XQuery topic tools - concept, user interface, development framework

Hans-Jürgen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

This paper defines the concept of topic tools, which are command line tools providing a single point of access to a range of functionality. Topic tools conform to a generic model of invocation syntax and basic tool behaviour, concerning user assistence, error diagnostics and invocation reuse. The paper proposes a comprehensive model of the user-perspective - syntax and behaviour - and it introduces a simple development framework making the creation of XQuery topic tools simple and fast. The support offered by the framework includes code generation and the use of a message interface which cleanly isolates the application code from user input and gives it access to validated and augmented information, rather than the raw data of user input. Key properties of framework-based topic tools are early availability, extensibility, user convenience, behavioural consistency and reliability based on very thorough and fully automated input validation.

Table of Contents

Introduction

XQuery ([4]) is a W3C defined language for processing XML and other information resources. The name is a misnomer, as XQuery is a general information processing language, rather than a query language. XQuery certainly offers excellent support for querying XML data, but this support should be viewed as a prominent feature of the language, not as its scope.

XQuery has an amazing conceptual simplicity. It is an expression language – everything is an expression, a query is an expression tree. An expression is a shape which has a value, and the value of an XQuery expression is a sequence of items, where each item is either a node, an atomic value, or a function. Nodes are the foundation of a generic model of navigation, as they are interconnected by node properties, and these connections can be used as "roads" along which to move – the XPath axes (e.g. child, descendant, parent and ancestor). The XPath functions `doc()` and `collection()` enable URI based access to local and remote XML resources, and as these are modelled as node trees, and as node trees can be traversed using those axes, XQuery programs operate in a logical space of information which is homogeneously structured (everything is a node) and which exposes any grain of content of any accessible XML resource (accessible via `doc()` or `collection()`), making it individually addressable. Powerful operations are available not only for addressing, but also for aggregating, transforming and evaluating those ubiquitous units of content. These capabilities, combined with utmost conceptual simplicity (expressions, values, full stop) might induce us to refer to XQuery not as a query language, but simply as *the XML expression language* – call it XMLe. And perhaps we should regard XML syntax, the XML data model and XMLe as integral parts of a single, unified information architecture which constitutes the deeper meaning of the three letters XML (XML = XMLs + XMLm + XMLe).

XQuery is currently a niche language. Only a tiny fraction of developers has a solid understanding of its capabilities, let alone the skills to use them. It might even be the case that acknowledged experts of XML do not have a real understanding of XQuery and that their view of XML technology and the scope of its usefulness is profoundly affected by this lack of understanding and experience.

Comparing the high capabilities of XQuery and its low adoption by developers, one may conclude that the industry is wasting a significant potential. To put it differently: the potential of XQuery is still locked. Any approaches designed to unlock the potential – facilitating the use of XQuery somehow and in some realm - might deserve special interest, as the possible gains are high. One such approach is RESTXQ ([3]), which simplifies the XQuery-based implementation of web services. This paper introduces another approach, `ttools`, a framework supporting the development of XQuery-based command line tools which are powerful, versatile and designed for continuous extensibility. The approach is based on the concept of a *topic tool*. A topic tool is a program providing a single command line interface to a range of functionality – the available (and continuously growing) functionality related to a "topic". This paper defines the concept, proposes a generic model of the user perspective and introduces a development framework enabling the fast and easy development of topic tools. It should be noted right at the start that the framework is extremely lightweight – 100% XQuery implemented, so that neither the use nor the development of topic tools requires any components apart from an XQuery processor.

XQuery for command line tools

XQuery is well-suited for the rapid development of divers and powerful command line tools, enabling the evaluation of distributed information sources. Key reasons for this potential is the navigational power of XPath and the uniformity of a data model which allows seamless integration of navigation and construction. But this very efficiency of XQuery can seduce the developer to create - at an amazing speed - an amazingly unsatisfactory situation: a heap of command line tools inconsistently named, and using input parameters with inconsistent names and semantics. Each of these tools, inspected in isolation, might be very useful, and yet the sum is difficult to use and to maintain. The longer this undisciplined development continues, the greater the loss of efficiency on both sides – the developer's and the user's. Many tools are at length abandoned (by users and by the developer), as it becomes too difficult to remember what exactly is available and how to use it, precisely. Wheels are frequently

reinvented, and yet, often no wheel can be found when it is really needed. The big bonus of efficiency offered by XQuery is lost.

In similar situations a key problem is the lack of any structure integrating the individual tools into a conceptual whole. Compare this to the realm of web services. Their functionality is structured into named services which are in turn structured into named operations. Each service provides a scope for conventions concerning names, structures and semantics. But now imagine there were only operations without containing services – instead of web services we would deal with “web operations”. We would get into trouble. We would be threatened by the same predicament which is created by an uncoordinated growth of command line tools.

A possible conclusion is that we should create command line tools which can be compared to web services. Service-like tools would act as containers for a set of related operations, providing a single point of access and the scope for a consistent vocabulary and patterns of tool behaviour. But this ambition is at cross purposes with a tool developer's desire to be agile and responsive. We want to add new functionality quickly and frequently. We want to be ready to augment the flexibility of existing functionality - adding further control parameters - as soon as the need is detected. XQuery is a scripting language, which encourages rapid and agile development. Committed to rapid development, we are reluctant to invest effort into structural integration.

The development of rich, service-like command line tools is very costly and challenging if starting from scratch – without purpose-designed development tools. This paper introduces a framework which changes the situation radically, enabling the instantiation of a service-like tool within seconds, the writing of a deliverable first version within an afternoon, and the continuous extension and refinement which always remains simple and straightforward, because extension can be accomplished without increasing the overall complexity. The second key benefit of framework based development is the high quality of the emerging tool, which is largely due to a rich input validation model, as well as the consistency of “look-and-feel” (invocation syntax and basic behaviour) across all tools built using the framework.

The framework is based on a generic concept of service-like tools, called *topic tools*. We start with the concept, proceed to the user-perspective (tool usage) and then inspect the developer-perspective – how to write topic tools.

Concept: topic tool

For the purposes of this discussion, the term "topic tool" denotes a particular class of programs. A **topic tool** is a program offering a single command line interface to multiple functionality, typically related to a single topic. The command line interface reflects a *unified* invocation model: the invocation of all topic tools has the same structure and is evaluated in accordance with the same syntactic conventions. There are also behavioural rules to which any topic tool adheres, for example concerning usage hints, user input validation, user input error reporting and the possibility to use stored invocations. Uniformity and user convenience are further encouraged by the standardization of rich parameter types for specifying input resources, name filters, path filters and other entities expressing complex details of the user's intent.

Goals

This paper describes a work with the following goals.

1. To define a generic model of a topic tool command line interface
2. To define generic features of topic tool behaviour
3. To provide a development framework for the implementation of topic tools, emphasizing
   - The speed of development (code generation, application code supported by infrastructure code)
   - The ease of development (reduction of complexity)

- Tool quality (robustness, predictability, validation and diagnostics)
- Tool extensibility (low effort to add operations and to extend existing operations)
- Framework extensibility (possibility to define new parameter types)

Topic tools: user perspective

The interface of topic tools reflects a unified invocation model: the invocation of all topic tools has the same structure, is evaluated in accordance with the same syntactic rules and can rely on the same set of behavioural features concerning error reporting, usage assistence and support for storing and reusing invocations.

