

# Topic Maps: Fundamentalism meets Pragmatism

Robert Barta

Bond University  
School of Information Technology  
Gold Coast, Australia  
Email: rho@bond.edu.au

## Abstract

After years of development, the Topic Maps community is in the process of consolidating its standards landscape. This work tries to outline an integrated scenario and sketches how the individual pieces may fit together. This is mostly achieved by a mathematical formalization for Topic Maps. It serves as basis for high-level data models and also for a query language.

## 1 Introduction

Topic Maps (Garshol & Moore 2004-01-25) is one of the assertional knowledge representation techniques quite comparable to RDF (O. Lassila and K. Swick 1993). The basic assumption is that *in the real world* there are subjects and relationships between these subjects. These are modelled - actually reified - by constructs like topics and associations in a topic map. This high-level view of *what Topic Maps are* has been the understanding on which TMDM (Garshol & Moore 2003-11-02), the data model for TMs, is based. This model allows for a rather straightforward software implementation and is meant to address mainly industry developers.

On the downside, this data model is not minimal in terms of the paradigms and concepts used. This makes it a difficult target for mathematical formalization which implies that formal semantics—as needed for ontology definition languages and query languages for Topic Maps—cannot be directly defined on it.

Attempts to define a more canonical TM model include (Kipp 2003), (Garshol 2004-07-22) and various iterations of the TMRM (Newcomb, Hunting, Algermissen & Durusau 2003), the ISO TM reference model. All of these—with the exception of the recent Q-model—have been developed in isolation, without relation to any TM-related standard. This is where the  $T^+$  model draws its motivation from; to not only carve out a minimalistic, formal structure for TM-organized information, but also a foundation for the upcoming query (TMQL) and constraint language (TMCL).

This paper first provides some background on the current understanding of the high-level paradigms for TMs as defined in TMDM. This serves as contrast to introduce the low-level  $T^+$  model (section 3). Included in the model is also some mechanics to navigate in maps (section 4). Section 5 sketches the relationship between  $T^+$  and the data model TMDM.

Obviously, the query language TMQL (section 6) and its formal semantics cannot be covered in full detail here. Instead we prefer to concentrate more on the integrating aspects via the use of examples. The last section lists several areas which need further investigation. It makes clear that most of the involved technologies and standards are still in flux.

## 2 The Pragma

Until recently, Topic Maps have been mostly known via their TAO (Pepper 2000) concepts. According to that, a topic map contains foremost topics; these are focal points to attach pertinent subject information, such as one or more names, textual information and other data (called *occurrences*), or URI references to external information. The TM standard also provides various options to connect a topic with the subject it is supposed to reify. In the case that a subject actually has a URI, say, because it is a document on a network or an object in a database, the TM author can use this URI as *subject locator*. Many subjects, though, will have no single canonical place; for these a weaker form, *subject indicators* can be used to indirectly pinpoint the subject in question.

While topics tend to contain more dictionary information attached to them, most semantic information is modelled with associations. These constructs are quite heavy-weight in that they connect any number of topics whereby these topics are used in different ways; one of which are role-player combos. These seem to be quite pivotal to the TM paradigm as they enable to encode that a certain topic plays a certain role in the context of the association. An example association would be "France Telecom holds 30% of the scandinavian telephone market" where the topic "France Telecom" would be the player, playing the role "shareholder", the "scandinavian telephone market" would be another player, playing, say, a "market" and the fraction "30%" would be playing the actual "share". Any number of these role-player combos may exist in a particular association.

Both, associations and occurrences, can actually also be typed. And they can be put into a particular context (scope) to regulate that they should only be respected if that scope is *active*, whatever that may mean for a particular application. A popular (ab)use of scope is to have URI occurrences scoped to individual natural languages (english, french, ...).

The paradigm also includes the concept of *merging* of maps, specifically to support the aggregation of knowledge from different sources. The standards prescribe minimal conditions when topics are regarded to be about the same subject, specifically using the subject locator and indication mechanism. Applications, still, are free to use more circumstantial evidence to prove *identity* of two topics.

