Semantic-Enabled Transformation Framework for Time Series

Robert Barta[1] and Thomas Bleier[2]

[1] rho information systems
rho@devc.at
[2] ARC Austrian Research Centers GmbH
thomas.bleier@arcs.ac.at

Abstract. Conventional processing of time series is done along a split horizon: on the one hand it has to handle quantitative data organized along the time axis, on the other hand meta data capturing circumstantial facts about the values, or about the time sequence as a whole. We propose to use an integrative approach using a domain specific language for the transformation of time sequences, covering arithmetic, temporal but also semantic aspects of such computations. In that we leverage Topic Maps as one existing semantic technology.

1 Introduction

Time series over sensor observations are the predominant data structure in many life science and geospatial applications. Whereever processes are observed, observation data is sampled and recorded together with environmental information when, how and under which circumstances a measurement is taken (or computation is performed).

Despite the simple concept of a chronologically ordered sequence of values, real-world time series can have a substantial variety, both in terms of the quantitative values involved, the number of values themselves, but also to which extent meta information is used. Processing needs also vary wildly, although there are some typical computation patterns:

- *Aggregation* is the condensation of quantitative information into more abstract, qualitative values. A gliding mean over SO2 values of the past half hour, for example, will aggregate over the time axis. The mean values do not simply stand for themselves; they have to incorporate the knowledge when and over which time frame they have been computed. Furthermore, aggregation methods will depend—among many other things—on the phenomenon itself or the procedure how the sensor came about the value in the first place.
  But aggregation can also be over geographical areas, or space in general, and also over semantic axes: If CO2 is known to be an instance of the class *greenhouse gas*, then an accumulated value of all greenhouse gases can be automatically created.
- *Disaggregation* is the inverse process to estimate quantitative information along time, space and semantic axes. One example are CO2 emissions of animals in a given area. If the total sum is known and so is the ratio of animals and an average emission per cow, sheep and camel, then the emissions of all sheep in the area can be factored out. This can be automatic if cow, sheep and camels are subclasses of animals in the background ontology.

It are these patterns which sit at the core of modern environmental monitoring and forecasting systems. For auditing, but also increasingly legal reasons more and more focus shifts from the data to the *meta data*, so that whole processing chains have to reliably keep track on that, how and why particular time series data is used for a particular decision. Recent EU regulations (INSPIRE) also mandate that environmental information is properly passed on between countries and public agencies and citizens. This must include meta information.

The challenge is that quantitative, temporal, spatial and semantic information has to be brought into one consolidated computational model. In this work we propose such a domain specific language, one which operates on time sequences. It should enable to specify transformations, not only based on the numerical data, but also any semantic data available, be that inside the time sequence or within an underlying semantic network. For practical reasons the language should be compatible with both predominant semantic technology stacks, RDF [12] and Topic Maps [7], and it should also degrade gracefully in the absence of any semantic network.

Our work is to be understood in the context of SWE [13] and is specifically targeted at enhancing sensor observation services (SOS [11]). The latter are not only instrumental to expose sensor measurements via a web service; also time series derived from original sensor values can be offered by specialized SOSes (virtual sensors). The necessary meta information to describe virtual sensors in SensorML [2] is directly linked to the processing model we propose.

Our contribution is

- to choose and customize a semantic framework to seamlessly host temporal information,
- to define a time sequence transformation language, Formula 3 (F3), and
- to demonstrate how underlying ontological information can be leveraged to perform informed semantic transformations.

In that we will proceed as follows. First we focus on Topic Maps as semantic technology and recapitulate the most important concepts together with a textual notation of our making. Then we turn to the query language (subsection 2.2). Using this as baseline, we defend our choice over the more main-stream RDF/S framework later in section 6.1 (Related Work). Why the TM data model is still suboptimal for our purposes and how it can be extended we cover in subsection 2.3.

The following larger section covers the language Formula 3 (F3). In order to keep this presentation compact, we traded formal grammar rules with canonical examples from the sensor web domain. We only hint at the fact that behind F3 a (process) algebra defines the formal semantics. The extension framework (new data types, operators, kernel functions, etc.) will be covered elsewhere. Section 5 finally demonstrates how F3 meta data management can be made *semantic* with the use of an underlying semantic network and a path expression language adopted from TMQL (query language for TMs).

2 Temporal Topic Maps

Topic Maps (TM, the ISO standard defines this in plural) is a knowledge representation framework quite comparable to the more main-stream RDF/S technology stack. While in the latter all information is couched in form of triples (subject, predicate, object), basic concepts in TM are designed in a more high-level, anthropocentric way. In the following we present these concepts in lockstep with a succinct text notation (AsTMa= [8]).

2.1 Factual Information

Topics represent subjects, which can be anything, physical or not. To further knowledge aggregation, topic identity can be supported by specially interpreted IRIs. In the case of objects which reside at certain network locations such identifiers will naturally be URLs. For example, a given SOS deployment can have its endpoint be used for identification:

```
demo-sos isa SOS-deployment = http://env05.arcs.ac.at/SOSsrv/
```

In the notation above such a *subject locator* IRI is symbolized by prefixing it with `=`. The topic identifier `demo-sos` is only local within the map and can be used there to refer to that topic. If a subject does not have a network address, then one (or several) *subject identifier*s can be used for identification:

```
arcs isa organisation ~ http://www.arcs.ac.at/
```

These identifiers are meant to *indirectly identify* the subject, such as web sites for organisations, images for persons, and so forth.

