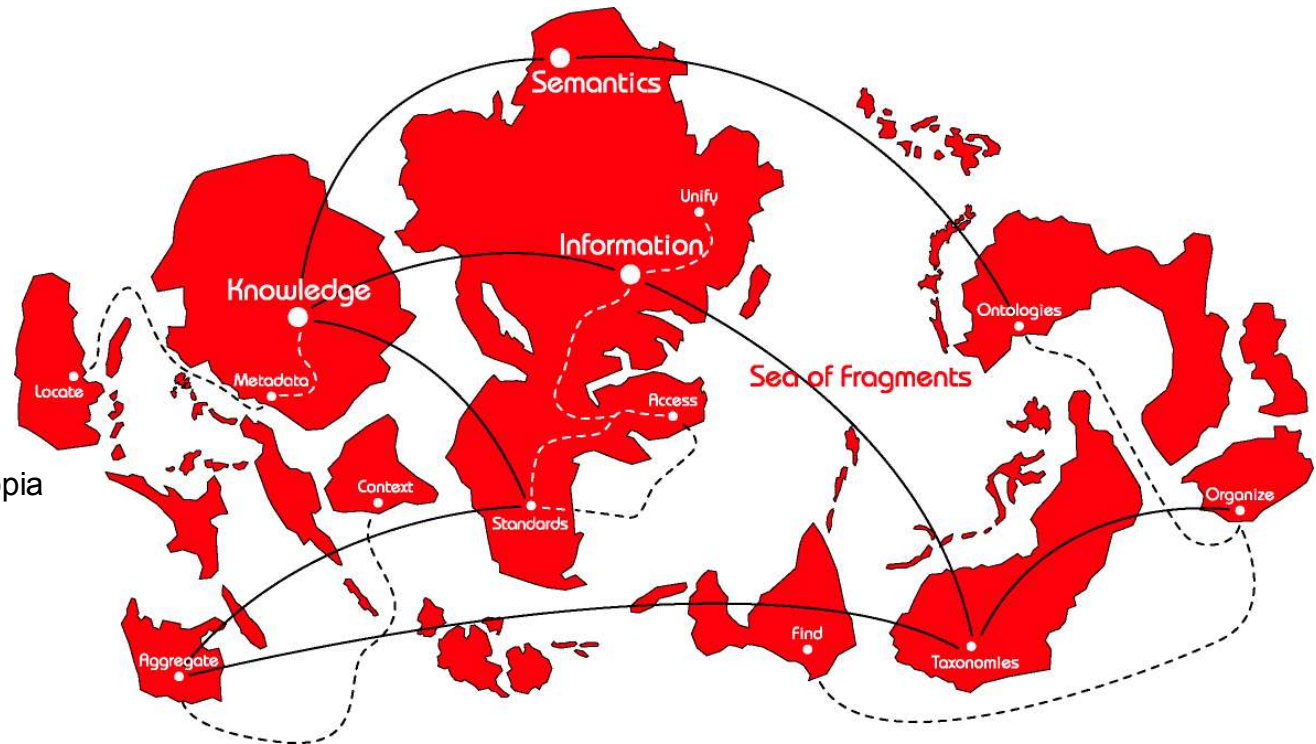


TMQL

An introduction



Lars Marius Garshol

Development Manager, Ontopia
 <larsga@ontopia.net>

2005-04-19

Agenda

- **A little background**
- **A tutorial in today's TMQL**
 - path expressions
 - SELECT expressions
 - FLWR expressions
- **Discussion**

Some background



TMQL? What? Why? When?

What is TMQL?

- The purpose of the TMQL work is to create a standard query language for topic maps
- Currently, the topic maps standard only standardizes *data*
- You can move your data between systems, but not your *application*
- Support for TMQL in topic map systems will provide some support for application portability

Why not just an API?

- **APIs can only be used by hard-core developers**
- **APIs are programming-language specific**
 - the failure of the DOM API for XML proves this
- **APIs are hard to use**
 - what would you prefer to use? XSLT or DOM?
- **APIs scale poorly for data access**
 - an API to TMDM must provide fine-grained access to all aspects of the model
 - it must also live with uncontrolled references to stateful data objects
 - an API cannot provide optimizations and alternative access strategies

What should TMQL be able to do?

