Geo-Semantic Time Series for Tourism Applications

Robert (rho) Barta[1] and Christoph Grün[2]

[1] rho information systems
rho@devc.at
[2] Technical University Vienna
Institute of Software Technology and Interactive Systems
christoph@ec.tuwien.ac.at

Abstract. Future semantically-enhanced tourism information systems will be very dynamic. All dynamic data can be organized into time sequences, i.e. sequences of values, each value not only carrying a core message, but also some circumstantial information (when, how, where). Our proposed approach is not only to represent these time sequences within a semantic network store. More importantly we demonstrate how to use an appropriate functional language, Formula 3, to compute new time sequences based on existing ones. These transformations can be time aggregations, but they can also include the underlying semantic network. Newly derived values then mark the emergence of events which can be reported back into the application code.

1 Motivation

Like many other semantically enabled systems, future tourism information systems will have to manage not only core information, such as about tourism sites, travel and event options, but also more and more tangential information. Only with the inclusion of environmental (weather, traffic, etc.) and operational information enough knowledge can be aggregated to fine-tune and individualize tourism offerings.

Semantic network technologies, such as RDF and Topic Maps are well equipped to host core and tangential information in a flexible form, but they have the severe deficiency that they (a) cannot represent information dynamics, and (b) provide any temporal semantics.

For the application developer this means that she will have to define the static model as ontology, but any dynamic aspects of the system will mostly reside inside the application code. If these (business) processes are not overly specific and involve only relatively simple transformations, then a semantic middleware can take over some, or even most, of the processing burden.

Modelling the dynamic aspects of a system implies that the temporal progress of data is made explicit. One approach borrowed from the Sensor Web (Sensor Web Enablement, SWE) is to identify particular *features of interest* in the system and to observe how the values evolve over time. The geographical position of a particular tourist over time, or the temperature measured at a particular site is such a feature. In any case the focus is on the *observation*, i.e. the measurement (or computation) of a quantity at a certain time and for a certain phenomenon. Additionally, any number of environmental and circumstantial information can be stored along this observation.

As soon as observations of the same feature are collected into a chronologically ordered time sequence, new time sequences can be derived from that using transformations. Quantitative transformations can range from simple aggregations, such as the mean value over a certain time interval, up to arbitrary functions. Qualitative transformations affect the *meta data*, i.e. the properties of the observations. Sometimes it is sufficient to simply

propagate them, so that the data can be interpreted properly; sometimes one needs an underlying semantic network to derive new properties based on the existing ones.

In this work we report about the current state of a research project revolving around the framework *Formula 3* (F3) to support above applications. It consists of a domain-specific language (DSL) for expressing transformations on time sequences, together with a corresponding compiler and run-time system. All this is packaged into the F3 middleware.

The F3 language is aware of *physical time*, *values with physical units* (meter, seconds, ...) together with the usual mathematical operators defined on it. That way computations together with their temporal aspects are explicitly documented, rather than implicitely within a standard procedural programming language. This more declarative approach also gives way to optimization, such as stream processing and parallelization (for multi-core platforms or inside a computing cloud).

The language also supports *property management* which is designed to operate with and in the absence of a semantic network. In that latter case, *semantic transformations* fall back to handling properties of observations and those for whole time sequences only within F3. As soon as a semantic network (in RDF or Topic Maps form) exists, property handling can use an existing semantic access language. For that we are not using SPARQL, as its syntactic structure makes it difficult to embed concise expressions into F3. Instead we adopt TMQL, the query language for Topic Maps, and adapt its path expression sublanguage for the use over RDF graphs.

2 Architecture

For the class of applications we are targeting we are postulating the following setup (Fig. 1). In an underlying tuple store (such as that from AllegroGraph) we will maintain instance and ontological information pertinent to tourists and tourism operators.

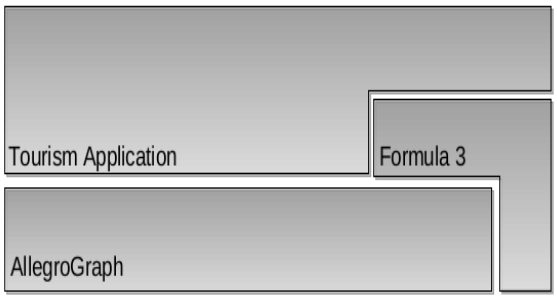

Fig. 1. Typical Tourism System Architecture

The application may directly approach the store and perform directly the required computations. Depending on the application developer's inclination, though, a certain amount of dynamic aspects can be abstracted into time sequence transformations and can so be expressed in F3. Depending on the nature of the transformation the middleware would evaluate the F3 expressions with or without the involvement of the store.