As also shown in the example above, topics can have types, i.e. are instances of a class. That itself is just another topic, to be elaborated on in this map or in some peripheral ontology. Topics can also have any number of *names* attached, signalled by `!`:

```
arcs isa organisation ~ http://www.arcs.ac.at/
 ! Austrian Research Centers
 ! acronym : ARCS
 ! branding: Austrian Institute of Technology
 ...
```

Names can be typed to allow to use different names for different purposes. While the first above is just a `name`, the next is an `acronym`, the other a `branding`. These types are again topics.

To attach values to topics *occurrences* can be used. To add, say, a `homepage` or the number of employees one would add to the above

```
...
homepage    : http://www.arcs.ac.at/
nrEmployees : 1000
```

The data types here are implicit (IRI and xsd:integer, respectively), but it can be made explicit as well. The types of occurrences themselves (`homepage` and `nrEmployees`) are further topics.

Relationships between topics are expressed via associations, whereby every involved topic is a player of a certain role. The fragment

```
provisioning (provider : arcs, service  : demo-sos)
```

means that `arcs` (in the role `provider`) provisions the `demo-sos` (in the role of a `service`). Obviously the whole association itself is also of a certain type (`provisioning`). Notably, an association has no intrinsic direction. It captures a certain fact, together with all involved parties. Other examples would be `marriage`s, or—to stay within the theme—`observation`s and `measurement`s. The roles themselves (`provider`, `service`) are also topics to be detailed somewhere to the extent necessary.

2.2 TM Query Language

Instead of using an API into a consolidated topic map, we leverage TMQL [6] as access language. Like any other query language TMQL has two concerns: (a) to locate and detect certain information in the queried topic map, and (b) generate output based on the detected information. One familiar type of output is the tabular form and it can be requested using a SELECT syntax:

```
select $p / acronym, $s =
where
  provisioning (provider: $p, service : $s)
```

A query processor will first try to find all associations which follow the pattern above, i.e. have the required association type and the given roles. Once such an association is found, the variables `$p` and `$s` will be bound to the respective players in the captured association. On the outgoing side, `$p` and `$s` will be used in the SELECT clause to evaluate *path expressions*. The expression `$p / acronym` would evaluate to all acronyms of what `$p` is currently bound to. The expression `$s =` would return all subject locators of the topic bound to `$s`. The overall result would be:

```
"ARCS", "http://env05.arcs.ac.at/SOSsrv/"
```

The query language is flexible enough to also generate XML output, not as string via text templates, but in an internal representation (DOM). For this, one has to switch into FLWR (For, Let, Where, Return) style:

```
return
<services>{
  for $p in // organisation,
      $s in // web-service
  where
     provisioning (provider : $p, service  : $s)
  return
     <service href="{$s =}">{$p / acronym}</service>
}</services>
```

While the WHERE clause remains the same, the variables and the values over which they range are made explicit. In the case of `$p` it should be all instances of `organisation` and for `$s` all instances of `web-service`. The returned content is now organized as an XML structure. The expected output would then be:

```
<services>
   <service href="http://env05...at/SOSsrv/">ARCS</service>
</services>
```

Additionally we assumed here that a `SOS-deployment` is a subclass of a `web-service`. Only then taxonometric reasoning will deliver the above result.

Inside such an XML query string it is also trivial to embed further topic map information, such as the name of the service provider:

```
<ows:ProviderName>
        {$p / acronym || $p / name}
</ows:ProviderName>
```

The example also shows how TMQL expressions can be used to deal with incomplete or highly variable data. Above, for instance, we looked first for provider `acronyms`. If there were none, the query would fall back to the full `name` for the provider (`||` is the shortcut '`or`').

Naturally TMQL supports loops over repetitive items, so it is straightforward to include, say, a list of SOS offerings:

```
return
<ows:Parameter name="offering">
  <ows:AllowedValues>{

  for $o in // offering [ . <-> location == vienna ]
  return
     <ows:Value>{$o !}</ows:Value>

  }</ows:AllowedValues>
</ows:Parameter>
```

The path expression `// offering` will compute all instances of `offering` in the map; notably not only direct ones, but also instances along any subclass hierarchy existing the map. Then the path expression continues with a filter (indicated by `[]` brackets). It only passes those things (each thing referenced with `.`) which have an association of type `location` with a topic `vienna`.

One by one, each Viennese offering is bound to the variable `$o`. With such a binding the RETURN clause is evaluated. It will extract the topic identifier (via `$o !`) and embed that into the XML fragment.

2.3 Extending the TM Model

While the generic Topic Map Model (TMDM [7]) is sufficiently equipped to host all information we need for the experiment, it does not do it elegantly, or efficiently. Rather than to shoehorn measurement data, temporal and spatial information into the model, we decided to experiment with rather minimal extensions to the official TM data structure. Naturally, these extensions will propagate to the notation and further to the query language.