Tool invocation model

The invocation of a topic tool specifies a named operation and (optionally) the values of named parameters. The syntax used is the same for any topic tool. Technically speaking, a topic tool has a single parameter named “request”, which is a structured string identifying the operation and supplying parameter values. Operation name and parameters are separated by a question mark, and parameters are separated by a comma. This is the general pattern:

```
operationName ? param1=value1, param2=value2
```

Literal commas within parameter values must be escaped using a backslash. Literal uses of the backslash character must also be escaped. If the parameter value has multiple items, these are supplied as a concatenated string. The item separator is either whitespace (one or more characters) or a semicolon, optionally surrounded by whitespace. Which item separator to use is prescribed by the topic tool for each parameter individually. If a parameter supports multiple items, literal uses of the item separator within parameter values must be escaped using a backslash. If no parameters are supplied, the request consists only of the operation name. Here come a few requests addressed to a topic tool called `xclear`, which offers various operations for analyzing XML document contents. The creation of this topic tool will be described in the tutorial proposed by a later section of this paper.

```
docs    ? doc=/logs/log01.xml /logs/log02.xml /logs/log03.xml
doctype ? docs=^/projects/xsd/xsd-fpml%*
names   ? docs=^/projects/xsd/niem-2.1%*
names   ? docs=^/projects/xsd/niem-2.1%*, scope=elem, names=*augment*
```

Which operations are available? Given an operation, which parameters are possible and which are mandatory, what is the minimum/maximum number of value items, and which character must be used as item separator? What are the parameter types, and what the default values, if any? The answers are stored in the *tool scheme*, which models any aspect of the tool interface. Every topic tool supports a `_help` operation providing usage assistence, which is based on the actual tool scheme. The `_help` operation of every topic tool supports the same set of parameters. The use of `_help` is facilitated by a syntactic shortcut: the operation name may be omitted (if parameters are specified) or replaced by a question mark (if no parameters are specified). Invocations #1 and #3, and invocations #2 and #4 are equivalent:

```
_help
_help?ops=paths, default=true, type=true
?
?ops=paths, default=true, type=true
```

Called without parameters, `_help` displays a simple summary of the topic tool, consisting of operation names and the parameter names and cardinalities:

```
REQUEST:
?

==>
TOOL: xclear

OPERATIONS   PARAMS
=================================================
_help        default, mode, ops?, type
dcat         dfd+
docs         dcat*, doc*, docs*
doctypes     attNames, dcat*, doc*, docs*, elemNames
names        dcat*, doc*, docs*, names?, scope
paths        dcat*, doc*, docs*, paths?
values       dcat*, doc*, docs*, nvalues?
=================================================
```

To see the types and default values, set the parameters `type` and `default` to "true". You can also restrict the report to particular operations by supplying a name filter as value of parameter `ops` (more about name filters later).

```
REQUEST:
?type=true, default=true, ops=_h* do*

==>
TOOL: xclear

OPERATIONS   PARAMS
=============================================================================
_help        default=false....... : xs:boolean
             mode=overview....... : xs:string
             ops................. : nameFilter?
             type=false.......... : xs:boolean

docs         dcat................ : docCAT* (sep=;)
             doc................. : docURI* (sep=WS)
             docs................ : docDFD* (sep=;)
               At least 1 of these parameters must be set: dcat, doc, docs

doctypes     attNames=false...... : xs:boolean
             dcat................ : docCAT* (sep=;)
             doc................. : docURI* (sep=WS)
             docs................ : docDFD* (sep=;)
             elemNames=false..... : xs:boolean
               At least 1 of these parameters must be set: dcat, doc, docs
=============================================================================
```

Operation and parameter names can be abbreviated by leaving out trailing characters, as long as the used prefix unambiguously identifies the operation or parameter, respectively. Operation and parameter names are case sensitive from the developer perspective, but case insensitive from the user perspective; entered names are normalized to the case defined by the tool scheme.

Another syntactic shortcut applies to parameters of Boolean type. They can be specified by only supplying the name (equivalent to param=true), or the name preceded by a tilde (equivalent to param=false). Example:

```
?ops=paths, ~default, type
```

An XQuery based topic tool receives the invocation string as value of the single external variable, named `request`. Call syntax of course depends on the actual XQuery processor. Using `BaseX`, for example, calling a topic tool looks similar to this:

```
basex -b "request=names ? docs=^/projects/xsd/niem-2.1/*" /projects/xclear/x
```

Tool scheme

A topic tool is defined by a **tool scheme** which models every aspect of the tool interface. The tool scheme supports both, the tool user and the tool developer in their work. The tool scheme is automatically constructed by collecting annotations contained by the tool sources, augmenting them and integrating them into a single XML document. The scheme specifies:

- operation names
- for each operation all parameter names
- for each parameter …
  - type
  - cardinality
  - item separator (if maximum number of occurrences greater 1)
  - (optional) default value
  - (optional) facets
  - (optional) normalizations

Every parameter has a *type* and a *cardinality*. The type can be a built-in schema type, a `ttools`-defined type (a generic type defined by the topic tools framework described in this paper, available in any framework-based topic tool) or a tool-specific type. All built-in schema types are supported, excepting `xs:ID` and `xs:NOTATION`.

An example of a framework-defined type is `docDFD`, which represents a filter extracting XML documents from directories. ("DFD" denotes "directory filter descriptor".) An example of a tool-specific type is `travelScenario`. This type is defined by a topic tool for evaluating touristic data; parameter values are structured strings describing a travel in terms of travel dates and traveller birthdates.

A *facet* constrains the parameter value beyond the constraints implied by the parameter type. A facet has a facet type and a string value. Each facet type defines a particular type of constraint applying to parameter item values. The following table lists the framework-defined facets, as of writing. Besides these generic facets, topic tools may also define tool-specific facets (see the section called “Advanced usage - extending the framework”).

Table 1.

Parameter facets defined by the `ttools` framework.

Facet	Meaning
length	string length
minLength	minimum string length
maxLength	maximum string length
minInclusive	minimum value, inclusive
maxInclusive	maximum value, inclusive

Facet	Meaning
minExclusive	minimum value, exclusive
maxExclusive	maximum value, exclusive
pattern	regular expression
values	enumeration
filter	a name filter
future	in the future (applicable to `xs:date`, `xs:dateTime` and `xs:time`)
past	in the past (applicable to `xs:date`, `xs:dateTime` and `xs:time`)
nonFuture	not in the future (applicable to `xs:date`, `xs:dateTime` and `xs:time`)
nonPast	not in the past (applicable to `xs:date`, `xs:dateTime` and `xs:time`)

The `_help` operation provides usage assistence, which is based on the tool scheme. An XML representation of the complete tool scheme can be obtained by calling `_help` with parameter `mode` set to `scheme`, for example:

```
basex -b "request=? mode=scheme" /projects/xclear/xclear.xq
```

Object parameter types

The topic tools framework defines several parameter types providing high value to the tool user, as well as to the tool developer. The parameter values are not treated as strings, but as shortcut representations of structured information. The structures obtained from the parameter values are recognized by powerful functions, and the behaviour of these functions conveys to the parameter value a special meaning. For example, when the string

```
get* fetch* ~*IFF#c*
```

is used as argument for a parameter of type `xs:string`, it is this string what arrives in the tool application. When the string is used as argument for a parameter of type `nameFilter`, however, the application does not receive a string, but a complex XML element, which contains elements and attributes representing positive and negative string matching conditions, regular expressions and regex options. The element is the XML representation of a filter, selecting names starting with “get” or “fetch” (case-insensitively), but excluding names ending with “IFF” (case-sensitively). The filtering behaviour is provided by functions that "understand" the element structure and to which the filter element is passed as a parameter. These functions and the XML element are of course designed jointly, the element for defining behaviour, and the functions for implementing behaviour. Element and functions should therefore be viewed as a conceptual unit encapsulating data and functionality, irrespective of whether the functions are formally bound to the element, as would only be possible when using an object-oriented language. Parameter types matching this pattern - they are parsed into specific elements which can be handled by specific functions - are therefore collectively called **object parameter types**. Conceptually, the user supplies objects, rather than strings. The user benefits from the high expressiveness of such parameters – the possibility to convey much information in a very concise way. The developer benefits from the rich functionality immediately available to him. An additional advantage for the user is the stylistic consistency which object parameter types enforce. Dealing with different operations and even with different topic tools, the style of specifying, say, name-based filtering is always the same.