Hereby F3 can be used in an *active* and a *reactive* mode. In the *active mode*, the application always stays in control, say, because it already hosts a main event loop, say,

that of a user interface. When indicated, the application can pass an F3 expression into the middleware, which will parse, compile and then evaluate it. Eventually it will return a result set of time sequences back to the application.

In the *reactive* mode F3 expressions will be first deposited into the Formula 3 middleware. That will then play the active role, reacting on new data (using its own listeners) and passing data to the application when appropriate (via listeners the application has registered).

In either case, F3 can access the tuple store when the transformation requires it.

3 Topic Maps as Convenience Layer

For technical but also conceptual reasons we use Topic Maps as a thin layer of semantic abstraction on top of existing resources, including those hosted in RDF graphs. This allows us to use the slightly more sophisticated addressing (subject addressing and indication), add extensions regarding geospatial and temporal data, but also leverage TMQL (Topic Maps query language, ISO 18048) as query and access language.

3.1 Geo-Spatial Data

While underneath the TM graph will be mapped onto an (RDF) store, on the surface we can formulate any manually created domain knowledge in a human-friendly notation: Sites, such as the Palace Schönbrunn can be represented with the following Topic Maps fragment:

```
schoenbrunn isa building from 1699                # exists from that point on
~ http://en.wikipedia.org/wiki/Schoenbrunn        # subject identifier ...
~ http://www.schoenbrunn.at/en/home.html          # ... indirecly identifies
! Schloss Sch nbrunn @de                          # name in german
! Schoenbrunn Palace @en                          # name in english
  geo:loc : 48 11 4 N, 16 18 43 E                 # property
  as palace is-planned-by                         # involved in association
     "Fischer von Erlach" as architect .
```

The fragment postulates the existence of an individual with an identifier `schoenbrunn`. The identifier is *local* and is only valid within the enclosing map. That individual is an instance of the class `building`. That existence is constrained to only start in the year 1699 (defaulting to Jan 1st, 00:00), and not to hold earlier. Other options to constrain are `before/since`, `from/after` or a `from .. to` temporal interval.

Differently to RDF every TM node (called *topic*) can hold any number of IRIs for identifications (to promote merging). The IRIs above are *subject identifiers* as they only indirectly identify the subject in question, namely the real physical palace, not a particular document.

All this is followed by various labels, i.e. names for the topic in different languages. The geographic location is then added as simple property (*topic occurrence*). For the format we use geographical longitude and latitude as format convention.

The final clause simply refers to the fact that the palace was designed by Fischer von Erlach. The clause implicitely introduces more topics (`palace`, `is-planned-by`, `architect`) where the use of the local identifier is quite handy. As we have only used the name for the "Fischer von Erlach" topic, one of two things will happen when this is internalized:

If a topic with that name already existed, then that topic is used. Otherwise a new topic with an artificial local identifier is generated and that name is attached to it.

In this overall assertion all nodes play certain roles: the "Fischer von Erlach" topic plays the role of the architect, Schönbrunn the role of a palace. It is the Topic Maps way to model relationships symmetric. The type `is-planned-by` simply characterizes the type of this relationship.

3.2 TMQL path expressions

TMQL has been primarily designed as a query language for the Topic Maps data model (TMDM). The language supports several syntactic styles how it can be used to query a semantic network and to construct results. The one style relevant for this discussion here is based on *path expressions*.

Differently to SPARQL it is assumed hereby that all query expressions have a specific node as starting point. Relative to that a path expression defines how the underlying graph is to be followed. If a path expression contains several subsequent steps, they will be followed in lexical order from left to right. With each *navigation step*, the graph is traversed and a new set of nodes is reached, so that followup steps are applied at each of the nodes, resulting in a new set. Additionally *filter steps* reduce the result set to those nodes which satisfy the filter.

Given the network about the Palace Schönbrunn above, we could start at the node `schoenbrunn` itself and ask for all names (labels) of that topic:

```
schoenbrunn / name [ @de ]
```

In the first step all name labels are generated with `/ name`. The subsequent filter selects only those names with a german label. Properties of topics (*occurrences* in TM parlor) can be as easily addressed with

```
schoenbrunn / geo:loc
```

The result is an object holding the coordinate. If no such occurrence existed for a topic the result would be the empty list.