The first step is to allow numerical values to have physical units, such as `5 kg` or `20 mg / m^3`. Rather than to host quantity and the value inside a topic or a dedicated association, we prefer to define a new basic data type. Accordingly, the impact on the model is minimal. Only the notation to create map content has to allow units:

```
temperature-vienna isa temperature
value: 18 celsius
```

The very same notation is extended into TMQL, also to compare values and perform computations with units.

Another modification concerns how values and topics can be related. According to the standard model literal values can only be hosted inside occurrences. To make them take part in an association one would

have to create a stub topic to hold the occurrence with the value. We lift this rather arbitrary restriction and allow values also to directly take part in associations, albeit only as players. As a secondary benefit we can now interpret occurrences as specialized associations, and names as specialized occurrences.

A more dramatic model extension is needed to naturally host *time sequences*. These are the most prevalent data structure in our targetted application domain, so an effective coverage greatly affects the scalability of any semantic system, both in terms of speed and complexity.

One particular value, say, a *measurement* inside such a sequence could be captured with an association:

```
measurement (value      : 30 mg / m^3,
             phenomenon : SO2)
```

Theoretically, the time aspect can be added using a predefined role `time`:

```
measurement (value      : 30 mg / m^3,
             phenomenon : SO2,
             time       : 2009-03-07T17:01)
```

The downside of this approach is that the lack of any temporal role inside an association leaves that time aspect open to interpretation. And so does the case when more than one such role exists. This makes interpretation by query processors difficult.

Another alternative would be to use the Topic Maps scope, an already existing mechanism to restrict the *validity* of an association. But *scope* is not a very well defined concept and is used for other contexts as well.

Instead, we redefined associations to have a time stamp, one which always exists. Such strict interpretation enables query processors to perform interval algebra operations (inside, outside, overlap, ...). If the timestamp is left undefined, then the association will range over all times. As physical events are never instantaneous but interval-based, an interval length can be added to the time stamp. We do allow the interval to be positive or negative to express subtle, but ultimately important information about when the value is created and whether its validity extends into the future or the past. Of course, the interval can be zero, covering the theoretical instantaneous case.

A typical example using time stamps with negative intervals is that of the gliding mean: The time when a mean value is computed will become its time stamp. The length of the time window over which the mean was built will be pointing into the past. Alternatively, mean values can also be computed over future time windows as is the case in non-causal systems.

On the notational side, this extension is trivial; we simply allow time stamps and time intervals to be added to an association:

```
measurement (value      : 30 mg / m^3,
             phenomenon : SO2 )
      at 2009-03-07T17:09:37 - 3 hours
```

3 Formula 3

F3 is a functional language that transforms time sequences. Time sequence processors (TSP, Fig. 1) can consume any number of sequences on the incoming side.

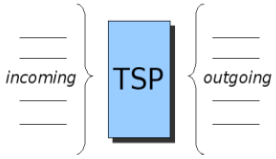

Fig. 1. Time Sequence Processor

TSPs are called a *source* if no sequence is expected. Typically these are constants or data fetched from a database backend. TSPs can produce any (finite) number of sequences on the outgoing side; *sinks* produce nothing and or are used for debugging, visualisation or again, database storage.

When a TSP is triggered into evaluation, it will consume a certain number of sequences on the incoming side. With these (and an additional variable binding to fine-control its behavior) the TSP will perform its computation. If there are still sequences left on the incoming side, then the computation will be repeated with those, continuing until the incoming side is exhausted. All partial results will be combined into one outgoing sequence of time sequences. A *greedy* TSP is one which consumes always all incoming sequences.

3.1 Time Series Abstract Data Model

While F3 makes no assumptions about the provenance of a time sequence, it has the abstract expectation that it is a linear array of chronologically ordered *slots*.

The time information within the slot is not just a time stamp (with a system specific precision). A time duration marks the temporal extension of the slot. That duration can be positive or negative, depending on whether the validity of the slot reaches into the future or the past. Slots also have a *logical time* which is the index in the sequence (starting with `0`).

The payload in the slot has the form of key/value pairs. Keys are either simple identifiers or take the form of QNames or IRIs. Values are either anything of the former or literals such as strings, integer, floats or application-specific objects such as images and matrices. They can also be time durations or time patterns.

When slots are combined into a sequence obviously their time stamp and their signed durations have to be honored. Any temporal overlaps have to be resolved to arrive at a *functional time sequence*, i.e. one which can deliver *one* slot for *one* particular time stamp.

3.2 Virtual Machine Operators

F3 defines a minimal set of primitive TSPs. As a whole they cover all possible computation patterns as all high-level language elements can be compiled into this set. Ignoring optimization, any implementation of F3 will only have to implement these operators.

- `Null`: This operator takes one sequence and creates none.
- `Nmap`: This operator takes one sequence and iterates over all slots. On each of them a lambda expression is evaluated returning a new slot. A new sequence is constructed from these slots.
- `Nreduce`: This operator takes one sequence and iterates over all slots. On each of them it will evaluate a lambda expression which aggregates the slot into an aggregate slot. That will be the only slot in the outgoing sequence.
- `Nfork`: This operator takes one sequence and evaluates a lambda expression on each slot. The result values are used for classification in that one outgoing sequence is generated for each different value, holding only those slots which produced exactly that value.
- `Ngrep`: This operator takes one sequence and evaluates a lambda expression on each slot. Slots for which that result is empty are discarded. With the others an outgoing sequence is constructed.
- `Tfork`: This operator takes one time sequence and slices it according to a time pattern and a window size. The time pattern (for instance *every 3 hours*) defines a number of time stamps, all computed relative to the start of the incoming time sequence. The time stamps are shifted along the window size, resulting in a number of individual time windows. These are used for slicing the incoming sequence into individual time sequences.
- `Tjoin`: This operator is the inverse of `Tfork`. It joins all incoming time sequences into one. Any temporal overlaps will be resolved.
- `Tee`: This is the identity operator. It echos all incoming sequences. It is used for debugging and visualisation.

