
```

More interesting is:

```
DAYS_PER_MONTH:map Month to nat1 = {1|->31, 2|->28, 3|->31, 4|->30, 5|->31, 6|->30,
                                    7|->31, 8|->31, 9|->30, 10|->31, 11|->30, 12|->31};
```

which demonstrates (finite) maps in VDM; it associates a month with its number of days for a non-leap year. The expression `1|->31` means the domain value 1 maps to the range value 31 (January has 31 days). The expression `DAYS_PER_MONTH(4)` is then 30, the number of days in April. For a leap year we have:

```
DAYS_PER_MONTH_LEAP:map Month to nat1 = DAYS_PER_MONTH ++ {2|->29};
```

The entire map does not need to be repeated. This says take the map for a non-leap year but 2 (February) is associated with 29 instead of 28, all other months retain the same value. Maps are a powerful abstraction in VDM; they are ideal for representing a state where the domain is typically a key that uniquely identifies the corresponding state value, and it maps to that value. (Example: the key is an identifier of a flight, and it maps to the full details of that flight.)

The range (`rng`) of a map is the set of all values that are a target of the map. Since the range of `DAYS_PER_MONTH` is the set of all numbers of days in a month, then the maximum of that set is the maximum number of days in any month.

```
MAX_DAYS_PER_MONTH:nat1 = Set`max[nat1](rng DAYS_PER_MONTH);
```

`` Set`max `` is a reference to the `max` function in module `Set` [8]. The presence of `nat1` in the function call indicates `max` is a polymorphic function (discussed in section 4.6).

4.4 Functions

This section selects some of the functions defined in the module and provides commentary.

The first function determines if a year is a leap year.

```
-- Is a year a leap year?
isLeap: Year +> bool
isLeap(year) == year rem 4 = 0 and (year rem 100 = 0 => year rem 400 = 0);
```

The general definition pattern for explicit functions is demonstrated. The first line names and declares the type of a function; `isLeap` is a function that takes a year as argument and returns a truth value. The subsequent lines specify properties of the function.

An explicit function definition is characterised by an expression that defines how the result of the function is computed. VDM also supports implicit function definitions, where the pre and post conditions are stated, but no information is provided concerning how the function value may be computed. We employ explicit function definitions since a goal is to allow model simulation.

The symbol `+>` indicates the function is total: that is, for any value in the domain of the function that satisfies its pre-conditions, the function is guaranteed to terminate with a value in the range.

The function `daysInMonth` specifies the number of days in a month with respect to a year. Recall it is used to define the invariant of type `Date`.

```
-- The number of days in a month with respect to a year.
daysInMonth: Year * Month +> nat1
daysInMonth(year, month) ==
  if isLeap(year) then DAYS_PER_MONTH_LEAP(month) else DAYS_PER_MONTH(month);
```

The values section of the specification already provides the number of days in each month (section 4.3), so the function is specified in terms of those maps.

The number of days in a year is the sum of the days in the months of that year.

```
-- The number of days in a year.
daysInYear: Year +> nat1
daysInYear(year) == Seq`sum([daysInMonth(year,m) | m in set {1,...,MONTHS_PER_YEAR}]);
```

`` Seq`sum `` is a reference to the `sum` function in module `Seq` [8].

The specification says: create a sequence whose items are the number of days in each month of the specified year, then calculate the sum of the numbers in that sequence, giving the number of days in the year. Note we must create a sequence rather than a set; if the latter was used then different months with the same number of days would be ignored.

The function `overlap` defines when two intervals overlap. The definition is simple though not immediately obvious:

(begins$_1$,ends$_1$) overlaps (begins$_2$,ends$_2$) if:

$begins_2 < ends_1 \wedge begins_1 < ends_2$

```
-- Do two intervals overlap?
overlap: Interval * Interval +> bool
overlap(i1, i2) == i2.begins < i1.ends and i1.begins < i2.ends;
--post RESULT = exists d:DTG & inInterval(d, i1) and inInterval(d, i2);
```

Note the commented out post condition which states the constraint more directly: two intervals overlap if there is an instant in time common to both. Type `DTG` has many elements, though it is finite. VDM interpreters cannot in general decide if a given type is finite or not. Consequently, types such as `DTG` are effectively treated as infinite, and the constraint is undecidable.

`within` defines when one interval falls wholly within another:

$(begins_1, ends_1)$ within $(begins_2, ends_2)$ if:

$begins_2 \leq begins_1 \wedge ends_1 \leq ends_2$