- **Return data from the topic map based on criteria**
 - single values
 - collections of values
 - values arranged in columns and rows
 - constructed XML values
 - topic map fragments
- **Usages**
 - building presentation logic
 - returning fragments from topic map servers
 - used in creating topic map applications of all sorts
 - updating the topic map (in part 2!)

The state of the TMQL work

- **A requirements document has been published**
- **A set of use cases has been published**
- **Solutions to use cases in four proposal languages collected**
- **Workshop held to evaluate proposals**
 - guidelines for first working draft provided
- **First working draft published 2005-02-18**

The purpose of this presentation

- **Firstly, to teach those who are interested TMQL as it is now**
- **Secondly, to let those who are interested shape TMQL as it will be**
 - the idea is that feedback on the language will help us improve it
- **In other words:**
 - let us hear your feedback!
 - if you like something, say it!
 - if you don't like something, say it!
 - if you don't understand something, shout it!
 - and so on :-)

larsbot: I'm presenting my view and my understanding here, not Robert's. I've tried to speak for both of us, but probably don't succeed entirely. Personal asides are written like this

The current TMQL

- **Has three sub-languages**
 - path expressions
 - SELECT expressions
 - FLWR expressions
- **The sub-languages build on each other**
 - path expressions can be used in SELECT expressions
 - path expressions and part of SELECT expressions used in FLWR expressions
- **Result is a language with a small set of core constructs**
 - this means that a lot of the syntactic variation matters much less to an implementation than it may seem
 - however, it has to seem simple to the users, too

TMQL tutorial



Path expressions

Path expressions?

- Path expressions are rather like XPath
- They start from a value, generate new values with "steps", and finally produce a value or collection of values
- Conditions can be used to constrain values generated
- This part is essentially AsTMA? and TMPPath reborn

How path expressions work

- **foo**
 - "foo" generates values, maybe one, maybe more
- **foo / bar**
 - the "/ bar" takes a set of values from "foo" and produces a new set of values from it
 - you could think of it as bar(foo), but it's a bit more complex, as you'll see
- **foo / bar / baz**
 - this is just simple chaining (like baz(bar(foo)))
- **foo / bar [quux] / baz**
 - the [quux] is a constraint; it removes values from what bar produces
 - this is where the analogy with functions starts breaking down

larsbot: I'm using the term "constraints" here, but Robert (and XPath) call these "predicates". We also use "predicate" for something else, however

The simplest possible path expressions

- **opera**
 - returns the topic type "opera" by ID
- **i'http://psi.ontopia.net/music/#opera**
 - returns the topic type "opera" by subject identifier

Finding the name of a topic

- **il-tabarro / bn**
 - finds all base names of the topic il-tabarro: "Il Tabarro", "The Cloak"
- **%m // opera / bn**
 - finds all base names of all opera topics

Finding the English name

- **il-tabarro / bn [@en]**
 - returns only the base name in the English scope ("The Cloak")
- **%m // opera / bn [@en]**
 - finds all base names of all opera topics that are in the English scope
 - if an opera has no name in that scope none of its names will make it
 - note that the opera itself is not returned