### 3 Fundamental Model

The model takes a model-theoretic approach by defining basic TM concepts. It homogenizes concepts such as topics, occurrences and associations into so-called *subject proxies*. The term 'subject proxy' (or short *proxy*) should indicate that a proxy stands for a certain factum in the real world. Topic maps then are simply collections of such proxies.

Proxies themselves are sets of labelled values. Interestingly, such a structure is suitable to express topic properties and association roles alike. Topic characteristics are interpreted as special-purpose associations, namely those which connect values to topics.

#### 3.1 Proxies and Properties

Since property values can be proxies themselves, the formal definition has to be recursive. We achieve this by bootstrapping first a trivial proxy which is the empty set  $\emptyset$ . *Bottom* does not have any property.

As our goal is to recursively define  $\mathcal{X}$ , the set of all proxies, we put *bottom* into  $\mathcal{X}$ . Given that and an arbitrary set of values  $V$ , we can then define also possible *properties*, those being key-value pairs,  $\langle k, v \rangle \in (\mathcal{X} \times V)$ . The first component of these tuples is called the *key*, the other is the *value* of this key. A single proxy then is simply a set of properties,  $\{p_1, \dots, p_n\}$ .

In this model, properties within a proxy have no ordering or no precedence over each other. It is only them which actually constitute the proxy. Still, for convenience, we freely attach identifiers to individual proxies to be able to refer to them. Other than this convenience, identifiers have no relevance.

As an example, we consider a proxy which supposedly *proxifies* a particular university:

```
{
  < name ,                "Bond University" >,
  < CRICOS-provider, "00017B" >
}

{
  name                => "Bond University",
  CRICOS-provider => "00017B"
}
```

While the block in the first notation is closer to the mathematical formalism, we prefer the one below. As value set we used here only strings; for the keys we have introduced adhoc identifiers to make the notation more readable. Technically, **name** and **CRICOS-provider** are only identifiers for other proxies which have to exist in our universe. If we never plan to be compatible with anything outside our university universe, then this can be completely arbitrary, such as

```
name                = { bottom => 42 }
CRICOS-provider = { bottom => 43 }
```

Using different integers, we ensure that these proxies are distinguishable from anything else.

This bootstrap mechanism opens also the avenue to introduce special proxies, such as *instance*, *class*, *subclass* and *superclass* which are ubiquitous in ontological applications. They also form a vehicle to define relationships *is-a* and *subclasses* which we need later.

If, for instance, we need to express that Bond University is an instance of the concept **university**, then another proxy can take care of this:

```
{
  instance => bond-university,
  class    => university
}
```

As noted earlier, the chosen structure of proxies makes the distinction between 'things' (modeled as topics) and 'connections' (modeled as associations) disappear. Everything is based on a qualified constellation of primitive values. As only the relative positioning of such values counts,  $T^+$  adopts a highly relativistic approach. Only the primitive things (sic) are expected to be grounded.

#### 3.2 Map

A *map* is simply a set of proxies. This makes it straightforward to compose maps. Given two maps  $m, m'$ , the elementary composition is fully based on set union:  $m \oplus m' = m \cup m'$ .

This is only possible under the assumption that any *merging* (as mandated by TMDM) is handled separately. Merging here is generic; only identical proxies are 'identified' and—as a map is a set—no redundant proxies are kept.

Merging, as defined in TMDM, includes then two aspects. First, a mechanism has to be found to state whether two proxies are to be *regarded to be about the same subject*. In a second step, these two proxies have to be actually merged into a new one.

We model merging with a partial function  $\bowtie: \mathcal{X} \times \mathcal{X} \mapsto \mathcal{X}$ . The fact that it is *partial* means that it may be applicable to some pairs of proxies (those to be merged) but not to others. We also require the function to be commutative as we understand merging to be commutative. Additionally,  $\bowtie$  should also be idempotent, as proxies merged with themselves should render exactly themselves.