```
 -- Does one interval fall wholly within another interval?
within: Interval * Interval +> bool
within(i1, i2) == i2.begins <= i1.begins and i1.ends <= i2.ends;
--post RESULT = forall d:DTG & inInterval(d, i1) => inInterval(d, i2);
```

The post condition states any instant in the first interval is also in the second interval, and again is commented out due to quantification over a type that cannot be confirmed automatically as finite.

Many of the remaining functions are concerned with the relationship between DTG values and durations; that relationship is one of the most fundamental parts of the specification. Dates are based on the (proleptic) Gregorian calendar. Any date or DTG can be expressed as a duration commencing at time 00:00:00.000 on 1st January year 0. Many of the functions are specified in terms of the duration from 'the start of time'. This representation is computationally expensive when running model simulations.

The next two definitions capture a common function: add (subtract) a duration to (from) a DTG.

```
-- Increase a DTG by a duration.
add: DTG * Duration +> DTG
add(dtg, dur) == durToDTG(durAdd(durFromDTG(dtg), dur))
post subtract(RESULT, dur) = dtg;

-- Decrease a DTG by a duration.
subtract: DTG * Duration +> DTG
subtract(dtg, dur) == durToDTG(durDiff(durFromDTG(dtg), dur))
pre dur <= durFromDTG(dtg);
--post add(RESULT,dur) = dtg;
```

In both cases the supplied DTG is converted to a duration, to (from) which the supplied duration is added (subtracted), and the result converted back to a DTG.

The post condition of the `add` function states it is an inverse of the `subtract` function. The reserved VDM identifier `RESULT` denotes the value returned by the function, and may be referred to in the post-condition (but not the pre-condition). The equivalent post-condition is commented out in `subtract`. Running a simulation of the specification would cause an infinite loop if both post-conditions were made explicit: verifying the post-condition of `add` involves calling `subtract`, but verifying the post-condition of `subtract` involves calling `add`, but verifying the post-condition of `add` …

Note the pre-condition of `subtract`: it is only possible to subtract a duration from a DTG if the duration corresponding to that DTG is at least as big as the duration to be subtracted.

Function `diff` computes the duration between two DTGs.

```
-- The duration between two DTGs.
diff: DTG * DTG +> Duration
diff(dtg1, dtg2) == durDiff(durFromDTG(dtg1), durFromDTG(dtg2))
post (dtg1 <= dtg2 => add(dtg1,RESULT) = dtg2) and
     (dtg2 <= dtg1 => add(dtg2,RESULT) = dtg1);
```

The duration between two DTGs is expressed in terms of the difference between their corresponding durations. The post-condition states adding the resulting duration to the smaller of the given DTGs equals the larger of the given DTGs.

`durToMinutes` computes the number of whole minutes in a duration, while `durFromMinutes` computes the duration of a given number of minutes.

```
-- The whole number of minutes in a duration.
durToMinutes: Duration +> nat
durToMinutes(d) == durToSeconds(d) div SECONDS_PER_MINUTE
post durFromMinutes(RESULT) <= d and d < durFromMinutes(RESULT+1);

-- The duration of a number of minutes.
durFromMinutes: nat +> Duration
durFromMinutes(mn) == durFromSeconds(mn*SECONDS_PER_MINUTE);
--post durToMinutes(RESULT) = mn;
```

A duration need not be an exact number of minutes, so the post condition of `durToMinutes` states the duration of the resulting minutes must be no greater than the supplied duration, and the supplied duration must be less than the duration of one greater than the resulting minutes (expressed another way, the remainder when dividing the duration by one minute is a duration that is less than a minute).

In contrast, the post condition of `durFromMinutes` states if the resulting duration is converted to minutes we get the originally supplied minutes (though it is commented out to avoid infinite computation). The two functions are closely related, but not inverses.

Function `durFromYear` computes the duration of a year:

```
-- The duration of a year.
durFromYear: Year +> Duration
durFromYear(year) == durFromDays(daysInYear(year));
```

It builds on other functions by first determining the number of days in the year, then computing the duration of that many days.

The following two functions convert between a DTG and a duration.