Name filter (`nameFilter`)

The type `nameFilter` enables the definition of a filter on names which is very simple for simple requirements, and at the same time expressive enough to enable very fine-grained filtering. The filter text is a whitespace-separated list of one or more filter items. A filter item is either a positive filter (if

not prefixed by the character ~) or a negative filter (if prefixed by ~). The filter retains all candidates which match at least one positive item (if there are any positive items) and does not match any of the negative items. Each filter item consists of a filter term, which may be accompanied by options. Any options are appended to the term, using character # as separator. A filter term can use glob syntax (if not accompanied by option `r`) or regex syntax (if accompanied by option `r`). By default, filter terms are matched case-insensitively, unless accompanied by option `c`. The following table show a few examples of `nameFilter` values.

Table 2.

Examples of name filter values and their meaning.

Filter string	Matches (case-insensitively if not otherwise stated)
Book	the string "book"
Book*	a string starting with "book"
~Book*	any string not starting with "book"
Find*Response	a string starting with "find" and ending with "response"
Find*Response#c	a string starting with "Find" and ending with "Response, case-sensitively
Find(Scart|Booking)RQ#r	a string matching the regular expression, using regex option 'i'
Find(Scart|Booking)RQ#rc	a string matching the regular expression, *not* using regex option 'i'
*RQ#c *RS#c	a string ending with “RQ” or “RS”, case-sensitively
~*Help* ~Options*	any string not containing "help" and not starting with "options"
*RQ#c *RS#c ~*Help* ~Options*	a string ending with “RQ” or “RS”, case-sensitively, but excluding any strings containing "help" or starting with "options"

Name filter map (`nameFilterMap`)

A related type is `nameFilterMap`, which is a mapping of name filters to values. The values are themselves constrained by a type, which defaults to `xs:string`. For example, the type specification `nameFilterMap(xs:integer)` denotes a name filter map using integer values, and `nameFilterMap` denotes a name filter map using string values. The value text is a % concatenated list of one or more entries. An entry specifies a mapping in the form "value : nameFilter", or the default value. If the default value contains a literal colon, it must be escaped with a backslash. For example, the following map

```
10 % 0 :  *id *timestamp* % 100 : status
```

specifies a default value (10) as well as two values (0 and 100, respectively) bound to different name filters. Such a map could be used, for example, for specifying the number of data values to be reported, dependent on the element names. Then no values would be reported for elements with a name ending on “id” or containing “timestamp”, 100 values for "status" elements, and 10 values for any elements not matching any of the two name filters.

Path filter (`pathFilter`)

Another derivative of name filters is the `pathFilter` type. It consists of a sequence of name filters, separated by single or double slashes, and it can be used to filter name paths (e.g. XPath data paths or

qualified file names). Each name filter specifies a step, which is applied either to the children of the previous step (if preceded by a single slash) or to its descendants (if preceded by a double slash). If the first step is not preceded by a slash, the filter behaves as if the first step were preceded by a double slash: the first step may match any step in the path. Contrary, if the first step is preceded by a slash, it must match the first step of the path. For example, the following path filter:

```
2.0/*.xsd
```

matches any path with a last step matching `*.xsd` and a second last step matching `2.0`. Path filters may be used, for example, in order to select XML items according to their name and location within a document, or to select files according to their name and location in the file system.

Directory Filter Descriptors (docDFD, txtDFD)

A very important reuse of name and path filters is given by **Document Filter Descriptors** (the types `docDFD`, `textDFD` and `csvDFD`). They serve to select files (XML, text or csv files) from directories. A filter descriptor consists of two or three % separated fields. The first field is a whitespace separated list of one or more directories in which to search for files. If a directory name is preceded by character ^, the directory is searched recursively, that is, the search includes all direct or indirect sub directories. The second field specifies a name filter which is applied to file names – only files with matching name may be selected. An optional third field refines recursive search. It supplies a path filter, which constrains the (sub) directories from which files can be selected. For example, the following value

```
^/projects/niem-2.1 ^/projects/niem-2.0 /project/niem-submissions % *.xsd
```

selects all xsd files in or under one of the niem directories, or in the directory `niem-submissions`. Adding a path filter, like so:

```
^/projects/niem-2.1 ^/projects/niem-2.0 /project/niem-submissions % *.xsd % c
```

would restrict recursive findings to files found in a directory located under (but not in) a directory with a name starting with “edxl”. A parameter of type `docDFD` only selects XML files, ignoring any non-XML files matching the descriptor; the effective parameter value (the value delivered to the application) is a sequence of document nodes. A parameter of type `txtDFD` selects all files matching the descriptor, and the effective parameter value is a sequence of strings, each string providing the contents of one file. A parameter of type `csvDFD`, finally, also selects all files matching the desciptor, but treats them as csv files and delivers to the application XML documents, each one representing the rows and cells of one csv file.

Document catalog (docCAT)

A document catalog (dcat) is an XML document containing a collection of references to other documents. The parameter type `dcat` has the following semantics: the user-supplied value is the document URI of a dcat; the value delivered to the tool application is the collection of all documents referenced by the dcat. A dcat can be created automatically by calling the built-in operation `dcat` and supplying to it a directory filter descriptor.

Tool specific object parameter types

The preceding sections introduced several object parameter types supported by any topic tool conforming to the model described in this paper. Apart from these, any topic tool may define *tool-specific* parameter types, including object parameter types. The `_help` operation supports a mode `types` which explains all generic as well as any tool-specific parameter types. See the section called “Advanced usage - extending the framework” for details about the implementation of tool-specific parameter types.

As an example, `otdsv`, a topic tool for inspecting touristic data conforming to the OTDS data model ([2]), uses a tool-specific parameter type `travelScenario`. A parameter value defines a trip in terms of booking date, start date and duration, as well as the ages or birthdates of all travellers. The following parameter value:

```
2014-04-02 2014-12-06 14 % 32 31 5 2
```

specifies a travel booked on 2014-04-02, starting on 2014-12-06 and lasting for 14 days, made by four persons aged 32, 31, 5 and 2 years. The tool implementation does not see the string, but a structured representation which is understood by various functions related to travel scenarios.