Given now a map  $m$  and such a partial function  $\bowtie$ , we so define a merged version of the map combining unmergeable and mergeable into a new map:

$$m|_{\boxtimes} = \{x \boxtimes y \mid x, y \in m\} \quad (1)$$

As  $m|_{\boxtimes}$  is an operator, it does not modify  $m$ ; it only provides a particular view on the map given a particular merger. This may appear in some conflict with TMDM where merging is defined as "process applied to a topic map in order to eliminate redundant topic map constructs in that topic map" insinuating that the map is actually touched (which is not the case).

It is also worth noting that the concept of *merging two maps* actually does not exist here. In our context, maps can only be combined using the composition as defined above into a resulting map. That then can be viewed under different merging regimes. Again, the whole mechanism is purely functional.

### 3.3 Simple Mechanics

Now that the basic structure of maps is defined, we can introduce some means to navigate through a given map. Very simple is it to reach out for all *local keys* of a given proxy  $x$ :

$$x \downarrow = \{k \mid \exists v: \langle k, v \rangle \in x\} \quad (2)$$

*Remote keys* are all those keys where a given proxy *is* the value in another proxy:

$$x \uparrow_m = \{k \mid \exists y \in m: \langle k, x \rangle \in y\} \quad (3)$$

Obviously, this navigation has to occur in the context of a map  $m$ .

More interesting, though, is to retrieve not the key involvements of proxies, but *values for a particular key*  $k$

$$x \rightarrow k = \{v \mid \exists \langle k, v \rangle \in x\} \quad (4)$$

While certainly useful, a more effective version of such navigation is to honor subclassing of concepts when following a property. For this purpose we define a relation, *subclasses<sub>m</sub>*, which holds for two proxies  $c, c' \in \mathcal{X}$ , if there is another proxy  $x \in m$  in a map which actually says so:  $x \rightarrow \text{subclass} = \{c\}$  and  $x \rightarrow \text{superclass} = \{c'\}$ . We write *subclasses<sub>m</sub>*( $k, k'$ ) if  $k$  is a superclass of  $k'$ .

Naturally we build the reflexive and transitive closure, *subclasses<sub>m</sub>\** of this relation to honor whole chains of subclassing. Given that, we can extend above navigation to

$$x \rightarrow_m k^* = \{v \mid \exists \langle k', v \rangle \in x: \text{subclasses}_m^*(k', k)\} \quad (5)$$

asking for those values not only attached to the key we specify, but also respecting keys which are direct or indirect subclasses of the provided key.

As an example, let us assume that the proxy *organisation-code* is a superclass of *CRICOS-provider*. Then the expression

```
bond-university -> organisation-code*
```

would also render the values for the *CRICOS* code.

Honoring subclassing can also be done when following a key in such a way, that a given proxy *is* the value:

$$v \leftarrow_m k^* = \{x \in m \mid \exists \langle k', v \rangle \in x: \text{subclasses}_m^*(k', k)\} \quad (6)$$

This time we traverse the property keys *backwards* and get returned a set of proxies.

## 4 T<sup>+</sup> Path Expression

As it is one of the objectives of the T<sup>+</sup> model to provide a basis for more extensive TM processing, we need to define a more expressive language to define certain constellations of proxies and proxy values. This language is couched in a purely functional paradigm and is defined as path language over the set of all maps. As a complete semantic definition of the language can be found in (Barta & Salzer 2005), we can fall back here to a more informal presentation.

Accordingly, a path expression itself is a chain of postfix operators, some of which provide navigation from a certain point in a map, some of which do filtering on results. Others deal with invoking functions or providing constant values. Such a postfix chain can then be applied to maps.

To make this work consistently, postfixes do not operate directly on maps but always on a sequence of tuples of values. In such sequences, every tuple contains values from the underlying value sets (strings, integers, etc.). As also proxies are values, it is straightforward to interpret a map as a sequence with tuples containing only a single value (the proxy) each.

When we apply a path expression to a tuple sequence  $s$  we use the notation  $s \otimes_m p$ . We can drop the index  $m$  on the operator  $\otimes$  if we silently assume a context map with which we operate.