```
-- The DTG corresponding to a duration.
durToDTG: Duration +> DTG
durToDTG(dur) == let dy = durFromDays(durToDays(dur))
                 in mk_DTG(durToDate(dy),durToTime(durDiff(dur,dy)))
post durFromDTG(RESULT) = dur;

-- The duration of a DTG (with respect to the start of time).
durFromDTG: DTG +> Duration
durFromDTG(dtg) == durAdd(durFromDate(dtg.date),durFromTime(dtg.time));
--post durToDTG(RESULT) = dtg;
```

`durToDTG` and `durFromDTG` are inverses which is captured in the post-conditions (for the same reason as before the post-condition of `durFromDTG` is commented out).

Function `minDTG` determines the smallest DTG in a set. It demonstrates the use of quantification.

```
-- The maximum DTG in a set.
maxDTG: set1 of DTG +> DTG
maxDTG(dtgs) ==  Set`max[DTG](dtgs)
post RESULT = iota d in set dtgs & forall x in set dtgs & d >= x;
```

The argument type is `set1` indicating the function only accepts non-empty sets (there can be no minimum element in an empty set). The function states: the result is the unique element in the set that is at least as small as any other element in the set. The `iota` quantifier denotes the unique item for which the subsequent statement is true (in this case, the unique smallest element). The `forall` quantifier is satisfied if the condition holds for all bound values (in this case, every element in set `dtgs`).

Quantifiers are a good way to capture specifications at an abstract level as they allow general statements to be made over potentially large sets. Quantifiers can also range over types, but if the type is infinite (or cannot be confirmed finite) the quantified statement is not decidable.

Functions that are useful in some contexts are to determine, given a date, the next valid date with the same day of month, or the next valid date with a nominated day of month. The functions are complicated by the fact that different months have different numbers of days, and in the case of February the number of days is not consistent across years. Examples of the next date are

2015-01-29	→	2015-03-29
2016-01-29	→	2016-02-29

Examples of the next date for a nominated day are

2015-01-29 / 12	→	2015-02-12
2016-01-12 / 17	→	2016-01-17

The following specifies these functions, named `nextDateForYM` and `nextDateForDay` respectively. First, we present an auxiliary function `nextYMDForDay`:

```
-- Given a year/month/day, find the closest next date on which the day of month is the same.
nextYMDForDay: Year * Month * Day * Day +> Date
nextYMDForDay(yy, mm, dd, day) ==
  let nextm = if mm = MONTHS_PER_YEAR then 1 else mm+1,
      nexty = if mm = MONTHS_PER_YEAR then yy+1 else yy
  in if dd < day and day <= daysInMonth(yy, mm)
     then mk_Date(yy, mm, day)
     elseif day = 1
     then mk_Date(nexty, nextm, day)
     else nextYMDForDay(nexty, nextm, 1, day)
pre dd <= daysInMonth(yy, mm)
--post RESULT = minDate({ date | date:Date & date > mk_Date(yy, mm, dd) and
--                                           date.day = day })
measure m_nextYMDForDay;

-- The measure function for nextYMDForDay
m_nextYMDForDay: Year * Month * Day * Day +> nat
m_nextYMDForDay(y,m,-,-) == ((LAST_YEAR+1)*MONTHS_PER_YEAR) - (y*MONTHS_PER_YEAR + m);
```

Given are a year, month and day value (a date expressed as its individual elements), and a target day of month. The goal is to determine the next date after the supplied date with the target day of month.

This function demonstrate the advantage of declarative specifications over operational specifications. The (declarative, commented out) post condition very clearly and explicitly captures the semantics: the minimum of all dates that are greater than the supplied date and have the same day of month. However, the binding ranges over all values of type DTG, so is not decidable, preventing its use due to the requirement of model simulation. The (operational) body of the function is far more complex, involving conditional tests and a recursive call.

Note the specification of a `measure` function. `nextYMDForDay` is a recursive function, thus execution may lead to infinite computation. A measure function takes the same arguments as the function being specified, and returns a natural number (non-negative integer) as result. The value of the measure function should be smaller in the recursive call than in the main function call. If the measure function reduces on each call, and the result can never be less than 0, then the function is guaranteed to terminate. This is another condition that can be checked during simulation. If the measure function is not less on the recursive call, the simulation highlights an error in the specification.

With auxiliary function `nextYMDForDay`, we can now specify `nextDateForYM` and `nextDateForDay`