3.3 Surface Syntax Operator

While applications can use an API to compose primitive TSPs for complex processing patterns, formulating transformation algorithms is more convenient using a compact syntax. Such a transformation can consist of (a) parameters for fine-control, (b) formal parameters for time sequences expected by the operator, and (c) blocks to generate values along a particular timeline.

At the beginning of a TSP definition a parameter block can define one or more modalities of that operator:

```
#-- modal parameters ----------
{ progress => 30 mins }
```

Each parameter can be associated with a default value which the invoking application can override.

This is following by a list of formal time sequence parameters. The example TSP expects two sequences, one which holds wind directions measured in degrees and another holding wind speeds:

```
#-- sequence parameters -------
@Direction { phenomenon => wind,
             unit       => degrees }
@Speed     { phenomenon => wind }
```

If a TSP is to return time sequences, then at least one block to generate values has to be declared. The first section of that handles the temporal aspect via the specification of a time pattern to be used, the second controls the quantitive aspect containing numerical computations, and the third aspect handles the meta data generation for that outgoing sequence.

```
#-- time pattern --------------
every $progress
#-- value generation ----------
< @Speed(t) if ( @Direction(t) < 30
            or   @Direction(t) > 330 ) >
#-- meta data -----------------
{ phenomenon => channel-0-speeds }
```

First the time pattern `every 30 mins` is used to compute time stamps starting from the earliest of the two incoming sequences. The time duration `30 mins` is taken from the modal parameter `progress` which is available as variable. For each of these times the `@Direction` sequence is sampled (under whatever interpolation regime that sequence is in) and that value is tested against the range `-30 .. 30`. If the direction falls within that range, the `@Speed` sequence is sampled at the same time and a value with the corresponding timestamp put into an outgoing slot. These slots are collected into an outgoing sequence. That is finally enriched by the meta information manifesting that these are speeds for a certain direction channel.

3.4 Modal Parameters

To fine-control the behavior of TSPs *modal parameters* can be declared using a simple key/value scheme. Key names are reinterpreted as variables to be used throughout the rest of a TSP definition. Values can be `undef`, any constant but also value expressions or time patterns.

The invoking application can optionally redefine the value of a modal parameter. If it does not, then the declared value will be used by default.

3.5 Time Patterns

The times at which new values have to be generated can be controlled via a time pattern language. That allows—in the simplest case—to enumerate individual times. But more general is the use of a declarative time pattern specification. That uses repeating temporal patterns, such as `every N hours` or `hourly at 12:00`. To increase the variability, time patterns can be hierarchical in that first a longer pattern is specified and within that a more fine-grained subpattern:

```
yearly:
   in May .. June : every 2nd week
   otherwise      : every 30 minutes
```

Starting with the start time of all involved time sequences, that pattern would create a yearly pattern whereby in the months May and June a time stamp will be computed every 14 days. In all other months a 30 minute rhythm will be used.

When using these patterns, then a special variable `t` is bound always to one timestamp, one at a time. Apart from generating physical times, there is also the option to use the index as logical time, such as in `every 2nd tick` to address every second time slot in the incoming sequence. If no pattern is provided the default is `every tick`. Using logical time, the variable `n` will always contain the current index, and `t` will be bound to the time in the current slot.

3.6 Sequence Parameters

Operators can use explicit sequence parameters to not only give an incoming time sequence a local name, but also to impose certain constraints on it. Only if all constraints are satisfied, evaluation will continue.

In the example

```
@Direction { phenomenon => wind,
             unit       => degrees }
@Speed     { phenomenon => wind }
```

for the first incoming sequence it is checked whether the phenomenon measured is actually wind and than in degrees. If this test passes, the sequence will be bound to a variable `@Direction`. The second incoming sequence will be bound to `@Speed` if it passes its own test.

The constraints themselves are given by key/value pairs. Only if the incoming sequence has that very key and that key links to the same value, then the constraint is satisfied. Values can be left `undef` to simply check for the existence of a certain key. In any case, the keys are again reinterpreted as variables. These can then be used throughout the rest of the TSP definition and is bound to the sequence value for that key.

The binding of incoming time sequences to sequence parameters is purely positional. Any unbound incoming sequence is left for another evaluation round of the same operator.

If no formal sequence parameter is declared, then a default one named `@_` will be used. It will always consume a single incoming sequence and it can be used implicitly within value expressions, i.e. `(t)` instead of `@_(t)` and `[n]` instead of `@_[n]`. If an operator is greedy and needs to consume all sequences, then the special `@...` must be used to indicate this. It cannot specify constraints.