Storing and reusing invocation

Complex functionality tends to involve a considerable number of invocation parameters, and object parameter types invite rather complex parameter values. A call with four or more parameters is a bit of a strain, and the more so if they involve object parameters. On the other hand, routine use of topic tools can be fairly repetitive, using often very similar calls which are only distinguished by a parameter or two, whereas several other parameters are set to the same values every time. In such situations in can be helpful to store invocations and reuse them either verbatim or partially overwriting or extending them on a call by call basis. Every topic tool supports the `storeq` feature: if called with two question marks (instead of a single question mark), the tool returns a simple XML representation of the invocation. After storing the invocation in a file, it can be referenced in future tool invocations, omitting the operation name and using the file name instead, preceded by the character @. An example follows.

Saving an invocation:

```
basex –b 'request=priceItems??otds=/projects/otds/otds.xml, cond=occupancy p
```

Reusing it unchanged:

```
basex –b 'request=@piStd.tti' otdsv.xq
```

Reusing and partially overwriting and extending it:

```
basex –b 'request=@piStd.tti?scope=absolute, summary' otdsv.xq
```

Summary

To summarize, the user of topic tools benefits from several features:

- single point of access to a whole range of functionality (a "topic")
- uniform invocation pattern for all operations and all topic tools
- high expressiveness enabled by object type parameters
- syntactical sugar (name completion, shortcut for Boolean values)
- comprehensive help functionality
- user-friendly input error diagnostics
- invocation reuse

The next section explains how topic tools are developed. It introduces a very lightweight framework which enables the developer to create topic tools quickly, to extend them with minimum effort, and to achieve high quality.

Topic tools: developer perspective

A topic tool is expected to support a well-structured command-line interface, perform fine-grained input validation, report user input errors in a comprehensive way and offer rich help functionality. Furthermore, remembering the high complexity which a topic tool may easily attain (many operations

and all in all very many parameters), one might expect that topic tools are hard to create, to maintain and to extend.

Using the `ttools` framework, the contrary is the case: development is easy, fast and, hopefully, enjoyable. Adding a topic tool operation is by far easier and faster than implementing its functionality by a new, non-topic tool written from scratch. An added topic tool operation conforms to the behavioural expectations (smart syntax, validation, error reporting and help) from the first minute of its prototypic existence. Achievement of this behaviour might easily take weeks to attain (and months to un-bug) if implemented as part of the tool development, rather than being provided by the framework via code generation and libraries. A new topic tool is instantiated in a minute, and in typical settings a first useful and deliverable version is ready after a few hours of work. After the initial shot, the tool tends to grow quickly and organically, no matter if guided by a complex set of initial requirements, or in response to frequently emerging requirements, or both. In all cases, growth is facilitated by the fact that the current amount of functionality does not create a burden making the addition of new functionality more difficult. To add a second or a twenty second operation tends to require the same amount of work, or usually the addition of operations even becomes increasingly simple thanks to utilities which have been created along the way to the current state.

This section explains how to write topic tools using the `ttools` framework. It starts with a high level overview of the support offered by `ttools`. It summarizes the workflow of tool development, describes how XQuery programming is facilitated and how code writing is complemented by the writing of annotations. This overview will be followed by a brief tutorial intended to give you a hands-on impression of the process of developing a topic tool. The tutorial takes you through the steps of creating a new topic tool and filling it with a first set of functionality. The final part of this section on topic tool development will discuss advanced usage of `ttools`: it explains how to extend the infrastructure provided by `ttools`, defining new parameter types and new parameter facets.

Overview

The development support offered by `ttools` has four main aspects:

- Code generation
- Pseudo objects
- Utility functions
- Built-in operations

The main module of a topic tool is generated. Code generation is based on *annotations* scattered over XQuery library modules. Annotations are not formal annotations as defined by XQuery 3.0, but XML snippets placed in XQuery comments conforming to a syntax convention. Annotations typically define tool operations in terms of an operation name and the names, types and other properties of operation parameters. Annotations may also define new parameter types and new kinds of parameter facets.

Code generation is effected by an application *build*. The build is triggered by a command line call of `ttools`, which is a topic tool for building topic tools. The build call provides a tool name and a tool directory. During an application build, `ttools` investigates all library modules found in the application directory, and all annotations which they contain are used. The independent annotations scattered over the modules are integrated into a single model of tool functionality, called the *tool scheme*. A tool scheme defines tool functionality in terms of named operations and operation-specific sets of parameters.

The tool scheme is translated into an XQuery main module implementing a command line interface which gives access to all operations. On invocation, the main module parses, evaluates and validates user input, creating a comprehensive error report, if errors occur. Otherwise the main module integrates user input into an XML *request message* and invokes the *operation provider function*, the function which implements the operation. The provider function is hand-written by the tool developer. It has a standardized signature, receiving a single parameter which is the request message assembled by the main module, and producing an annotation-defined result type. The request message is an XML

element intended to be used exclusively via a set of access functions designed for the purpose. These access functions can be regarded as the *message interface*, functionality bound to its data content. The request message thus resembles a program object, which hides user *input data* behind an interface of accessors to `input information`.

For example, if a parameter is typed as a document URI, the user data is a URI, but the accessor returns a document node. While the user supplies a multiple-valued parameter as a concatenated string, the accessor delivers the parameter as a sequence of typed items. Annotation-defined default values are delivered in a way indistinguishable from explicitly supplied parameters. The difference between input data and input information is especially relevant in the case of object parameters (e.g. name filters), where input data are strings from which "objects" are constructed: XML elements designed to provide functionality when passed to the appropriate functions (e.g. `tt:matchesNameFilter`). It is important to realize how the use of the request object and its accessor interface in general, and the use of object parameters in particular, may simplify the work of application writing significantly.

`ttools` based development profits from *utility functions* in a dual way: (a) using "object functions", as just described, which are functions designed for use with XML elements constructed by `ttools` infrastructure; (b) general purpose functions, providing useful functionality not related to `ttools` specific structures. Examples of general purpose functions are a function for parsing CSV formatted text and a function for creating tabular text.

Every `ttools` based topic tool supports a range of *built-in operations*, implemented by modules which are automatically added to the tool directory during tool instantiation. A prominent role plays the generic *help operation*, which translates the tool scheme into usage information. Other built-in operations provide generic functionality which may add to the usefulness of the topic tool. As an example, built-in operation `doctypes` reports the document types (root element names and namespaces) encountered in a set of XML resources, which may be specified by any combination of document URIs, directory filter descriptors (see the section called "Directory Filter Descriptors (`docDFD`, `txtDFD`)") and document catalogs. Using further parameters, additional information about the documents can be requested.

Workflow

The `ttools` based development of a topic tool can be schematically described as the following workflow.

1. **Tool instantiation** – command line call creating and initializing the new topic tool.

2. **Prototyping** – command line call creating the initial version of a new XQuery library module, intended to implement a list of operations specified by a call parameter; the generated code contains initial versions of annotations and function definitions; the generated module is immediately callable after re-building the topic tool (see below).

3. **XQuery code writing** – developing the functions which implement the operations, as well as auxiliary functions, as appropriate.

4. **Annotation editing** – adding the definitions of new operations, or changing / extending the definitions of existing operations.

5. **Tool building** – command line call re-building the tool, translating the current annotations into the command line interface.