In the following we use as running example a tuple sequence listing universities and their vice chancellors:

```
[ < bond-university, dr-someone >,
  < qut-brisbane, dr-else > ]
```

### 4.1 Navigation

Applied to the above sample data, the path expression `<- instance -> class` would try to find all proxies where the *bond-university* proxy would play the role *instance* and then would try to find that proxy which plays *class* in that one. For *bond-university* the expected result would be probably *university*, for the other proxy in the tuple possibly *person*, so that the overall result would be

```
[ < university, person >,
  < university, person > ]
```

Complications arise when there are more than one results, or not a single one. The formalism takes an approach similar to SQL, in that tuples are *multiplied* accordingly. In our example we would see

```
[ < university, person >,
  < entertainment, person >,
  < university, person > ]
```

if we—absurdly—would assume that *Bond University* is also some form of entertainment establishment.

## 4.2 Projection

Another postfix is designed to select particular columns from an incoming tuple sequence. The postfix `pi_1` would select only the second column in the incoming tuple sequence. With this, it is also possible to create new tuples based on existing ones. A path expression

```
< pi_1, pi_0 -> staff-count >
```

for example, would flip the first two columns of an incoming tuple sequence, whereby the staff count property of the university is computed. All other columns are ignored.

## 4.3 Filtering

Projection also comes handy when results are to be filtered to satisfy a particular criterion, as in

```
[ pi_0 <- instance -> class = university ]
```

The semantics prescribes that the condition inside `[]` is evaluated for each individual tuple of the incoming tuple sequence. If we assume that it contains only one column, then  $\pi_0$  would select that value there. This as starting point, we would then navigate to all proxies which are its classes. This result (potentially a list) will be compared with `university` which is on the right-hand side of the comparison. Also there a path expression can be used, so the result being another tuple sequence.

If there is only one common tuple in these two sequences, then the comparison is successful and the tuple we had used as the basis for evaluating the filter is allowed to be in the overall result sequence. Otherwise it is suppressed.

Worth noting here is that this implements a form of *exists semantics*: If there exists a single commonality, then the condition evaluates to true. The path expression language also provides *forall semantics* via the inequality operator. In our example, the postfix

```
[ pi_0 <- instance -> class !=  
  university ]
```

would filter out all organisations which are not universities. Like above, the expressions to the left and the right of the comparison operator are evaluated on a single incoming tuple. Different to above, though, the condition itself is true if there is not a single common value in the respective subresults.

## 4.4 Sorting and Functions

As one would expect, tuple sequences can be sorted into *ordered tuple sequences*. This is quite straightforward; a *sort* postfix makes sure that the sequence content itself is maintained, only the tuples are ordered according a given ordering on tuples.

Also functions can be executed on path expressions. Returning to our example sequence containing organisations, we might want to apply the following postfix on our sample data:

```
integer-product (pi_0 -> staff-count,  
                pi_0 -> average-salary)
```

The expectation is to get a tuple sequence where each tuple contains only the product of the respective staff count and average salary. To achieve this, first all path expressions which are used as parameters are evaluated on each individual incoming tuple. In our case the result tuple sequences thereof will likely contain only one single value each. These values are organized into a parameter tuple which is passed into the function. Its result is a single value; if this whole procedure is repeated for every tuple in the incoming organisation tuple sequence, the overall result is another tuple sequence with one value for every organisation.

If the path expressions specified as parameters do result in multiple values, then this results in the several invocations of the function, one for each combination. Here a fair amount of magic is working behind the scenes.

## 4.5 Concatenation and Alternation

Given two path expressions, it is also possible to chain them, whereby we define as semantics of  $p \circ q$  simply via

$$s \otimes (p \circ q) = (s \otimes p) \otimes q \quad (7)$$

While this serves as a form of *and* between path expression, there is also an *or* combination in the form of *alternation*

$$s \otimes (p || q) = (s \otimes p) + (s \otimes q) \quad (8)$$

It simply means that the individual result tuple sequences are combined into one.