3.7 Simple Value Generation

Value generation follows mostly the syntax and semantics of conventional programming languages such as FORTRAN, C or Java. This includes the notation for constant values, the usual prefix and infix operators, the general function invocations and the precedence grouping with parenthesis. An example would be

```
log ( @Speed(t) / 1000  + $speed )
```

There are, however, some language specifica. Conditionals, i.e. expressions where the evaluation depends on a condition, are not written with `if` cascades or a ternary operators, but instead use individual postfix `if` clauses:

```
 < [n]         if n.depth < -100 m
or [n] * 0.01  if n.depth <  100 m
or 0           otherwise >
```

Depending on the `depth` component different formulas will be used to generate a value. The `otherwise` is syntactic sugar as the lexical order is used for testing individual conditions.

When expressions inside a condition are evaluated, they actually do not return a `TRUE` or `FALSE` value as this data type per se does not exist in the language. Instead `undef` is used for `FALSE`, anything which is not `undef` is regarded to be `TRUE`.

3.8 Units

Life sciences being the main application domain for the language, all expressions are also aware of physical units, specifically those from the SI system. This starts with constants having units, such as `3kg` or `27.7 m/s`. But it also implies that all computations must respect units as well. In expressions such as `@Speed(t) - 100 km/h` the *physical dimensions* of all operands must match, i.e. the `@Speed` time sequence must have only values with *length per duration*.

Every expression can also be *unit-converted*. One way of conversion is to impose an additional unit onto the value of the expression. In

```
@A[n] <-< mg
```

every value would get `mg` as unit. If it already had a unit, that would be added as if the computation `@A[n] * 1 mg` had been used. In the other direction any existing unit can be relinquished:

```
@A[n] >-> m
```

The processor will convert any value with a length dimension into the number of meters, dropping the unit altogether from the value leaving a simple scalar. That mechanism can also be used to scale values. In `@A[n] >-> 1 km` the values are converted to kilometers, or even leaving the SI system with `@A[n] >-> inch` which converts into inches.

3.9 Slot Selection

When using time patterns which iterate over the incoming time sequence(s), a natural way to access a particular slot is to use an index. Intuitively, the first (and oldest) value is addressed via a subscript `[0]`, the next with `[1]` and so forth. To count from the last (and youngest) value one has to use negative values: `[-1]` retrieves the last value, `[-2]` the second-to-last, etc.

Referring to one particular value in the sequence is only moderately useful. If one needs to operate on each individual value one needs to address the current value `[n]`. Logical indices can also be used to timeshift sequences. In the same way as `[n]` always points to the current value, `[n-1]` refers to the previous, `[n-2]` refers to that before, and `[n+1]` to the next. To shift a sequence into its own future, one would write `< [n-1] >`, and to shift it into its past `< [n+2] >`.

In the case that the iteration over time sequences is based not on logical but a physical time, the current slot can be addressed via `(t)`, `t` symbolizing the current time. Similar to above a particular past or future can be referred to by providing a negative or positive duration, such as in `(t - 30 secs)` or `(t + 3 days)`.

3.10 Aggregations

Slots can also be addressed in groups using a logical range. This is only used together with aggregations, so that

```
< [n-2 .. n].sum >
```

produces sums of the last 3 values. Such ranges can be open to the left or to the right. This reflects the traditional interval notation where parentheses are used:

```
< (n-2 .. n].sum >
```

Now only the last two preceeding values are added.

Aggregation intervals can also be specified via the physical time, such as in `( t-3 hours .. t ].mean` to compute a 3 hour mean value. Again the interval can be open to either side.

The aggregation functions themselves are predefined (`mean`, `sum`, `prod`, `max`, `min`, `count`). All work type sensitive, in the sense that for the first two the plus operator for that data type is used, for `prod` the multiplication and for `max` and `min` the comparison. That list can only be extended outside the language. The same is true if the aggregation functions need a special treatment of undefined values.

If the selected interval does not contain any value, only `sum` and `count` render something defined (`0`), all other aggregates become undefined.

3.11 Property Management

According to the abstract data model of the language, properties are always key/value pairs. This is to ensure that properties can be easily mapped to RDF triples and Topic Map information items, such as occurrences, names and associations. Properties can also be virtually supplied by the programming environment in that pre-registered functions dynamically compute properties.

The syntax ensures that properties work equally on individual slots, whole time sequences and even sequences thereof. Some properties *inherit downwards* in that they apply to a single slot if the whole sequence has them. One example is `unit` where individual slots have to redefine this property to override any sequence-wide value. The same applies to `location`.

Slot Properties Only in simple cases a time sequence will have one single value in each slot. In general, slots will contain any number of properties, each with a key and a corresponding value, be that interpreted as data or meta data. To access a value, it has to be dereferenced via its key:

```
[n] . phenomenon
```

If such a property did not exist in that slot `undef` would be returned. That makes it simple to use as test for property existence as in `[n] if [n] . phenomenon`.

On the outgoing side, slots with their properties can be created using the following canonical syntax:

```
{
  value      => A[n],
  phenomenon => iso:SO2
}
```

Obviously one particular key can appear only once. The key must be an

identifier (or QName or IRI), the value can be specified via an arbitrary value expression. That expression is always evaluated in the current variable binding.