```
-- Given a date, find the next date on which the day of month is the same.
```

```
nextDateForYM: Date +> Date
nextDateForYM(date) == nextDateForDay(date, date.day);
--post RESULT = minDate({ dt | dt:Date & dt > date and dt.day = date.day});

-- Given a date and specific day, find the closest next date on which the day of month is the
-- same as the specified day.
nextDateForDay: Date * Day +> Date
nextDateForDay(date, day) == nextYMDForDay(date.year, date.month, date.day, day)
pre day <= MAX_DAYS_PER_MONTH;
--post RESULT = minDate({ dt | dt:Date & dt > date and dt.day = day });
```

A collection of functions format elements from the model as strings per ISO 8601. Two of those functions are

```
-- Format a date and time as per ISO 8601.
formatDTG: DTG +> seq of char
formatDTG(dtg) == formatDate(dtg.date) ^ "T" ^ formatTime(dtg.time);

-- Format a date as per ISO 8601.
formatDate: Date +> seq of char
formatDate(mk_Date(y,m,d)) ==
  Numeric`zeroPad(y,4) ^ "-" ^ Numeric`zeroPad(m,2) ^ "-" ^ Numeric`zeroPad(d,2);
```

A formatted DTG is the string consisting of the formatted date followed by the separator ‘T’ followed by the formatted time. A formatted date consists of the formatted (zero padded) year, month and days values, connected by the separator ‘-’.

A DTG with time zone offset is normalised (transformed to UTC) by normalising its time, and if required adding (subtracting) one day to (from) the date.

```
-- Normalise a DTG value such that it is expressed without time zone offset.
-- Applying the offset may result in a change of date.
-- Example: 2001-01-01T01:00+02:00 becomes 2000-12-31T23:00Z.
normalise: DTGInZone +> DTG
normalise(dtg) == let mk_(ntime,pm) = normaliseTime(dtg.time),
                      baseDtg = mk_DTG(dtg.date, ntime)
                  in cases pm:
                       <PLUS>  -> subtract(baseDtg, ONE_DAY),
                       <MINUS> -> add(baseDtg, ONE_DAY)
                     end;
```

The time is normalised, and combined with the date. There are three possibilities:

1. Application of the offset does not change the date;
2. Application of the offset changes the date and the time zone is ahead of UTC – the date is adjusted backwards by a day;
3. Application of the offset changes the date and the time zone is behind UTC – the date is adjusted forwards by a day.

The `normaliseTime` function applies an offset to a time to give the corresponding UTC time. The calculated time, if necessary, wraps across a day boundary. The result indicates the direction of wrapping.

```
-- Normalise a time value with a time zone offset to the UTC value, wrapping across the day
-- boundary. Return an indication if the normalisation pushes the time to a different day.
-- Example: 01:00+02:00 (01:00, two hours ahead of UTC) becomes (23:00Z,<PLUS>) indicating
-- the original time with offset is on the day after the UTC time.
-- Similarly, 23:30-01:15 becomes (00:45,<MINUS>).
normaliseTime: TimeInZone +> Time * [PlusOrMinus]
normaliseTime(time) ==
  let utcTimeDur = durFromTime(time.time)
  in cases time.offset:
       mk_Offset(offset,<PLUS>)
         -> -- Zone offset ahead of UTC
            if offset <= utcTimeDur
            then -- No day change
                 mk_(durToTime(durSubtract(utcTimeDur,offset)), nil)
            else -- UTC time one day earlier
```

```
                 mk_(durToTime(durSubtract(durAdd(utcTimeDur,ONE_DAY),offset)),<PLUS>),
       mk_Offset(offset,<MINUS>)
         -> -- Zone offset behind UTC
            let adjusted = durAdd(utcTimeDur,offset)
            in if adjusted < ONE_DAY
               then -- No day change
                    mk_(durToTime(adjusted),nil)
               else -- UTC time one day later
                    mk_(durToTime(durSubtract(adjusted,ONE_DAY)),<MINUS>)
     end;
```

This function demonstrates the use of pairs in VDM (a special instance of an n-tuple where n=2). The function returns an element of type `UTC*[PlusOrMinus]`; a pair whose first element is of type `UTC` and second element is of type `[PlusOrMinus]`. Given elements `utc` and `pm` of those respective types, the expression