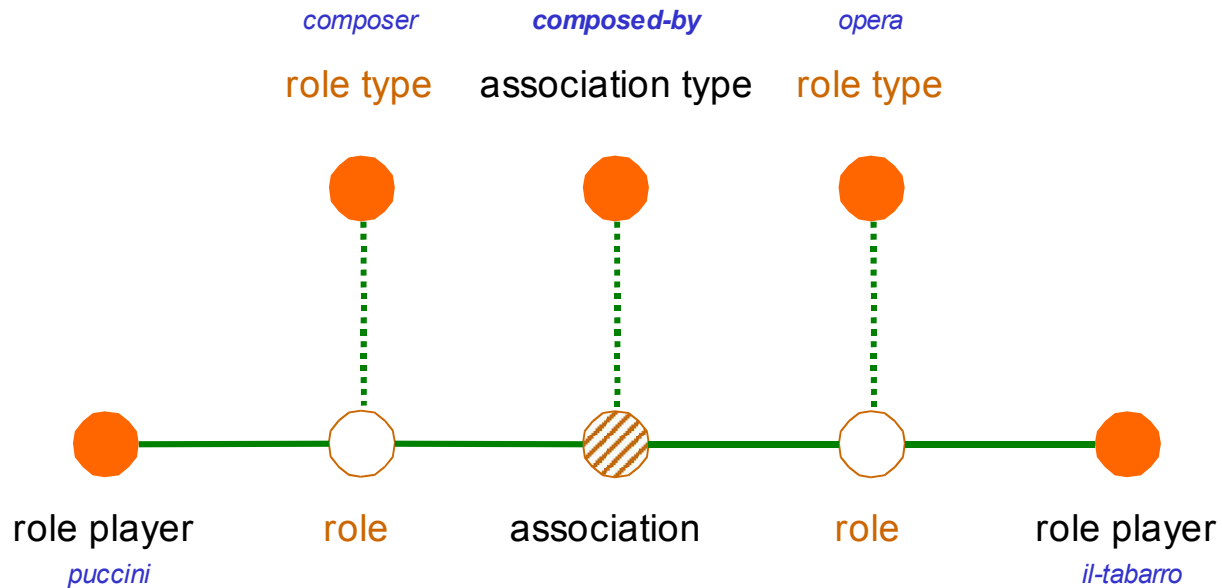
Finding all operas

- **%m // opera**
 - %m must be a variable containing the topic map
 - the "// x" operator filters a collection, producing only instances of x
- **%m [* opera]**
 - this is actually the same expression with a different syntax

Finding the libretto

- **il-tabarro / oc**
 - returns all external occurrences of il-tabarro
 - we want only the librettoes, however
- **il-tabarro / oc [*libretto]**
 - this only gives us the occurrences of type libretto

The structure of associations



Finding the composer (step by step)

- **il-tabarro -> opera**
 - this gives us all associations where il-tabarro plays the role of "opera"
 - the "->" operator is a little special; it goes via the association roles, but produces the associations rather than the roles
 - that's too wide; we want only the associations of type "composed-by"
- **il-tabarro -> opera [* composed-by]**
 - this solves that problem, by adding a predicate filtering by type
 - however, we didn't want the associations, but the topic at the other end
- **il-tabarro -> opera [* composed-by] / composer**
 - the last step finds the roles within the association that are of type "composer", then produces the topics playing those roles
 - **note:** the result of "/ composer" is not the roles of type composer, but the topics that play roles of type composer in the associations produced by the previous step

larsbot: some asymmetry here in that filtering by type is done implicitly by "/ foo" on associations, but not on topics

More complex combinations

- **%m // composer [. -> person [* born-in] / place = italy]**
 - finds composers born in italy
 - the [...] after composer is a condition on the composers included in the result
- **%m // composer [. -> person [* born-in] / place != italy]**
 - composers not born in Italy
- **%m // composer [! (. -> person [* born-in] / place)]**
 - composers not born anywhere
- **%m // opera [. / rd [* premiere-date] < 1900]**
[. -> opera [* composed-by] / composer = puccini]
 - operas by Puccini premiered before 1900

SELECT expressions



A bit more power

Predicates

- **This sub-language uses predicates for the queries, like tolog**
- **In one sense it's tolog fitted on top of the path expressions**
- **General workings:**
 - predicates generate bindings for *variables*
 - the result of a query is therefore all possible combinations of values for the variables in the the query that make the conditions in the query true
 - you can think of this as a *table of results* with one column per variable
 - how this is achieved is up to the processor
 - however, you can think of it as happening predicate by predicate from start to finish

Finding the composer (in one step)

- **composed-by(il-tabarro : opera, \$COMPOSER : composer)**
 - this is a *dynamic association predicate*
 - every association type can be used as a predicate
 - the meaning of this one is: find every topic that plays the role of "composer" in a "composed-by" association where il-tabarro plays the role of "opera"
- **composed-by(il-tabarro, \$COMPOSER)**

Finding the operas

- **composed-by(\$OPERA : opera, puccini : composer)**
 - here we start from the composer (since he is given explicitly) and find all his operas (since the opera is a variable)

Finding all combinations

- **composed-by(\$OPERA : opera, \$COMPOSER : composer)**
 - here we find all the opera/composer combinations (both are variables)

Is it true that...?

- **composed-by(il-tabarro : opera, puccini : composer)**
 - equivalent to asking "is it true that puccini composed il-tabarro?" (no variables)
 - returns a table with no columns and 0/1 rows
 - 0 = false (he didn't)
 - 1 = true (he did)

Finding all operas by the same composer

- **composed-by(il-tabarro : opera, \$COMPOSER : composer),
composed-by(\$COMPOSER : composer, \$OPERA : opera)**
 - this finds all operas by the composer of "il-tabarro", including "il-tabarro" itself
 - first the composer of "il-tabarro" gets bound into \$COMPOSER
 - then the second predicate is evaluated, starting from the composer, finding all operas composed by this composer
- **composed-by(il-tabarro : opera, \$COMPOSER : composer),
composed-by(\$COMPOSER : composer, \$OPERA : opera),
\$OPERA != il-tabarro**
 - this query excludes "il-tabarro" by adding an extra condition

Composers who wrote ...