The key `value` is predefined and simplifies the syntax when a slot has one distinguished value, the rest being meta data. Then namely

```
< A[n] * 2 { phenomenon => iso:SO2 } >
```

can be used instead of the canonical

```
< { value      => A[n] * 2,
    phenomenon => iso:SO2 } >
```

The key `value` is also the one used by default when accessing slots within expressions. The syntax `A[n]` itself is a shortcut for the canonical `A[n].value`. Other properties of the slot have to be explicitly accessed via their corresponding keys.

Some properties are predefined as they have a special meaning in the language. For time aggregation, for instance, the `delta` property will cover the time interval over which was aggregated. If several slot values are involved in a computation, then the time interval they cover is used. Otherwise `delta` defaults to `undef`.

Also the `unit` property is handled by the language according to the computation. If an incoming value had `m/s` and is divided by a value with a time dimension, then the outgoing property for `unit` will be `m/s^2`. It is only defined if there is exactly one `value` component.

Time Sequence properties Also individual time sequences can have properties attached. Again, some have a predefined meaning, such as `start` and `end`. These, respectively, represent the start and the end time of the sequence. As every sequence is working under a particular interpolation regime, the property `interpolation` returns an identifier for that interpolation method.

On the outgoing side, time sequence properties can be added directly after the value generator:

```
@A @B < .... > { location   => vienna-stephansdom,
                 phenomenon => @A . phenomenon }
```

Some default handling here allows to reduce language noise: If the TSP is operating only on the default sequence, then all its properties are automatically propagated. Only if the sequences are explicitly declared, then the properties have to be as well.

3.12 Composite Values

In many cases the values within the properties will be numbers or strings, so that the language can directly operate on them. In general, though, values might be matrices, images or arbitrarily complex. These objects would be completely opaque to the language.

One way to handle this, would be to make the application developer write special accessor functions for these complex objects and overload the relevant arithmetic operators. But in practical cases it is much more convenient to give the language access to value components and let *it* handle the arithmetic.

Even though F3 cannot know anything about the internal structure of such objects, it can postulate a generic accessor syntax which allows to drill down into any object. In the example

```
[n] . value . columns [3] [4]
```

the dot notation is used to traverse an assumed tree structure within the `value` property. And indices are used to select by a number. Alternatively to the dot we also allow a slash `/` to insinuate a path language:

```
[n] . value / columns [3] [4]
```

At evaluation time, the language processor hands over such path expressions to the object which has to resolve the path to return a simple value.

4 Operator Algebra

To reuse operators and reduce the overall complexity, individual operators can be combined to form larger ones. One way is to pipeline them, so that the result of one operator becomes the input of the operator next in the pipeline. In the following example the incoming sequence is first incremented by one, then the results are doubled.

```
< [n] + 1 >  |  < [n] * 2 >
```

Pipelines can be extended to any number of stages, a single operator being just a trivial pipeline. If one stage produces more sequences than the next stage can consume, again the repetitive evaluation semantics from section 3 is used. Consequently, the expression

```
< [n] + 1 > < [n] - 1 >   |   < [n] * 2 >
```

is equivalent to

```
< ( [n] + 1 ) * 2 > < ( [n] - 1 ) * 2 >
```

Generators (section 3.3) can be used to stack time sequences on top of each other. But also for already existing operators it is possible to stack them. That is achieved by connecting them with `&` (or alternatively with commas):

```
< [n] + 1 > & < [n] - 1 >
```

When evaluating a stacked operator S, all incoming time sequences will be duplicated and subjected to each of the inside operators. The time sequences produced by those are then stacked on top of each other, honoring the lexical order in which the stacking was defined inside S. Consequently, the expression

```
< 2 * [n] > | < [n] + 1 > & < [n] - 1 >
```

is equivalent to the single operator

```
< 2 * [n] + 1 > < 2 * [n] - 1 >
```

As one would expect, the `&` binds stronger than the pipelining operator `|`. That precedence can be overridden by grouping inside `()` parentheses.

5 Semantic Properties

From here on it is assumed that the Formula 3 processor has access to an underlying topic map. That semantic network contains information pertinent to the application domain, in the geosemantic case generic concepts from O&M [3], SensorML but also necessary geographical information and background ontologies covering observable phenomena. Such a network may be *materialized* or it may be *virtual* where external resources are mapped dynamically into the map [1].

Depending on the needs, there are several levels of engagement with the semantic network.

5.1 Identification Regime

When testing for certain properties and when creating keys and identifiers one important aspect is a consistent identity management for any addressed subjects. Only with this quality assurance measure a robust and long-term management of data is feasible.

In this scenario a Formula 3 processor accesses the underlying topic map whenever an identifier, a QName or an IRI is used in the property handling. In the case of a simple identifier that is interpreted as topic identifier of an existing topic inside the map; when an IRI is used, then that must be one subject identifier for a topic there.

The resolution of QNames, such as `xsd:integer`, is more complex: First the QName prefix (`xsd`) is interpreted as topic identifier in the underlying map. That topic must be an instance of an `ontology` as—according to Topic Maps concepts—it has to reify a map with all the vocabulary in that corresponding namespace. That ontology is then consulted and in there a topic with the topic identifier `integer` must exist.