```
        mk_(utc,pm)
```

denotes the pair element. The #n operator is used to extract the individual elements from the pair, so

```
        mk_(utc,pm).#1 = utc
        mk_(utc,pm).#2 = pm
```

VDM supports tuples of arbitrary length. For example, a triple might be `nat*char*bool`. The definition of function `normalise` demonstrates the use of tuples when pattern matching.

In the case that normalisation does not require the date to be adjusted, the second element of the pair is not required. Consequently, the return type of the function indicates the `PlusOrMinus` element is optional by enclosing it in brackets: `[PlusOrMinus]`. When creating an expression whose type is optional, the absence of a value is expressed in VDM with the constant `nil`.

4.5 ISO 8601 Supplement

The ISO 8601 module models dates, times, durations and intervals, and specifies a wide variety of functions that are of general interest when manipulating those model elements. At times it is necessary to process dates and times where full details are not available. For example, in some situations a date and time may be specified by a day of month, hour and minute (DDHHMM), with context determining the year and month (typically the context is the time at which the DDHHMM value is processed).

The ISO8601Supp module models various types that represent partial date/time values. The associated functions are primarily concerned with: given a partial date/time value and a reference time, determine the full date/time details with respect to the reference time. For example, say we have the DDHHMM value 010130 (0130 on the first day of some month), the reference time is 2000-12-31T23:00Z, and we need to determine the absolute date/time value nearest the reference time with day of month 1 and time 0130: the value sought is 2001-01-01T01:30Z. Further examples:

DDHHMM	Context DTG	Derived DTG
291200	2001-03-01T23:00Z	2001-03-29T12:00Z
202018	1969-07-16T13:32Z	1969-07-20T20:18Z
111720	1972-12-11T19:54Z	1972-12-11T17:20Z

The definitions `nearestToDHM` and `vicinityDHM` capture this function.

```
        -- The DTG nearest to a given DDHHMM with respect to a reference DTG.
        -- If there are two candidates equally distant from the reference DTG, choose the earliest.
        nearestToDHM: DDHHMM * ISO8601`DTG +> ISO8601`DTG
        nearestToDHM(dhm, ref) ==
          let neighbourhood = vicinityDHM(ref,dhm)
          in ISO8601`minDTG({dtg | dtg in set neighbourhood
                                 & forall d in set neighbourhood
                                        & ISO8601`diff(ref,dtg) <= ISO8601`diff(ref,d) })
        post isConsistentDHM(dhm, RESULT) and ISO8601`finestGranularity(RESULT, ISO8601`ONE_MINUTE);
        --let candidates = { dtg | dtg:ISO8601`DTG
        --                         & isConsistentDHM(dhm, dtg) and
        --                           ISO8601`finestGranularity(dtg, ISO8601`ONE_MINUTE) }
        --in ISO8601`minDTG({dtg | dtg in set candidates
        --                         & forall c in set candidates
        --                                & ISO8601`diff(ref,dtg) <= ISO8601`diff(ref,c)});

        -- The set of DTGs in the temporal vicinity of a reference DTG whose day/hour/minute match a
        -- given DDHHMM and seconds/milliseconds are 0. Guaranteed to include the closest DTG earlier
        -- than the reference DTG and the closest later.
        vicinityDHM: ISO8601`DTG * DDHHMM +> set of ISO8601`DTG
        vicinityDHM(mk_ISO8601`DTG(date,-), dhm) ==
          let previous = ISO8601`previousDateForDay(date, dhm.dd),
              next = ISO8601`nextDateForDay(date, dhm.dd),
              time = mk_ISO8601`Time(dhm.hh, dhm.mm, 0, 0)
          in { mk_ISO8601`DTG(previous, time), mk_ISO8601`DTG(date, time), mk_ISO8601`DTG(next, time) }
        post forall d in set RESULT & isConsistentDHM(dhm, d) and d.time.second = 0 and d.time.milli = 0;
```

Function `nearestToDHM` determines the nearest DTG value (with the required day, hour and minute) to the reference from a set of DTGs in the neighbourhood of the reference. `vicinityDHM` computes that neighbourhood with the guarantee that it includes at minimum the date/times immediately before and immediately after the reference that satisfy the constraint.

The commented out alternative definition of `nearestToDHM` specifies the function more directly without the need for the supporting function `vicinityDHM`, but is not decidable since the comprehension ranges over all DTGs.

4.6 The Set and Seq Modules

The primary purpose of this paper is to provide a commentary on the date/time specification. However, that specification employs other low level specifications for manipulating sets, sequences, and numeric values. In this section we provide some commentary on the Set and Seq modules.

In these cases, there are no type definitions. We are simply specifying general purpose functions over sets and sequences to supplement the primitive functions provided by VDM.

The cross product of two sets is the set of all pairs whose first element is drawn from one set and second element from the other.