- **SELECT \$COMPOSER**
WHERE
 - composed-by(\$COMPOSER : composer, \$OPERA : opera),**
 - based-on(\$OPERA : result, \$WORK : source),**
 - written-by(\$WORK : work, shakespeare : author)**
- the query binds three variables, but the SELECT keeps only \$COMPOSER

English names of Puccini operas

- **SELECT \$OPERA / bn [@en]**
WHERE
 composed-by(puccini : composer, \$OPERA : opera)
 - here we use a path expression in the SELECT clause in order to get the right base name
- **SELECT \$NAME**
WHERE
 composed-by(puccini : composer, \$OPERA : opera),
 \$NAME = \$OPERA / bn [@en]
 - here the '=' predicate is used to achieve the same thing inside the query itself

larsbot: the = predicate is strictly speaking not necessary in tolog, but has proven itself to be very useful. Not sure if this is the case in TMQL.

Puccini operas before 1900

- **SELECT \$opera / bn**
WHERE
 \$opera / rd [* premiere-date] < 1900,
 composed-by (\$opera : opera, puccini : composer)
 - here we use path expressions to get hold of the occurrence values
 - this can alternatively be done with a *dynamic occurrence predicate*
- **SELECT \$opera / bn**
WHERE
 premiere-date(\$opera, \$date), \$date < 1900,
 composed-by(\$opera : opera, puccini : composer)

larsbot: we are not sure whether to keep dynamic occurrence predicates or not. path expressions can do the same thing

Finding all operas

- **\$OPERA : opera**
 - the ':' is a built-in predicate for topic types
 - in tolog this was "instance-of(\$OPERA, opera)?"

More complex combinations

- **\$C : composer, born-in(\$C : person, italy : place)**
 - composers born in Italy
- **\$C : composer, not(born-in(\$C : person, italy : place))**
 - composers not born in Italy
- **\$C : composer, born-in(\$C : person, \$P : place), \$P != italy**
 - composers not born in Italy, take 2

Alternatives

- **\$C : composer, { born-in(\$C : person, italy : place) |
born-in(\$C : person, sweden : place) }**
 - composers born in Italy or Sweden
- **\$C : composer, born-in(\$C : person, \$P : place),
{ \$P = italy | \$P = sweden }**
 - same, formulated differently
- **\$C : composer, born-in(\$C : person, \$P : place),
in(\$P, italy, sweden)**
 - yet another way to do it

larsbot: 'in' predicate suggested by Geir Ove; never made it into official tolog, but might be worth considering

Optional clause

- **\$C : composer, { homepage(\$C, \$H) }**
 - returns all composers with their home page, if they have one, and null if they don't
- **SELECT \$C, \$C / oc [* homepage] WHERE \$C : composer**
 - doing the same in the select clause

larsbot: we're not sure about these, since path expressions in SELECT do much the same thing

Alternative syntax

- **SELECT \$COMPOSER**
WHERE composed-by(\$COMPOSER : composer, \$OPERA : opera)
AND based-on(\$OPERA : result, \$WORK : source)
AND (written-by(\$WORK : work, shakespeare : author) OR
written-by(\$WORK : work, goethe : author))
 - the difference is that we use AND instead of "," and OR instead of "{ ... | ... }"

larsbot: no decision taken on alternatives here

Non-existential queries

- **\$CLUSTER : cluster,**
part-of(\$MACHINE : part, \$CLUSTER : whole),
is-down(\$MACHINE)
 - finds all (machine, cluster) combinations where the machine is down
 - however, we want the clusters that are down because all machines in them are down
 - how to do that?

larsbot: all tolog queries are existential in the sense that they ask "are there any \$Xs and \$Ys so that this is true?" and there is enough with *one* (\$X, \$Y) for this to work.