Path expressions can also be used to address certain topics differently. For the architect von Erlach we can use his name and follow the *name axis* from above in the reverse direction:

```
"Fischer von Erlach" \ name
```

That would also identify the topic and comes handy when the internal (or any other) identifier is not known. With a similar mechanism it is also possible to identify a topic using the geo-coordinates:

```
"48 11 4 N, 16 18 43 E" \ geo:loc
```

As with names, the result may not be unique, though.

From that particular architect node above we could then find all things he has planned:

```
"Fischer von Erlach" \ name <-> is-planned-by
```

The axis `<->` would find all associations a topic is involved in, would check whether any of the associations is of type `is-planned-by` and would traverse these associations. For

bilateral associations (linking two topics), always the topic *at the other end* is the result of that step. For multilateral associations all topics except that from which we started are in the result set.

If we needed to make sure that we only receive `building` topics, i.e. instances of `building`, we have to filter for this type:

```
"Fischer von Erlach"
      \ name
      <-> is-planned-by [ . isa building ]
      / name [ @de]
```

The dot (`.`) symbolizes the *current topic* as the filter is applied to each topic in the result set before. Additionally, we have asked for the german names of these buildings.

We could also query for all buildings in the graph. This is achieved by first asking for all nodes in the graph, and then filtering for the required type:

```
// * [ . isa building ]     # or even shorter // building
```

Naturally type transitivity as in RDF(S) is honored, so that anything classified as `castle` will be returned if `castle` is a subclass of `building`.

If the requirement would also include to find all nodes representing buildings in existence in the year 1700, then we need to temporally qualify that existence statement:

```
// * [ . isa building in 1700 ]
```

As soon as temporal constraints have been added, all times will be interpreted as time intervals. This, by default, applies to years and other usual time durations. Apart from the temporal inclusion (`in`), other interval operators such as explicit intervals (`from 1650 to 1750`) are provided.

4 Data Dynamics in Tourism Information Systems

As with many *semantic applications*, operational parts of tourism systems have to integrate information from many, often heterogenous sources. In our running example we will restrict ourselves to ticket purchases at a certain tourism site, say, the Schönbrunn Palace. Additionally, we will integrate weather sensor data, in particular temperature measurements.

To present a rich picture for what one can use the F3 infrastructure, we first start with consolidating and sanitizing the sensor input to establish a minimal semantic quality. Gradually we will then add (a) data manipulation steps, ranging from (b) purely numeric computations, (c) temporal aggregation and simple property-based transformations, to finally progress then to (d) fully-blown semantic transformations and semantic aggregations.

4.1 Semantic Canonicalisation

Data usually originates from a variety of sources, and in a plethora of formats and representations. Integration into the F3 infrastructure is achieved in one of two ways: Either by using a loading module for supported formats such as CSV (comma separated values) or one of the XML formats. If alternatively time sequence data resides in an existing database, then F3 offers naturally a programming interface.

In the simple case of a CSV file, ticket purchase information could be delivered into the F3 infrastructure as follows (blanks added for better readability):

```
Date;                  ID;"Value";Code
10/03/2008 12:10;T000005;        ;A
10/03/2008 12:20;T000006;        ;C
10/03/2008 12:30;T000007;        ;Z
10/03/2008 12:40;T000008;15      ;G
10/03/2008 12:50;T000009;        ;A
10/03/2008 13:00;T000010;        ;G
```

Apart from the date (including hour, minutes, etc.), the ticket ID and its class (`A` = adult, `C` = child, `Z` = concession and `G` = group) are recorded. The number of persons is listed in the column "Value" and happens to be only defined in the case of group tickets. Otherwise this number is understood to default to 1.

To uplift this data fragment into a proper F3 time sequence, all implicit information has to be made explicit. Hereby one has to decide how this data is organized into a sequence of observations, each of these observations occupying a *time slot* in the sequence. One obvious choice is to use the date as time stamp for the observation; still, there is the subtle decision whether the observations above are *instantenous* or last over the full minute.

Another decision to make is which column is supposed the *natural value* in the observation and which other columns are regarded *circumstantial*. In a somewhat arbitrary way we here choose the number of tickets to be the value. To make sure that the default value mentioned above is made explicit we have to visit every value, test it whether it is undefined (or, equivalently, empty) and insert then the default value:

```
< 1 if [n] = ''
or [n] otherwise >
```

Given our original time sequence as input, the above F3 operator will create a new one as output. `[n]` hereby symbolizes a current observation value in the input, one at a time. This F3 expression mandates that if that current value happens to be empty, `1` is emitted, otherwise the value itself is the result. This exercise is so repeated for every time slot in the time sequence. The implicit loop is symbolized by the enclosing `< ... >` pair.