6. **Tool testing.**

Writing XQuery code

Coding is significantly simplified by the packaging of all input information into a single container (the request message) and its exposal via accessor functions (e.g. `tt:getParam`). Especially helpful are three facts:

- *Guaranteed availability*. Any mandatory operation parameter *is* supplied. If it were not, the function would not be called in the first place. Whether or not the annotation declares a default value, concerns user convenience, but not the developer: a mandatory value is present.
- *Guaranteed validity*. For any operation parameter a *valid* value is supplied. Validity refers to the type specification including item type, cardinality constraints and optional facets. The developer can simplify his work by protecting himself from inappropriate values: he can add facets to the parameter annotation, e.g. specifying enumerated values, a regular expression, minimum or maximum values. The infrastructure will not call the function unless user input satisfies all constraints.
- *Object parameters*. The definition of particular parameter types prescribes the parsing of user-provided text value, resulting in the construction of "objects", which are XML elements designed to be processed by functions. An example is the parsing of a name filter string into a name filter element which can be passed to a `tt:matchesNameFilter` function. The use of object parameters can reduce the complexity of application development, as the request accessors deliver the objects, not the source text.

An interesting aspect concerns the information flow between XQuery functions. When the operation providing function delegates some tasks to other functions, these may have a function signature including a request message parameter. The request message passed on may be constructed from scratch, as the message interface includes functions enabling such a construction. But it may also be the original request, or a modified copy of it - the request interface includes functions for creating copies with parameters removed, overwritten or added. When the request passed on is not constructed from scratch, but received from the outside, whether or not subsequently modified before being passed on, it may carry information which the called function evaluates, but which the calling function need not be aware of. As an example, consider a function which operates on the results of some kind of search. Such a function may pass the original request on to the search function, trusting the request (which it received from the user and which has passed validation) to contain all information relevant to the search function. Now the search function may *evolve*, incrementally extend its parameter usage and evaluate additional input parameters, without the calling function being aware of it – search control parameters do not concern it and are not inspected. As a result, the search function may evolve freely and independently of other functions operating on the search results, as the request message *hides* the changes of search related input. If today the search function begins to evaluate some additional input, the code of search result processing functions need not be changed. The only adaptation necessary concerns the annotations: in order to benefit from the extended search functionality, a search result processing operation should add the new, search related parameters to its own operation signature. But even this adaptation can as well be postponed, until the need is felt.

Such an interaction of functions may be called message-based – as opposed to parameter list based. `ttools` provides a sophisticated set of message related functionality, including functions for transforming messages by extension, value overwriting and reduction. The primary use of this functionality is the dealing with the original request message, constructed by infrastructure from user input. However, beyond that, the availability of message related functionality encourages the practise of message-based interactions among functions. A recommended design pattern of topic tools can be summarized as follows: (a) identify pieces of core functionality, which have a *potentially growing set of input information*; (b) provide such functionality with a message-based signature, rather than a parameter list based signature. This approach can considerably help in the process of evolving complex applications in a safe way.

Writing annotations

Writing code and editing the annotations are two activities performed in tandem. While writing the code of an operation, the developer switches over to the annotations whenever he decides to modify operation input, typically by adding a further control parameter. Then he resumes code writing, referring to the added parameters using request message accessors.

The annotation of an operation defines the operation in terms of its parameters. For every parameter, the following information is specified (the last two items optionally):

- parameter name

- parameter type, including cardinality (e.g. ?, *, {3}, {1,3})
- facets (e.g. regular expression, value list, minimum/maximum values)
- membership in parameter groups (for joint cardinality constraints, e.g. when at least one of several parameters must be supplied)

The editing of annotations takes effect immediately after re-building the topic tool. This is achieved by a command line call (invoking the `build` operation of `ttools`) and usually lasts less than a second.

Advanced usage - extending the framework

Developing topic tools using the `ttools` framework gives access to the automated construction of object parameters and automated facet validation. `ttools` ships with a range of built-in object parameter types and facet kinds.

A topic tool may extend the framework by defining *tool specific* parameter types and facet kinds. This is very easily done. In order to define a tool specific object parameter type, one has to write a parsing function which constructs from an input string an element of the appropriate structure. The signature must take a single input parameter of type `xs:string`, and the return value must be a single item of arbitrary type (`item()`). Similarly, in order to define an additional facet kind, one must write a facet validation function which will validate a parameter value item against the new facet kind. The signature must take two input parameters, the first of type `item()` receiving the value item to be checked, the second of type `xs:string` receiving the facet value; the return value must be `element(z:errors)?`. A library module containing type definitions and/or facet validation functions must contain an annotation of type `extensions`, which binds the parsing functions to parameter types and the facet validation functions to facet kinds. The implementation of framework extensions may for example look as follows:

```
(:~
 : ttoolsExtensions.mod.xq - application specific extensions of the parameter pa
 :)

(:~@extensions
<extensions>
    <type name="travelScenario" parser='parseTravelScenario'/>
    <facet name="blacklisted" validator='checkBlacklisted'/>
</extensions>
:)
module namespace m="http://www.ttools.org/otdsv/xquery-functions";
...
declare namespace z="http://www.ttools.org/structure";
...
declare function m:parseTravelScenario($text as xs:string)
        as item() {...}

declare function m:checkBlacklisted($item as item(), $facetValue as xs:string)
        as element(z:errors)? {...}
```

After dropping this library module into the tool directory, the annotations of operations may define parameters of type `travelScenario` and constrain parameters to conform to a `blacklisted` facet. The definition of additional object parameter types is typically accompanied by functions taking elements of the respective structure as input. These functions constitute the "interface" of the new object type and largely determine the value which the object type has for the tool application.

It should be remembered that object construction and facet validation are performed automatically, so that the application never needs to call the functions which it provides for the definition of additional types and the evaluation of additional facet kinds.

Summary

The development of a topic tool is simplified by code generation. What remains to be done is the writing of functions which are exposed as operations. Code writing is accompanied by annotation editing, which provides the code with an environment of validated information. Frequent extension of this information – for example caused by stepwise refinement of functionality – is accomplished in a painless way. Whenever the need arises, just add a parameter to the operation annotation: from now on the parameter value is available to the operation code.

Code writing is greatly simplified by having all external input encapsulated in a single container – the request message – which is accessed via functions, completely isolating application code from the original forms of external input. Application code is thus embedded in an infrastructure which takes on the responsibility of input validation and input augmentation. In the case of so-called object parameter types, the embedding infrastructure transforms user input into entities corresponding to program objects.

The message-related functionality provided by `ttools` also encourages the use of messages in function signatures in general, whether or not the function is exposed as a topic tool operation. Messages enable the "tunneling" of information, so that a calling function may be unawares of the information requirements of the called function. The use of message-controlled functions thus favours a robust integration of complex functionalities - like search and processings of search result - which is an important concern in general, and when developing topic tools in particular.

Getting started with topic tools

This part of the paper proposes a short tutorial which may give you some hands-on impressions of the development of topic tools. We set out to create the first version of a command line tool which assists the user in the analysis of XML resources. We have already downloaded `ttools` (download URI will be added here), which is itself a topic tool supporting the development of topic tools. We have further installed BaseX ([4]), an open source XQuery processor supporting XQuery 3.0.

We decide to call the new tool `xclear`, and we plan to supply the first version with three operations:

- `names` - reports names of elements and attributes
- `paths` - reports data paths of elements and attributes
- `values` - reports data content of elements and attributes

In what follows, we assume that the `ttools` installation directory is `/projects/ttools`, and that `xclear` shall be installed in directory `/projects/xclear`.