## 5 TMDM Disclosure

$T^+$  has the concept of *disclosure*. It is a specification which tells an application how it supposed to experience a given map. Effectively, it is an *ontological commitment* disclosing two things: First, which basic data types are used for the values in the map; and, secondly, which basic concepts every map contains and which rules they follow.

To *disclose a TMDM view*, we first have to consider the intrinsic data types used. They are strings, URIs, XML fragments. In the latest version it also allows applications to host arbitrary data as long as a URI exists to identify the data type. Still, all data is conceptually stored as *text* implying that applications possibly need functionality to convert from and to text.

To give an application the impression that it deals with a TMDM instance, a  $T^+$  disclosure has to detail the rules how each information item in TMDM is effectively mapped onto one (or more) proxies in an equivalent  $T^+$  map.

On the whole, this is a lengthy process (Barta & Heuer 2005); one illustrative example is that of TMDM association items. Each `assoc` has exactly one type component `assoc.type`. Assuming that this type is represented by a proxy `t` and that association itself by `a`, we explicitly model the *instance-class* relationship in  $T^+$  with a dedicated proxy:

```
{ instance => a, class => t }
```

This procedure is completed for all other TMDM information items, whereby special provision for scope, subject addressing and indication are made.

## 6 TMQL

The upcoming query language for TMs (TMQL) will be aligned conceptually with the abstraction of maps provided by TMDM. In the following, first a short overview demonstrates the various layers of syntax as they have been introduced to address a wider developer community.

After sketching the language, it can be shown how to transform the individual language layers into the bottom layer, that of TMQL path expressions. To define their semantics, they are further mapped into T<sup>+</sup> path expressions.

### 6.1 Surface Syntaxes

#### 6.1.1 SELECT Style

The first syntax flavor follows more the syntactic conventions of SQL. A query expression

```
SELECT $uni
FROM file:test.xtm
WHERE
    is-located-in ($uni : object,
                  sydney : location) ,
    $uni is-a university
```

would first consume the topic map stored at the URI `file:test.xtm`, would then try to identify in this map all associations of type `is-located-in` where `sydney` plays the role `location` and would then bind the role player of `object` to the variable `$uni`. In a next step the query processor would check whether the topic currently bound to `$uni` is a (direct or indirect) instance `university`. If so, the binding will be kept, otherwise it will be discarded.

This process is repeated whereby all these bindings are collected. The overall result (a sequence of variable bindings) is returned to the calling application. As any number of variables can be named in the `SELECT` clause, such query expressions always return sequence of values.

#### 6.1.2 FLWOR Style

The very same query can also be achieved with another style, FLWOR:

```
FOR $uni IN file:test.xtm
WHERE
    is-located-in ($uni : object,
                  sydney : location) ,
    $uni is-a university
RETURN
    ($uni)
```

Again, the provided URI will be used to resolve the TM content. The `FOR` loop would then iterate (conceptually) over all information items in the map and would select those which would satisfy the condition in the `WHERE` clause. If that is the case - and this is checked exactly as with SQL expressions - then the `RETURN` clause is evaluated.

In our case that `RETURN` clause generates one value tuple (consisting of a single value which is bound to a university topic). As the process is repeated, then the overall result is again a tuple sequence.

What makes FLWOR expressions more flexible, is that with them not only tuple sequences can be constructed, but also XML content and even TM content. The former is relevant for those situations where a TM query engine is used in an XML application server context. The latter comes handy when TM knowledge has to be transformed into other TM content whereby incoming and outgoing ontologies differ. For the following discussion we ignore this feature of TMQL.

#### 6.1.3 Path Expressions

The third TMQL flavor is more primitive and follows a path expression paradigm:

```
file:test.xtm // university
    [ . <- object
      [ * is-located-in ]
      -> location
      = sydney ]
```

It is evaluated from the left to the right. Like before, first the topic map content is sourced. That is then interpreted as an unordered sequence of information items.