```
-- The cross product of two sets.
xProduct[@a,@b]: set of @a * set of @b +> set of (@a * @b)
xProduct(s,t) == { mk_(x,y) | x in set s, y in set t }
post card RESULT = card s * card t;
```

The specification demonstrates the use of comprehensions which in some situations can be used to capture the meaning in a concise manner. In this case it says create the set of all pairs whose first element is drawn from `s` and second element is drawn from `t` (also referred to as the Cartesian product).

The post condition states the number of items in the resulting set is the product of the numbers of items in the argument sets.

The specification also demonstrates polymorphism in VDM. The function takes as arguments a set of one kind of thing (denoted by `@a`), and a set of another kind of thing (denoted by `@b`), and delivers a set of pairs. The first element of each pair is a `@a` and the second element of each pair is a `@b`. The cross product function does not care what types `@a` and `@b` actually denote, so the function is polymorphic.

When a polymorphic function is invoked in VDM, it is necessary to specify explicitly the actual type arguments that replace the type variables in the definition (type instantiation). For example, given `x = {1,2}: set of nat` and `y = {'a', 'b'}: set of char` then the call to `xProduct` must be

```
        xProduct[nat,char](x,y)
```

The function `toSeq` creates a sequence whose elements are those of a supplied set.

```
        -- The sequence whose elements are those of a specified set, with no duplicates.
        -- No order is guaranteed in the resulting sequence.
        toSeq[@a]: set of @a +> seq of @a
        toSeq(s) == cases s:
                      {} ->        [],
                      {x} ->       [x],
                      t union u -> toSeq[@a](t) ^ toSeq[@a](u)
                    end
        post len RESULT = card s and elems RESULT = s
        measure size;
        /*
          A simpler definition would be
            toSeq(s) == [ x | x in set s ]
          This would however assume an order relation on the argument type @a.
        */
```

There is no guarantee of the order of the elements in the result set. The only guarantee is that the sequence contains exactly the same elements as the set. Note however the commented out definition, which demonstrates polymorphism combined with order relations. Sequence comprehension over sets in VDM is restricted types that admit an order relation. Consequently, the commented out definition is only computable if the argument set has an order relation; if not, a runtime error is thrown. There is no way in VDM to attach a pre-condition to the function definition that states "type @a has a defined order relation". We choose to maintain full generality and allow arbitrary types, at the cost of a more complex specification.

The function `xform` in the `Seq` module takes a sequence and a function, and delivers the result of applying the function to each element in the sequence, i.e. it transforms the sequence. This is generally called the 'map' function but we avoid that name here due to the primitive VDM type of the same name.

```
        -- Apply a function to all elements of a sequence.
        xform[@a,@b]: (@a+>@b) * seq of @a +> seq of @b
        xform(f,s) == [ f(x) | x in seq s ]
        post len RESULT = len s and (forall i in set inds s & RESULT(i) = f(s(i)));
```

Like `xProduct`, `xform` is a polymorphic function with two type arguments; all we care is that the function transforms a value of one type to another type, and that the supplied sequence contains values of the first type, not what the actual types are.

The definition of `xform` further demonstrates the value of comprehensions, this time over sequences, capturing directly that the function argument `f` is being applied to each item in the sequence `s`.

The elements in a sequence are indexed commencing at 1, so `s(i)` is the i^{th} element of sequence `s`. The expression `inds s` denotes the set of all indexes of `s`, so if `s` has *n* elements, then `inds s` is the set {1,2,…,n}.

The function `fold1` in the `Seq` module takes a non-empty sequence and a binary function, and delivers the result of iterating the function over the elements in the sequence.