Tool instantiation

The very first step of tool development is tool instantiation. We call the `ttools` operation `new`, specifying a tool directory whose name is equal to the intended tool name:

```
basex -b "request=new?dir=/projects/xclear" /projects/ttools/ttools.xq

==>
============================================================

Topic tool created:  xclear
Tool directory:      /projects/xclear

The tool can already be called. Example:
   basex -b "request=?" /projects/xclear/xclear.xq
```

```
Use operation 'add' for adding module prototypes. Example:
   basex -b "request=add?dir=/projects/xclear, mod=fooMod, ops=fooOp barOp fooba

================================================================
```

As a result, the tool directory `xclear` has been created and filled with various XQuery library modules, as well as a generated main module `xclear.xq`, which can be called.

```
01.07.2014  22:12             4.473 xclear.xq
07.04.2014  23:47               599 _constants.mod.xq
27.04.2014  12:50             8.656 _csvParser.mod.xq
28.06.2014  13:11            11.855 _dcat.mod.xq
28.06.2014  12:39             5.172 _docs.mod.xq
01.07.2014  22:12             1.155 _extensions.mod.xq
29.06.2014  08:52             8.333 _help.mod.xq
15.04.2014  00:09            19.666 _nameFilter.mod.xq
15.04.2014  00:13            14.944 _nameFilter_parser.mod.xq
07.04.2014  22:29            20.893 _namespaceTools.mod.xq
07.04.2014  22:29             1.744 _reportAssistent.mod.xq
29.06.2014  09:38            55.935 _request.mod.xq
28.06.2014  13:18            10.144 _request_getters.mod.xq
12.04.2014  08:29             2.908 _request_parser.mod.xq
28.06.2014  12:02             3.260 _request_setters.mod.xq
27.04.2014  14:37             3.606 _stringTools.mod.xq
```

Exploring the initial tool version

A topic tool has a single parameter, `request`. If supplied with the parameter value "?", a topic tool is expected to give an overview of its operations. We give it a try:

```
basex -b "request=?" /projects/xclear/xclear.xq

==>
TOOL: xclear

OPERATIONS   PARAMS
=================================================
_help        default, mode, ops?, type
dcat         dfd+
docs         dcat*, doc*, docs*
doctypes     attNames, dcat*, doc*, docs*, elemNames
=================================================
```

Our freshly created tool has already four operations! Operation `_help` is what is called when the request value starts with a question mark, rather than an operation name. The other three operations are built-in operations augmenting the functionality of any topic tool we build using `ttools`. For example, operation `doctypes` provides basic information about a set of documents. In order to take a closer look at its parameters, we refine our `_help` request, specifying the Boolean parameters `type` and `default`, in order to learn the parameter types and see any default values. We narrow down the response to operation `doctypes` by setting the optional parameter `ops` to the value `doct*`:

```
basex -b "request=?default, type, ops=doct*" /projects/xclear/xclear.xq

==>
TOOL: xclear
```

```
OPERATIONS   PARAMS
=============================================================================
doctypes     attNames=false..... : xs:boolean
             dcat............... : docCAT* (sep=WS)
             doc................ : docURI* (sep=WS)
             docs............... : docDFD* (sep=;)
             elemNames=false..... : xs:boolean
               At least 1 of these parameters must be set: dcat, doc, docs
=============================================================================
```

The parameter types `docURI`, `docDFD` and `docCAT` are very important, because they are *the* parameter types used for supplying the tool with XML documents. If an operation demands input which may consist of a set of documents, rather than necessarily a single document, it is a recommended pattern to use a triple of parameters of type `docURI`, `docDFD` and `docCAT`, respectively, and (if appropriate) mark them as a group of which at least one parameter must be set. This gives the user great freedom how to supply input documents, as he can use any combination of three approaches:

- `docURI` typed parameter: expects a document URI
- `docDFD` typed parameter: expects a directory filter descriptor
- `docCAT` typed parameter: expects the document URI of a document catalog ("dcat")

A dcat can be created calling operation `dcat` and supplying it with a document filter descriptor. If stored in a file, the dcat can later be supplied to any parameter of type `docCAT`. To get a feeling for our new tool, we invoke operation `doctypes`, supplying it with a directory filter descriptor. Suppose we are interested in the XML resources directly or indirectly contained by a directory `/projects/xsd/niem-2.1`. This is how we request a `doctypes` report:

```
basex -b "request=doctypes?docs=^/projects/xsd/niem-2.1/*" /projects/xclear/xcl

==>
  <z:doctype name="schema@http://www.w3.org/2001/XMLSchema" count="123">
    <doc href="file:///C:/projects/xsd/niem-2.1/ansi-nist/2.0/ansi-nist.xsd"/>
    <doc href="file:///C:/projects/xsd/niem-2.1/ansi_d20/2.0/ansi_d20.xsd"/>
    <doc href="file:///C:/projects/xsd/niem-2.1/apco/2.1/apco.xsd"/>
    ...
  </z:doctype>
```

We just used a directory filter descriptor as input to an operation. One may also first use it to create a dcat, and later use this dcat in order to supply an operation with all documents which the dcat references. To create a dcat, we call the built-in `dcat` operation, supplying the directory filter descriptor as value of parameter `dfd`:

```
basex -b "request=dcat?dfd=^/projects/xsd/niem-2.1/*" /projects/xclear/xclear.x

==>
<dcat dirs="^/projects/xsd/niem-2.1" files="" subDirs="" countFiles="123" t="20
  <doc href="file:/projects/xsd/niem-2.1/ansi-nist/2.0/ansi-nist.xsd"/>
  <doc href="file:/projects/xsd/niem-2.1/ansi_d20/2.0/ansi_d20.xsd"/>
  <doc href="file:/projects/xsd/niem-2.1/apco/2.1/apco.xsd"/>
   ...
</dcat>
```

If we store the output in a file (say, `/projects/dcats/dcat-niem-2.1.xml`), we can then supply the file name to any parameter of type `docCAT`, effectively passing to the tool all documents referenced by the catalog. For example, the following call of `doctypes` yields the same result as the first one:

```
basex -b "request=doctypes?dcat=^/projects/dcats/dcat-niem-2.1.xml" /projects/x
```

By now we have some feeling how to *use* a topic tool, namely how to specify aggregate input in alternative ways. We are ready to fill our tool with new functionality.

Adding prototypes

Our tool shall be helpful when we want to explore an XML document, or a set of documents. The first version shall have three operations: `names`, `paths` and `values`. We decide to place their implementation in a single module, called `items`. A convenient way to create the module is to use the `add` operation of `ttools`. We specify the topic tool (parameter `dir`), the module name (parameter `mod`) and the operations (parameter `ops`):

```
basex -b "request=add?dir=/projects/xclear, mod=items, ops=names paths values" /

==>
============================================================
XQuery module created: items.mod.xq
Operations:            names paths values

Directory:             /projects/xclear
Topic tool:            xclear

The new operations are available. Example:

   basex -b "request=names?doc=doc1.xml doc2.xml doc3.xml" /projects/xclear/xcl

To implement them, edit these functions: names paths values
============================================================
```

We check that new operations have appeared in the tool interface:

```
basex -b "request=?" /projects/xclear/xclear.xq

==>
TOOL: xclear

OPERATIONS   PARAMS
=================================================
_help        default, mode, ops?, type
dcat         dfd+
docs         dcat*, doc*, docs*
doctypes     attNames, dcat*, doc*, docs*, elemNames
names        dcat*, doc*, docs*
paths        dcat*, doc*, docs*
values       dcat*, doc*, docs*
=================================================
```