Should default values depend on additional conditions, they can be formulated in a *waterfall IF* structure:

```
< 1  if [n] = '' && [n].Code != 'G'
or 10  if [n] = ''
or [n] otherwise >
```

With the above we have captured an additional default for group sizes. The selector `[n].Code` refers to the `Code` property of the observation, so that now an estimated value for *group* tickets is provided.

The other aspect of the semantic uplifting concerns the *meta data* (properties), that for each individual time slot and that for the sequence as a whole. In our data example, the use of `Code` does not reflect its meaning. To adapt that to the background ontology, F3 provides dedicated syntax to generate properties:

```
< [n] { rdf:type => tou:adult-ticket if [n].Code = 'A'
                 or tou:group-ticket if [n].Code = 'G'
                 or .... ,
        tou:ticket-ID => [n].ID
        }  >
```

With this new, additional operator we again step through each individual value in the sequence. But this time, the data value itself is simply echoed via `[n]`. While the time stamps themselves remain the same, new properties for the observation are generated. In particular a more explicit *type* of the observation and also a dedicated ticket ID property is generated. All original attributes are lost in this step.

Also for the sequence as a whole it is prudent to specify more context. This can be achieved by simply appending

```
                  { f3:phenomenon => tou:ticket-sale,
                    tou:location  => tou:schoenbrunn }
```

to the above operator. The outgoing time sequence will now be tagged. `f3` is the native namespace for F3 and contains a number of terms which are interpretable by the F3 middleware.

4.2 Temporal Aggregation

Once the data is brought into shape, it can be further processed. One typical use case are temporal aggregations, such as a sum, count or mean over a past time interval. Hereby F3 is aware of not only the *logical time* (symbolized by `n`), but also of a *physical time* (`t`):

```
< [t - 1 hour .. t].sum > every 1 hour
```

As before such an F3 operator will consume one particular time sequence. But instead of iterating *logically* through the sequence, i.e. along the index of the time slots, an explicit time pattern `every 1 hour` is specified. The pattern acts like a calender, producing time stamps every hour. Hereby it is starting with the start time of the incoming sequence and will repeat the 1-hour pattern until the end of the incoming sequence.

For each of these time stamps the expression inside `<>` is evaluated. In the case above an interval will be built, one which ends at the current time stamp, `t`, and which starts one hour earlier. All values within this time span will be consulted and they will be aggregated as specified with `sum`.

Many variations are possible: The time pattern can be much more elaborate, taking different frequencies for different intervals (days, weeks, months, years, ..). Also the time interval itself can vary, being open or closed on either side, or extend into the future instead of the past (non-causal processing).

The following example demonstrates how one can maintain semantic consistency with F3. In the following we will compute the average income per hour from the earlier ticket sales time sequence. This is done in two stages: First the income per ticket is computed:

```
< [n] * tou:prices ([n].rdf:type) > { f3:foi => tou:ticket-income,
                                      iso:currency => iso:euro }
```

`f3:foi` stands for *feature of interest*, a term which becomes useful when the information is offered via a sensor web service. The currency vocabulary we assume to live in some external ISO 4217 vocabulary. In the computation we postulate a function `tou:prices`: It has to be registered with F3 before evaluation and it looks up the price per ticket depending on the ticket class.

In the second step that income is scaled to the hour:

```
< (t -1 hour .. t].sum / 1 hour > every 10 mins
```

Not only are we now using an asymetric interval (it is open on the left), the sum is *per hour*, i.e. the value has a unit compatible with `1/sec`. Also note that the time pattern now creates values every 10 mins, creating sliding windows over the incoming sequence.

4.3 Resource Integration

One of the concerns of integration of different resources is that they need to be *semantically compatible* before data is merged.

Let us assume we wanted to add to the ticket sales time sequence a current temperature as a new property. It means that we now deal with two input sequences instead of one. The original `@Tickets` sequence deals with the ticket sales, and a new `@Temp` sequence about the phenomenon temperature:

```
@Tickets { f3:phenomenon => tou:ticket-sale }
@Temp    { f3:phenomenon => tou:temperature }
  < @Tickets[n]
          { +temperature => @Temp(t) }
  > every tick of @Tickets
```

In contrast to the operators used before we have to name the incoming sequences in front of the operator. Not only have we given the two input sequences names (they are only valid within that very operator), we also have provided certain properties these sequences are expected to expose (so called *guards*): Unless the `@Tickets` does cover ticket sales and the `@Temp` is about temperatures, the operator will refuse to work.