```
        -- Fold (iterate, accumulate, reduce) a binary function over a non-empty sequence.
        -- The function is assumed to be associative.
        fold1[@a]: (@a * @a +> @a) * seq1 of @a +> @a
        fold1(f, s) == cases s:
                         [e]   -> e,
                         s1^s2 -> f(fold1[@a](f,s1), fold1[@a](f,s2))
                       end
```

```
--pre forall x,y,z:@a & f(x,f(y,z)) = f(f(x,y),z)
measure size1;

size1[@a]: (@a * @a +> @a) * seq1 of @a +> nat
size1(-, s) == len s;
```

For example, folding the addition operator over a numeric sequence, such as

```
fold1[nat](add, [1,2,3,4])
```

is equivalent to

```
add(1,add(2,add(3,4)))
```

or

```
1 + 2 + 3 + 4
```

The function `add` is defined in the Numeric module [8]. When a polymorphic function (`fold1`) is called, the instance of the type parameter (`nat`) must be stated explicitly between the function name and its arguments.

The specification demonstrates pattern matching on sequences in VDM. Case analysis is performed on the argument `s` for the singleton sequence, and a sequence of two or more items[4]. Most interesting is the latter case, where the pattern is `s1^s2`. This says the matched sequence (`s`) is split into two sub-sequences which, when concatenated, give the original sequence. The only guarantee is the sub-sequences contain at least one item each. The specification cannot make any assumptions about how the sequence is sub-divided when matching. If it was required, for example, to specifically break it into the first item and the remainder, the pattern would be `[e]^s2`.

The pre-condition states the function argument must be associative. Since the function is polymorphic, any type could be instantiated, including infinite types. Consequently, the pre-condition is commented out to allow simulation of the model.

Another example of a polymorphic function that depends on an order relation is the `ascending` function in module `Seq`:

```
-- Is a sequence presented in ascending order?
ascending[@a]: seq of @a +> bool
ascending(s) == forall i in set {1,...,len s - 1} & s(i) <= s(i+1)
--pre Type argument @a admits an order relation.
post RESULT <=> descending[@a](reverse(s));
```

The definition states that for any two consecutive elements in the sequence, the first must be no greater than the second, hence the sequence is presented in ascending order. The function is polymorphic in the element type of the sequence, and those elements are compared for order (via the `<=` relation). However, being a polymorphic function there is no guarantee the type of elements in an application of ascending have an order relation. Furthermore, there is no way to formalise the order requirement as a pre-condition in the specification. It is incumbent that any use of the specification guarantees the argument type admits an order relation. Model checking would raise a runtime error if this condition was contravened.

5 Definitions

API	Application Programming Interface
DTG	Date-Time Group; a date in the Gregorian calendar and time with optional offset from UTC.

[4] The sequence argument of `fold1` has type `seq1 of @a`, the type of non-empty sequences, so the function would fail if invoked with the empty sequence. Consequently, the case analysis does not need to account for the empty sequence.

IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
RFC	Request For Comments
UTC	Co-ordinated Universal Time
VDM	Vienna Definition Method
VDM-SL	VDM Specification Language

6 References

[1] ISO 8601:2004 Representation of dates and times, at http://www.iso.org/iso/catalogue_detail?csnumber=40874

[2] IETF RFC 3339 Date and Time on the Internet: Timestamps, at https://www.ietf.org/rfc/rfc3339.txt

[3] Vienna Development Method, at https://en.wikipedia.org/wiki/Vienna_Development_Method

[4] VDM-10 Language Manual, TR-001

[5] Modelling Systems – Practical Tools and Techniques in Software Development, Second Edition, J. Fitzgerald and P. G. Larsen, Cambridge University Press, 2009

[6] Combinatorial Testing for VDM, in: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods, 2010, pp. 278-285. P. G. Larsen, K. Lausdahl and N. Battle.

[7] Overture Tool, at http://overturetool.org/

[8] ISO 8601, at http://overturetool.org/download/examples/VDMSL/