This query is non-existential because it is asking "is there any \$X where all the \$Ys are like this?". The language is geared towards the other kind of query, which makes this type of queries difficult

Non-existential queries (the hard way)

- **\$CLUSTER : cluster,**
not(part-of(\$MACHINE : part, \$CLUSTER : whole),
not(is-down(\$MACHINE)))
 - finds every cluster [where there is no machine in the cluster [that is not down]]
 - in other words: clusters where all machines in the cluster are down
 - this may be easier to see if formulated like this: we're asking "is there any \$X so that there are no \$Ys that are not like this?"
 - this works, but requires substantial mental gymnastics on the part of query authors not deeply versed in logic (ie: 99.9% of all query authors...)

Non-existential queries (the easy way)

- **\$CLUSTER : cluster,**
EVERY part-of(\$MACHINE : part, \$CLUSTER : whole)
SATISFIES is-down(\$MACHINE)
 - this finds [clusters [where every machine in the cluster [is down]]]

Features from tolog

- **SELECT \$composer, count(\$opera) WHERE ...**
 - counts the number of operas per composer
- **SELECT \$composer, count(\$opera) WHERE ... ORDER BY \$opera**
 - sorts by the number of operas
- **SELECT ... WHERE ... LIMIT 5 OFFSET 2**
 - same as in SQL

larsbot: we want similar capabilities, but haven't really settled how

Inference rules

- **inspired-by(\$COMPOSER, \$AUTHOR) :-
 composed-by(\$COMPOSER : composer, \$OPERA : opera),
 based-on(\$OPERA : result, \$WORK : source),
 written-by(\$WORK : work, \$AUTHOR : author).**

inspired-by(\$COMPOSER, shakespeare)

larsbot: Robert is skeptical of this, and doesn't want to commit to any particular kind of inferencing

FLWR expressions



Even more power

FLWR expressions

- **Directly inspired by XQuery and the way it uses XPath**
- **Uses predicate lists in the same way as SELECT expressions**
- **How it works**
 - (for | let)* where? return
 - in other words, for and let can be repeated in any order
 - where is optional
 - return is not optional

Using just RETURN

- **RETURN opera**
 - gives us the topic "opera"
- **RETURN opera / bn**
 - gives us the base names of the topic "opera"

Using FOR

- **FOR \$opera in %m // opera RETURN \$opera**
 - gives us all operas
 - path expression produces all operas, each one gets a binding for \$opera, and the RETURN is evaluated once for each binding

Using WHERE

- **FOR \$topic in %m WHERE \$topic : opera RETURN \$topic**
 - first produces all topics, then filters away those which are not instances of opera, and finally returns the opera instances
 - ie: same as previous query
- **FOR \$opera in %m // opera**
WHERE composed-by(\$opera : opera, puccini : composer)
RETURN \$opera
 - returns operas composed by Puccini

Using LET

- **FOR \$opera in %m // opera**
LET \$composer := puccini
WHERE composed-by(\$opera : opera, \$composer : composer)
RETURN \$opera
 - all operas composed by Puccini

larsbot: LET lets us bind temporary values, but the predicate list is good at that anyway. Not clear that LET is actually needed

RETURN is like SELECT (almost)

- **FOR \$opera in %m // opera**
WHERE composed-by(\$opera : opera, \$composer : composer)
RETURN (\$opera, \$composer)
 - returns all operas with one composer each
 - WHERE cannot generate new tuples, only remove them
 - it can bind variables, but only to one value

RETURN is like SELECT (2)

- **FOR \$opera in %m // opera**
FOR \$composer in %m // person
WHERE composed-by(\$opera : opera, \$composer : composer)
RETURN (\$opera, \$composer)
 - now we get all opera/composer combinations
 - the two FORs generate all possible combinations
 - the WHERE then removes the ones that aren't connected

RETURN can do more than SELECT

```
<rss>
```

```
  <title>Recent opera premieres</title>
```

```
  <link>http://...</link>
```

```
{ for $opera in %m // opera
```

```
  let $premiere = $opera / oc [* premiere ]
```

```
  order by $premiere desc
```

```
  return
```

```
    <item>
```

```
      <title>{$opera / bn}</title>
```

```
      <link>http://...opera.jsp?id={id($opera)}</link>
```

```
      <description>Premiere on {$premiere}</description>
```

```
    </item>
```

```
  } </rss>
```

larsbot: the id function is just conjecture at this point. we need to be able to do this somehow, but not sure how yet

RETURN can make topic maps

- **But we're not yet sure how :-)**
 - this functionality will be added, but we've been too busy with other things so far