The postfix `// university` finds all instances of `university` and also these are organized now into a sequence. This sequence is filtered by using the condition wrapped into `[]` brackets. This condition is evaluated on each individual university topic: First it will be checked whether such university plays the role `object` somewhere in an association. If such is found, yet another nested condition `[ * is-located-in ]` makes sure that only those associations are used further which are instances of `is-located-in`. Finally, for all such associations it is checked whether they contain a role of type `location` and whether the player of such role is `sydney`.

Path expressions can return arbitrary tuple sequences, not just sequences of single values:

```
file:test.xtm // university
    < . -> staff-count, . -> name >
```

Obviously, path expressions are best suited for rather short queries as they are to be expected in a number of application situations and in command-line oriented applications. Still, their readability and manageability suffer quickly with increasing query complexity, especially since they do not contain any mechanism to introduce and use variables.

## 6.2 Surface Canonicalization

Along with the architecture of the language, the process to define a formal semantics is multi-layered. First we will map all query expressions according to the `SELECT` syntax onto expressions following the FLWOR syntax. Next, we show how every expression according to the FLWOR syntax can be flattened into one following path expressions.

All these transformations are only structural and are performed on an abstract syntax tree which we assume that any query expression is parsed into at the start of the process. Notationally, we use here `[]` to indicate optional parts, `()` to group parts and `<>` to wrap things which may

appear any number of times, separated by commas. Terminals are enclosed with apostrophies, non-terminals are not. The syntax tree then would contain a (labelled) node for every non-terminal and one for each terminal.

### 6.2.1 Select

Any SELECT expression follows the structure

```
select-expression -> select-clause
                    [ from-clause ]
                    'WHERE' exists-clause
select-clause     -> 'SELECT'
                    < value-expression >
```

The parse tree transformations are described via a mapping ==> which takes a particular subtree of the syntax tree and replaces it with another tree. As a first example, the following transformation provides a default value for the topic map to be queried if a SELECT query expression does not explicitly name one:

```
select-clause 'WHERE' exists-clause
==>
```

```
select-clause 'FROM' '%_'
              'WHERE' exists-clause
```

Obviously, the whole SELECT clause and the whole condition in the WHERE clause are copied verbatim when a matching subtree is going to be transformed. What is added, is a FROM clause using the special variable %\_ where the language—by default— expects the queried topic map to be bound to.

The problem we are ignoring here is that a FROM clause must follow the syntax

```
from-clause     -> 'FROM' tm-content
```

and not simply

```
from-clause     -> 'FROM' '%_'
```

To fix this, we silently assume that the right-hand side of our transformations is always parsed according to the TMQL syntax. With that, we can keep the notation concise.

To map SELECT expressions into FLWOR expressions we use the following mapping:

```
'SELECT' < value-expression >
'FROM' tm-content
'WHERE' exists-clause
==>
'FOR' < variable 'IN' tm-content >
'WHERE' exists-clause
'RETURN' '(' < value-expression > ')'
```

Much in this transformation is reshuffling of content. The list of value expressions moves from the SELECT clause into a RETURN clause whereby the () pair makes sure that we return tuples (and not other content). The WHERE clause is not changed at all, only a FOR clause is added.

The mapping introduces explicitly those variables which are implicitly used in the SELECT version of the query. The variables are all taken from the *exists-clause*, acknowledging the fact that all variables there are to be understood to be existentially quantified.

### 6.2.2 Where

One obvious step to convert parts of FLWOR expressions into TMQL path expressions is to interpret a WHERE clause as a filter:

```
'WHERE' path-expression
```

==>

```
'[ path-expression ]'
```

Hereby we assume that we already have succeeded to convert an entire exists clause into a path expression. To arrive there we first have to convert the exists clause itself

```
exists-clause -> 'SOME' < variable-iteration >
                'SATISFY' bool-expression
```

into a path expression equivalent. Obviously, the boolean expression in the SATISFY clause has to be converted into a path expression first; but once this is done it is obviously to be used as filter:

```
'SATISFY' path-expression
```

==>

```
'[ path-expression ]'
```

To feed the filter with content to be tested, the variable iteration is made explicit and the variable itself is effectively removed. So, for example

```
SOME $a IN // university, $b in // location
```

becomes eventually

```
< %_ [ . class::university ] ,
    %_ [ . class::location ] >
```

which effectively represents a tuple sequence where every tuple contains exactly two values, the first one being a topic of instance *university*, the other one of *location*. Since both 'columns' are independent, all possible combinations are generated. The `class::` prefix signals only one of the possible axes along which one can navigate through maps.