5.2 Path Expressions for Values

A further step is to allow TMQL path expressions everywhere where simple expressions in Formula 3 are allowed. At evaluation time these path expressions are evaluated against the underlying topic map. Any existing variable binding can be passed into the path expression. With that the following is possible:

```
< value     => [n] . amplitude,
  intensity => {   // wave-forms [ ./low  <= $value ]
                                [ $value <= ./high ]
               } >
```

For every value in the one (anonymous) input sequence the

`amplitude` property is extracted and propagated as `value` property to the outgoing sequence. Additionally, the amplitude value is bound to the variable `$value`. The TMQL path expression is wrapped in {} brackets. It will first find all instances of `wave-forms` in the underlying map. It then will filter out those which have a high-low range inside which the `$value` lies. That remaining wave form topic will be returned in form of its topic identifier. With small modifications of the path expression also the subject identifier or topic name(s) can be requested.

Path expressions can also be used for properties of outgoing sequences as the following example demonstrates:

```
@A{ pheno => undef } # formal parameters
  < ..... >       # generating values
  { risk-level => { $pheno / risk } }
```

The `pheno` property of the incoming sequence `@A` is first bound to the variable `$pheno`. Then—when it comes to create the result properties—the phenomenon is used as starting topic to find a `risk` occurrence in the underlying map. Whatever is returned here (string, URL, ...) is embedded as value of the property.

5.3 Path Expressions as Properties

TMQL path expressions not only can provide a property value, but also implicitly define the key as well. All this depends on *what* a path expression actually returns.

In the most simplest case, a path expression can return an occurrence. That—according to the Topic Maps paradigm—consists of a type, a value (and the scope). Ignoring the scope, the type can be naturally interpreted as key and the occurrence value as the corresponding value. This is demonstrated with the following:

```
{                          # properties
  { tsunami / wikipedia }, # path expr
  { tsunami / homepage  }  # path expr
 }
```

When these properties are generated for a slot or for a whole sequence then first the `tsunami` topic is located in the underlying map. Then an occurrence of type `wikipedia` is looked for. If it exists, then the type `wikipedia` will be used as key and the WikiPedia URL as value. Similarily for the `homepage` occurrence.

What works for occurrences also works for names. Also here the name type will be used as key. The name itself is always a string and it will serve as property value. Also Topic Maps associations have such an embedding rule: Here per association role one property is generated. The role itself is used as key, the player for that role is the value.

6 Related Work

As the work here is rather architectural in nature, we touch several areas. First and foremost it is the choice of the semantic technology which impacts on the available infrastructure, feature sets and limitations. As we have chosen Topic Maps for our experiment, we will first argue this decision with some rationale. The remaining sections deal with similar processing models from which we have drawn ideas for the language F3, and with ways to extend a semantic network model with temporal information.

6.1 Topic Maps vs. RDF Rationale

While there has been some work in conceptually mediate between the RDF and the Topic Maps model ([5] [4]) the two semantic technology stacks differ in various aspects.

In contrast to RDF, TM have been designed to be *subject-centric*, rather than *resource-centric*. Accordingly, Topic Maps include a dedicated identification regime with subject locators and identifiers to control how to address resources, physical objects and abstract concepts. In that they avoid any discussion *what a URI actually means* or any need to resolve theme on the network (`httpRange-14` issue). One practical consequence thereof is that *merging* of maps is based not only on the equivalence of two node IRIs within graphs, but also whether these IRIs are used as locator or identifier. Map merging is then more robust as several such identifiers may exist for one and the same subject. The identification regime also does not make it necessary to resort to heavy-weight ontology-based mechanisms, such as `owl:equivalentClass` or `owl:sameAs`.

In terms of statements Topic Maps offer not only single-valued properties (equivalent to RDF triples with literals), but also multilateral associations involving more than two topics. Multilateral statements not only avoid the use of blank nodes, something which adds to inferencing complexity. They also allow to directly model N-ary relationships and therefore put a topic in a relationship *in that particular context* (relativistic modelling). All associations are symmetric in nature avoiding the need to keep multiple versions of a property only to constrain later explicitly in an ontology that one is the inverse of the other.

Any statement context can be further refined with the use of a *scope* (not mentioned earlier). While somewhat underdefined, it limits in a standardized way the *validity* of a statement, a feature so useful that many RDF programming frameworks offer it and that SPARQL mimicks with the GRAPH concept.

TMs have no limitations on the use of class/instance relationships. One and the same topic can be a class and an instance in the same map. While this may have theoretical implications in some reasoning

scenarios, it drastically simplifies modelling of many (if not most) real-world scenarios where *sets of sets* are needed.

Like RDF, Topic Maps also allow to reify statements. The difference is that in TM only already asserted statements can be reified, staying consistent with a subject-centric approach.

In terms of the standards stack, Topic Maps use a fundamentally different layout. Instead of defining independently an ontology language (OWL) and a query language (SPARQL) directly on the model (RDF/S), Topic Maps first position the query and access language (TMQL) on top of the model (TMDM), committing hereby to closed world assumption and a particular inferencing regime. The constraint language (TMCL) is fully defined in terms of the query language; otherwise there is no ontology language for Topic Maps. With this setup the Topic Maps standards architecture limits the range of possible ontology languages, but it leads to a leaner overall model and a single point of definition for the semantics. That has a direct impact on the formal semantics and on optimization techniques when querying maps with known constraints.