Within the operator itself the ticket values are simply copied with `@Tickets[n]`. The properties for each time slot are enriched by a new one, `temperature`, all others are kept. If we had dropped the leading `+`, then `temperature` would have replaced all properties.

If we had omitted an explicit clause `every tick of @Tickets`, then the language semantics would make the F3 processor visit *every time slot* of *every involved time sequence*. This is not what we want in this case, so selecting explicitely the sequence from which we intend to visit the time slots is important. But once such a time slot is considered, its physical time is known and for that time the temperature can be computed with `@Temp(t)`. How exactly, will depend on the current interpolation policy for the sequence `@Temp`.

4.4 Geo-Spatial Constraints

To add a geosemantic constraint, we now additionally require that the temperature is measured on a location within a 2 km radius from the Schönbrunn palace. For this purpose, we have to preprocess the `@Temp` sequence and suppress those values which have been observed further off:

```
  < [n] if my:distance ([n].tou:location, 'schoenbrunn') <= 2 km >
```

In this solution the function `my:distance` has to be provided by the application. Notably, that function could return the distance in any length unit (meters, miles, or even parsec); F3 - being fully unit-aware - would convert it appropriately to compare it with the 2000 meters.

The alert reader may notice here that the property `tou:location` was never set for the individual time slots, while it was set for the sequence as a whole. Currently the language semantics dictates that such properties are inherited from the sequence to each time slot.

In order to put more burden onto F3, though, we can also use a predefined function `geo:distance`. But that would expect as parameters *coordinates*, not locations, or their identities as above. In our setting these coordinates can be queried off the underlying semantic network. For this purpose we use the TMQL path expression (3.2):

```
< [n] if geo:distance ( schoenbrunn { / geo:loc },
                        [n].tou:location { / geo:loc } )  <= 2 km >
```

In the same way as `schoenbrunn / geo:loc` finds Schönbrunn's position, the second path expression above would find the coordinates for the location in the observation. The braces around the path expressions disambiguate them from F3 algebraic expressions.

4.5 Classification and Semantic Aggregation

Many analyses of time sequence data involve to classify values into different clusters. If we had ticket purchases from all over Vienna in an incoming sequence and needed to separate these by location, then a predefined operator called `Nclassify` directly performs classification:

```
Nclassify ( lambda => [n].location )
```

The criterion along which will be classified is provided as *lambda* expression, i.e. a nameless F3 expression. The output of this operator is a set of time sequences, one for each location. To not loose this information, all sequences will be tagged accordingly with their location.

But sometimes it is not the location itself, but a criterion involving a semantic query. Using again a path expression we can also classify according to the type of the location:

```
Nclassify (lambda => [n].tou:location { >> type } )
```

Again we syntactically separated TMQL path expressions.

This form of *semantic aggregation* can make full use of TMQL path expression. The following would aggregate purchases along the architects:

```
Nclassify (lambda => [n].location { <-> is-planned-by })
```

5 Formula 3 Processing Model

F3 is a functional language that transforms time sequences. Time sequence processors (TSP, Fig. 2) can consume any number of sequences on the incoming side.

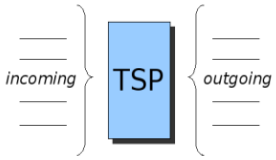

Fig. 2. Time Sequence Processor

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.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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 5 is used. Consequently, the expression

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

is equivalent to

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

Generators as above 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.

6 Current Status and Future Work

One challenge in building geo-semantic applications is that quantitative, temporal, spatial and semantic information has to be brought into one consolidated computational model. In this work we proposed 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 and Topic Maps, and it is designed to degrade gracefully in the absence of any semantic network.

The project is spearheaded by a Perl-based prototype for the F3 language, followed by a reimplementation in Python which can be used via JPython inside a JVM. Also the TMQL path expression has been prototyped in Perl, as has most of the Topic Maps based infrastructure. The integration of path expressions into F3 has not been fully achieved. Also many temporal features are still missing.

A mapping onto an AllegroGraph tuple store is next on the agenda with promising experiments. With that an existing tourism ontology together with typical sites as instance data can be hosted. Much concern here is on the efficient representation of whole time series. The ultimate goal is to be able to bundle instance data, ontologies, F3 expressions and tourism related functions into one deployable package.