From the above it becomes clear that

```
'SOME' path-expression
```

==>

```
path-expression
```

can be used to get rid of the SOME keyword.

### 6.2.3 Association Templates

One of the boolean conditions which can be used are association templates, like the one we have used earlier

```
is-located-in ($uni : object,
               sydney : location)
```

While they can be interpreted as predicates, they can also be seen as path expressions which are supposed to find all associations of the given type and with the given configuration of role types and players:

```
%_ // is-located-in [ . -> object = $uni ]
[ . -> location = sydney ]
```

The first thing to observe is that these two expressions are not exactly equivalent. In the predicate interpretation we tacitly assume that all relevant associations which match have *only* two roles, one for `object` and one for `location`. In the interpretation using path expressions we use these two roles inside filters; the found association items may well contain more roles.

To disambiguate these two cases (allow more roles to exist or not), TMQL has an additional syntax:

```
is-located-in ($uni : object,
               sydney : location,
               ...)
```

The ellipsis `...` should indicate that other roles may exist. In this form, the above path expression truthfully represents the intended semantics. For the other, restricted case (which we do not detail here) we have to explicitly encode into the path expression that no other roles except those mentioned may exist in a matched association.

The second observation is that path expressions actually do not contain references to any variables. The values over which variables iterate are provided at the outside, again via path expressions.

The general case of translating association templates into path expressions is then covered by the mapping

```
tm-reference '(' < path-expression > ','
             '...' ')'
```

==>

```
'%' '//' tm-reference < path-expression >
```

whereby we again assume, that the roles inside already have been transformed

```
path-expression_1 ':' path-expression_2
```

==>

```
'[' ']' path-expression_1 '='
      ']' path-expression_2 ']'
```

### 6.3 T<sup>+</sup> Downtranslation

By design, TMQL path expressions and T<sup>+</sup> path expressions are structurally similar, so that the mapping is mostly straightforward.

### 6.4 Toplevel Semantics

When an arbitrary query expression is about the evaluated, we expect the parse tree to expand under a node labelled *query-expression* (that is the start non-terminal for the TMQL grammar). In this process, the environment will also pass in values bound to variables to the query processor. This context is simply captured as a partial function *c* from variables to values.

To evaluate a query expression in such a given context the following steps have to be taken. First the the *query-expression* is transformed according

to the surface canonicalization until it contains only path expression subtrees, conditional (*if-then-else*) subtrees and subtrees which invoke a function.

In this process most of the variables are removed, replaced by iterators over map items. Only global variables remain and these are then replaced by the values in the passed-in context. It is an error if variables remain in the query expression.

The resulting TMQL path expression is then mapped into T<sup>+</sup> path expressions whereby all values are also transformed. While primitive values remain as they are, all TMDM items are translated into their T<sup>+</sup> pendant. Only then the T<sup>+</sup> path expressions is evaluated according to its evaluation semantics. The result—a tuple sequence—is the overall result. It is yet unresolved how other content (XML or TM content) can be described.

During this evaluation the abstract processor may encounter also a subtree labelled with *if-then-else*. If that is the case, it will first evaluate the condition attached to this node and will follow the *then* branch iff the evaluation of the condition has resulted in any non-empty content. Otherwise it will continue with the evaluation of the *else* branch.

Another subtree which may be encountered is one containing a function invocation which itself follows the syntax

```
function-invocation -> tm-reference
'(' < value-expression > ')'
```

The value expressions are canonicalized and evaluated as described so far. The result of all value expressions together is then interpreted as a tuple sequence, which implies that for the function invocation not only one set of parameters is available, but as many sets as are tuples in the sequence.