There are also significant differences between the query languages:

- SPARQL only uses a pattern matching approach to detect certain node constellations in the underlying graph. TMQL offers that too, but additionally a path language to navigate to nearby corners of a map. The path expression language is powerful enough so that (almost) all queries can be expressed with it. It can also be used in SELECT clauses to further postprocess information bound to variables.
- TMQL can return customized XML content directly to be used by the invoking application, not just according to one particular standardized schema. This enables optimizations within TMQL and avoids situation where SPARQL query returns many NULL results which are eventually ignored by the application.
- As TMQL subscribes to the *closed world assumption* (CWA) it can offer a straightforward `NOT` operator within the WHERE clause. Many applications using SPARQL resort to postprocess the results.

The differences reflect that Topic Maps address rather controlled (and controllable) application scenarios, whereas RDF is more targeted to the (open) Semantic Web.

6.2 Temporal Extensions

There seem to be two schools how to embed temporal information into an existing semantic network model. The first, unobtrusive approach taken by OWL Time [9] is to define a dedicated vocabulary covering things *time events*, *durations* and *intervals* together with their relations (*contained-in*, *overlaps*, and so forth). Given an appropriate data type for the representation of dates, process information can be modeled as nodes labeled in this vocabulary.

The intrusive method is to modify the model itself. In the RDF space this has been proposed by [10]. While their background is to capture historical events and incremental changes, their line of argumentation is valid in the environmental monitoring domain and holds equally well for Topic Maps, as it does for RDF. Specifically the ability to reason over temporal aspects much more effectively than using a vocabulary approach is relevant for applications on a larger scale.

Extending the Topic Maps model by an intrinsic temporal component on associations—or in fact on any statement—we follow this approach. The variation we introduce is to also store the time interval (even together with a direction) to even better reflect the nature of our data corpus.

7 Summary

One of the driving factors for this work is to offer a consistent framework—conceptually and then in terms of programming languages—to manipulate time series of observation data. This is relevant for both, adhoc virtual sensors as well as for *from the craddle to the grave* long-term management.

Preliminary work on integrating semantic technologies into time series processing had shown that first not only concepts from the life sciences domain, but also that of the sensor web domain have to be aligned. Only then an integration of involved programming languages seemed feasible, something which also suggests global optimization opportunities.

One downside of our approach to extend a semantic technology model with temporal aspects is certainly that off-the-shelf software cannot be used. On the upside time sequences can now be hosted natively inside topic maps, make pathway for a range of other applications.

Our ontological coverage of O&M and SensorML terminology at this stage is rather incomplete and will have to be improved in further efforts. One goal is to be eventually able to mediate process descriptions between F3 and SensorML. In this step the strong identification regime of Topic Maps might prove helpful for discovery.

To evolve F3 itself a research prototype has been implemented. In that, the obvious intent was to benefit as much as possible from existing functional programming techniques and pave the way for large-scale deployments (such as in clouds).

The integration with semantic networks is also ongoing work. Still a number of ideas have not been explored yet. Path expressions, for instance, could also be used in incoming sequence properties, not just to compute values from the topic map, but also to provide the tests themselves. Also the component selector sublanguage could be mapped onto TMQL path expressions, furthering the integration and making it more obvious to host time sequence data itself inside the semantic network.

References

1. R. Barta and T. Bleier. Semantically enabled SOS with Topic Maps, 2008. FOSS4G 2008, Free and Open Source Software for GeoSpatial Conference.
2. M. Botts and A. Robin. Sensor Model Language (SensorML), Open Geospatial Consortium Inc., ogc 07-000. 2007.
3. S. Cox. Observations and Measurements, Part 1 - Observation Schema, Open Geospatial Consortium Inc., OGC 07-022r1. 2007. Open Geospatial Consortium Inc., OGC 07-022r1.
4. L. M. Garshol. Living with topic maps and rdf, 2003. Technical Report.
5. L. M. Garshol. Q: A model for topic maps: Unifying rdf and topic maps, 2005. Extreme Markup Languages 2005.
6. L. M. Garshol and R. Barta. TMQL, Topic Maps Query Language, working draft. ISO/IEC JTC1/SC34.
7. L. M. Garshol and G. Moore. ISO 13250-2: Topic Maps - data model, 2008-06-03. http://www.isotopicmaps.org/sam/sam-model/.
8. L. Heuer and R. Barta. AsTMa= 2.0 language definition. 2005. http://astma.it.bond.edu.au/astma=-spec-2.0r1.0.dbk.
9. J. R. Hobbs and F. Pan. Time ontology in owl, w3c working draft 27 september 2006, 2006. W3C.
10. T. Kauppinen, J. Väätäinen, and E. Hyvönen. Creating and using geospatial ontology time series in a semantic cultural heritage portal, 2008. ESWC 2008.
11. A. Na and M. Priest. Sensor Observation Service, Open Geospatial Consortium Inc., OGC 06-009r6. 2007.
12. O. Lassila and K. Swick. *Resource Description Framework (RDF) model and syntax specification, Technical report, W3C*. Camo AS, 1993.
13. G. Percivall and C. Reed. OGC Sensor Web Enablement Standard. *Sensors & Transducers Journal, Vol. 71, issue 9, pp.698-706*, 2006.