Indeed, the tool now has three new operations - `names`, `paths` and `values`. Their parameter sets are just a first proposal which we will extend or change. Now we inspect the generated module `items.mod.xq`. A complete listing can be found in the appendix Appendix A, *The `items` module as generated by `ttools`*. Here we look at selected parts. The text begins with this:

```
(:
 : ------------------------------------------------------------------------
 :
 : items.mod.xq - Document me!
 :
```

```
 : ------------------------------------------------------------------------
 :)

(:~@interface
<interface>
   <operations>
      <operation name="names" type="node()" func="names">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="paths" type="node()" func="paths">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="values" type="node()" func="values">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
    </operations>
</interface>
:)

module namespace f="http://www.ttools.org/xclear/xquery-functions";
...
```

The `build` operation of `ttools` searches all library modules in the tool directory for text enclosed between `(:~@interface` and the closing `:)`. This text is interpreted as XML annotation declaring operations which the containing library module contributes to the topic tool. Each operation is defined in terms of a name, a function implementing it (@func, defaulting to the operation name), a return type (@type, defaulting to `node()`), parameters (<param>) and optional parameter groups (<pgroup>). Parameter groups can be used for expressing cardinality constraints referring to a group of parameters, rather than to a single parameter - for example "at least one (at most one / exactly one) of these parameters (...) must be specified".

Each parameter is defined in terms of a name (@name), a type (@type), (optionally) facets (@fct_xyz) and (optionally) membership in a parameter group (@pgroup). Facets constrain valid parameter values beyond the constraints implied by the type. The type consists of an item type, optionally followed by a cardinality constraint. The parameters as they were generated are of course only meant as a starting point, to be extended and/or changed. We note that the generated module conforms to the recommended practise to support three alternative ways of specifying input documents. It uses three parameters of types `docURI`, `docDFD` and `docCAT` and joins them into a parameter group, constraining the user to provide at least one of these parameters.

Now we inspect the generated functions implementing the operations. For example:

```
(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:names($request as element())
```

```
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    return
        <z:names countDocs="{count($docs)}">{
           ()
        }</z:names>
};
```

So the initial version just creates an element echoing the operation name and fills it with a @countDocs attribute displaying the number of input documents. Let's try, using a directory `/projects/xsd/niem-2.1` as input:

```
basex -b "request=names?docs=^/projects/xsd/niem-2.1/*" /projects/xclear/xclear

==>
<z:names xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123"/>
```

First operation, first implementation

Now we know that we are "connected" – the tool can be supplied with input documents - and we are ready to add code. Here comes a first version of the `names` operation:

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $atts :=
        for $n in distinct-values($docs//@*/local-name(.))
        order by lower-case($n) return <z:a n="{$n}"/>
    let $elems :=
        for $n in distinct-values($docs//*/local-name(.))
        order by lower-case($n) return <z:e n="{$n}"/>
    return
        <z:names countDocs="{count($docs)}">{
           <z:atts count="{count($atts)}">{$atts}</z:atts>,
           <z:elems count="{count($elems)}">{$elems}</z:elems>
        }</z:names>
};
```

We repeat our call of the `names` operation:

```
basex -b "request=names?docs=^/projects/xsd/niem-2.1/*" /projects/xclear/xclear

==>
<z:names xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123">
  <z:atts count="29">
    <z:a n="abstract"/>
    <z:a n="attributeFormDefault"/>
    <z:a n="base"/>
    ...
  </z:atts>
  <z:elems count="40">
    <z:e n="annotation"/>
    <z:e n="any"/>
    <z:e n="anyAttribute"/>
    ...
  </z:elems>
</z:names>
```

First refinement - adding a parameter

Now let us think about refinement. We want to give the user the possibility to determine the scope – only elements, only attributes, or both. To achieve this, we add to the annotation of operation `names` a new parameter:

```
<param name="scope" type="xs:string" fct_values="att, elem, all" default="all"/
```

Note the default value and the `values` facet, constraining valid parameter values to the strings “att”, “elem” and “all”. The function is easy to adapt: (a) add a line for reading the parameter value and (b) wrap the output of attribute and element names in `if` expressions dependent on the parameter value. Here comes the new version:

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $scope as xs:string := tt:getParams($request, 'scope')      (: READ PARA
    let $atts :=
        for $n in distinct-values($docs//@*/local-name(.))
        order by lower-case($n) return <z:a n="{$n}"/>
    let $elems :=
        for $n in distinct-values($docs//*/local-name(.))
        order by lower-case($n) return <z:e n="{$n}"/>
    return
        <z:names countDocs="{count($docs)}">{
            if ($scope eq 'elem') then () else                      (: DEPENDENT
                <z:atts count="{count($atts)}">{$atts}</z:atts>,
            if ($scope eq 'att') then () else                       (: DEPENDENT
                <z:elems count="{count($elems)}">{$elems}</z:elems>
        }</z:names>
};
```

Note the type of the parameter value in this line:

```
let $scope as xs:string := tt:getParams($request, 'scope')
```

We can confidently use the type without “?”, as the annotation implies a cardinality of exactly one item, and the framework will never call the function unless the user input is valid. We re-build and call `_help` in order to verify that our annotations have been compiled into code:

```
basex -b "request=build?dir=/projects/xclear" /projects/ttools/ttools.xq
basex -b "request=?type, default, ops=names" /projects/xclear/xclear.xq

==>
TOOL: xclear

OPERATIONS   PARAMS
================================================================================
names        dcat.............. : docCAT* (sep=WS)
             doc............... : docURI* (sep=WS)
             docs.............. : docDFD* (sep=;)
             scope=all......... : xs:string; facets: values=att, elem, all
               At least 1 of these parameters must be set: dcat, doc, docs
================================================================================
```

We check that validation proceeds as it should, supplying an invalid `scope` value:

```
basex -b "request=names?docs=^/projects/xsd/niem-2.1/*,scope=attributes" /proje

==>
Invalid call
============

INVALID_PARAMETER_FACET       Parameter 'scope': item value (attributes) not am
                                 values (att, elem, all).

-------------------------------------
```

Calling the tool correctly, we get the correct result:

```
basex -b "request=names?docs=^/projects/xsd/niem-2.1,scope=att" /projects/xclea

==>
<z:names xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123">
  <z:atts count="29">
    <z:a n="abstract"/>
    <z:a n="attributeFormDefault"/>
    <z:a n="base"/>
    ...
    <z:a n="use"/>
    <z:a n="value"/>
    <z:a n="version"/>
  </z:atts>
</z:names>
```

Adding a name filter

We add one more refinement: the user may optionally specify a name filter, so that only items with a matching name are returned. It does not take more than a minute to achieve this. First we add an optional parameter `names` to the annotation, using the object parameter type `nameFilter`:

```
   <param name="names" type="nameFilter?"/>
```

The adaptation of the function is minimalistic. We read the parameter value:

```
   let $filter as element(nameFilter)? := tt:getParams($request, 'names')
```

and add a couple of predicates to the path expressions returning attributes and elements, respectively:

```
    let $atts :=
        for $n in distinct-values($docs//@*
            [not($filter) or tt:matchesNameFilter(local-name(.), $filter)]/loca
        order by lower-case($n) return <z:a n="{$n}"/>
    let $elems :=
        for $n in distinct-values($docs//*
            [not($filter) or tt:matchesNameFilter(local-name(.), $filter)]/loca
        order by lower-case($n) return <z:a n="{$n}"/>
```