The *tm-reference* allows to uniquely identify the function itself; it may be an internal, language-intrinsic or an external, imported one. For each tuple in the parameter tuple sequence this function is now invoked. It may return simple, or compound values. In any case these are collected into a result tuple sequence.

## 7 Future Work

While this presentation might give an impression how the final standards landscape might be architected—namely using T<sup>+</sup> as fundamental model, mapping TMDM instances onto T<sup>+</sup> maps and using T<sup>+</sup> path expressions as basis for the formal semantics of TMQL—this is neither settled or complete.

One current area of research is to finalize a first complete public draft of the TMQL semantics. In there are still various unresolved issues. This development has to be in lieu with the specification of TMQL itself. Also the existing TMDM disclosure has attracted criticism which resulted in various attempts to reformulate it.

Additionally, there are upcoming TM standards which are not covered here. One of them is TMCL (constraint language), which might have a similar scope as OWL has for the RDF stack. Currently, efforts are underway to map TMCL onto T<sup>+</sup> path expressions as the latter can be used to express constraints. While viable, a simpler and

more robust avenue might be to map TMCL expressions onto TMQL expressions. This, of course, only works if TMQL has the necessary expressivity.

Another development is the introduction of a shorthand notation for TMs which should complement XTM, the existing, more canonical serialization of TM content into XML. Obviously, such a language must specify how particular expressions are to be deserialized into a TMDM instance. What is interesting, though, is that assertions of factual TM data in such a language can be written in ways very similar to TMQL queries. The precise connection is yet unknown.

And finally, on a different front, the Topic Maps landscape has to be positioned more precisely against that for RDF. Various attempts exist, a recent one in (Cregan 2005), but most use TMDM as starting point. It may be worth trying to use  $T^+$  instead.

## References

- Barta, R. & Heuer, L. (2005), 'A TMDM Disclosure using Tau'.  
**URL:** <http://astma.it.bond.edu.au/junk/tmdm-disclosure.pdf>
- Barta, R. & Salzer, G. (2005), 'The Tau model, formalizing Topic Maps'.  
**URL:** <http://crpit.com/confpapers/CRPITV43-Barta.pdf>
- Cregan, A. (2005), 'Building topic maps in owl-dl'.  
**URL:** <http://www.mulberrytech.com/Extreme-/Proceedings/html/2005/Cregan01-/EML2005Cregan01.html>
- Garshol, L. M. (2004-07-22), 'A proposed foundational model for Topic Maps'.  
**URL:** <http://www.mulberrytech.com/Extreme-/Proceedings/html/2005/Garshol01-/EML2005Garshol01-toc.html>
- Garshol, L. M. & Moore, G. (2003-11-02), 'ISO 13250-2: Topic Maps - data model'.  
**URL:** <http://www.isotopicmaps.org/sam/>
- Garshol, L. M. & Moore, G. (2004-01-25), 'ISO/IEC JTC1/SC34, Topic Maps - XML syntax'.  
**URL:** <http://www.isotopicmaps.org/sam/sam-xtm/>
- Kipp, N. A. (2003), 'A mathematical formalism for the Topic Maps reference model'.  
**URL:** <http://www.isotopicmaps.org/tmrm-/0441.htm>
- Newcomb, S. R., Hunting, S., Algermissen, J. & Durusau, P. (2003), 'ISO/IEC JTC1/SC34, Topic Maps - reference model, editor's draft, revision 3.10'.  
**URL:** <http://www.isotopicmaps.org/tmrm/>
- O. Lassila and K. Swick (1993), *Resource Description Framework (RDF) model and syntax specification, Technical report, W3C, Camo AS*.  
**URL:** <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.html>
- Pepper, S. (2000), 'The TAO of Topic Maps'.  
**URL:** <http://www.gca.org/papers/xmleurope-2000/papers/s11-01.html>