Done! After a re-build, we can retrieve name-filtered item names:

```
basex -b "request=build?dir=/projects/xclear" /projects/ttools/ttools.xq
basex -b "request=names?docs=^/projects/xsd/niem-2.1/*,scope=att, names=*min*" /
```

```
==>
<z:names xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123">
  <z:atts count="1">
    <z:a n="minOccurs"/>
  </z:atts>
</z:names>
```

Second operation, first implementation

Now we tackle the second operation, returning data paths. First we write a little function which returns for any XML attribute or element its data path:

```
declare function f:path($n as node())
        as xs:string {
    string-join(
        $n/ancestor-or-self::node()/concat(self::attribute()/'@', local-name(.)
        '/')
};
```

With this function in place, the operation is implemented very quickly:

```
declare function f:paths($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $paths :=
        for $p in distinct-values($docs//(*, @*)/f:path(.))
        order by lower-case($p)
        return <z:path p="{$p}"/>
    return
        <z:paths countDocs="{count($docs)}" countPaths="{count($paths)}">{
           $paths
        }</z:paths>
};
```

After a re-build, we can retrieve a path list:

```
basex -b "request=build?dir=/projects/xclear" /projects/ttools/ttools.xq
basex -b "request=paths?docs=^/projects/xsd/niem-2.1/*" /projects/xclear/xclear

==>
<z:paths xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123" countP
  <z:path p="/schema"/>
  <z:path p="/schema/@attributeFormDefault"/>
  <z:path p="/schema/@elementFormDefault"/>
   ...
</z:paths>
```

Adding a path filter

Now we refine the operation by adding a path filter parameter. This enables the user to search for paths with a particular pattern, e.g. ending with a particular element or attribute, or all children or descendants of elements with a particular name. This is easily done. We extend the operation annotation by an optional filter parameter, using the object parameter type `pathFilter`:

```
   <param name="paths" type="pathFilter?"/>
```

And we change the function code as follows. We read the parameter value:

```
   let $filter as element(pathFilter)? := tt:getParams($request, 'paths')
```

and we introduce a predicate which filters the paths, rejecting any candidates not matching the path filter, if a path filter has been supplied:

```
   distinct-values($docs//(*, @*)/f:path(.))[not($filter) or tt:matchesPathFilt
```

After a re-build, we can, for example, learn all data paths pointing to elements and attributes under `xs:appInfo` elements:

```
basex -b "request=build?dir=/projects/xclear" /projects/ttools/ttools.xq
basex -b "request=paths?docs=^/projects/xsd/niem-2.1/*, paths=appinfo//*" /proj

==>
<z:paths xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123" countP
  <z:path p="/schema/annotation/appinfo/@source"/>
  <z:path p="/schema/annotation/appinfo/ConformantIndicator"/>
  <z:path p="/schema/annotation/appinfo/resource"/>
  ...
</z:paths>
```

Third operation, first implementation

This operation reports data values found in attributes and elements. For each attribute and element name, a sample of values is displayed. In the first version, the sample size is set to the value 10. The complete code looks like this:

```
declare function f:values($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')

    let $limit := 10
    let $atts := $docs//@*
    let $elems := $docs//*[not(*)][text()]
    let $attNames := distinct-values($atts/local-name(.))
    let $elemNames := distinct-values($elems/local-name(.))
    let $attValues :=
        for $name in $attNames
        let $values := distinct-values($atts[local-name(.) eq $name])
        let $show := for $v in $values[position() le $limit]
                     order by lower-case($v) return <z:v v="{$v}"/>
        order by lower-case($name) return
            <z:a n="{$name}" count="{concat(count($show), '/', count($values))}
    let $elemValues :=
        for $name in $elemNames
        let $values := distinct-values($elems[local-name(.) eq $name])
        let $show := for $v in $values[position() le $limit]
                     order by lower-case($v) return <z:v v="{$v}"/>
        order by lower-case($name) return
            <z:e n="{$name}" count="{concat(count($show), '/', count($values))}
    return
        <z:values countDocs="{count($docs)}">{
            <z:atts countAtts="{count($atts)}">{$attValues}</z:atts>,
```

```
            <z:elems countAtts="{count($elems)}">{$elemValues}</z:elems>
        }</z:values>
};
```

After a re-build we can request reports which yield for each attribute and element name a sample of values found in items with that name:

```
basex -b "request=build?name=xclear, dir=/projects/xclear" /projects/ttools/tto
basex -b "request=values?docs=^/projects/xsd/niem-2.1" /projects/xclear/xclear.
```

```
==>
<z:values xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123">
  <z:atts countAtts="102132">
    <z:a n="abstract" count="1/1">
      <z:v v="true"/>
    </z:a>
    <z:a n="attributeFormDefault" count="2/2">
      <z:v v="qualified"/>
      <z:v v="unqualified"/>
    </z:a>
    <z:a n="base" count="10/816">
      <z:v v="ansi-nist:ALSCodeSimpleType"/>
      <z:v v="ansi-nist:BTYCodeSimpleType"/>
      <z:v v="ansi-nist:COLCodeSimpleType"/>
      <z:v v="ansi-nist:CSICodeSimpleType"/>
      <z:v v="ansi-nist:CSNCodeSimpleType"/>
      <z:v v="ansi-nist:CSPCodeSimpleType"/>
      <z:v v="ansi-nist:NISTImageType"/>
      <z:v v="s:AugmentationType"/>
      <z:v v="s:ComplexObjectType"/>
      <z:v v="xsd:token"/>
    </z:a>
    ...
</z:values>
```

Adding a name filter map

Now we want to make the size of value samples dependent on the item name, so that the user may instruct the tool, for example, to return no values at all for some names, and different numbers for different name patterns. To achieve this, we extend the operation annotation, adding a parameter of type `nameFilterMap(xs:integer)`:

```
   <param name="nvalues" type="nameFilterMap(xs:integer)?"/>
```

Then we modify the function code as follows. We read the parameter value:

```
   let $fmap as element(nameFilterMap)? := tt:getParams($request, 'nvalues')
```

and set the number of terms dependently on the item name:

```
   let $limit := tt:nameFilterMapValue($name, $fmap, $defaultLimit)
```

That was it. The following call, for example:

```
basex -b "request=values?docs=^/projects/xsd/niem-2.1/*, nvalues=0 % 5:docum* b
```

sets the value sample sizes to the following values:

- 0 : by default
- 5 : if the name starts with “docum” or equals ”block”
- 10: if the name starts with “target”
- 20: if the name equals “use”

The result looks accordingly:

```
<z:values xmlns:z="http://www.ttools.org/xclear/structure" countDocs="123">
  <z:atts countAtts="102132">
    <z:a n="abstract" count="0/1"/>
    <z:a n="attributeFormDefault" count="0/2"/>
    <z:a n="base" count="0/816"/>
    <z:a n="block" count="2/2">
      <z:v v="extension"/>
      <z:v v="restriction"/>
    </z:a>
    <z:a n="default" count="0/12"/>
    ...
</z:values>
```

See appendix for the full source code of this operation, as well as module `items.mod.xq` as a whole, both its initial state as generated by `ttools` (Appendix A, *The `items` module as generated by `ttools`*), and the final state after our coding work (Appendix B, *The `items` module after editing it in the tutorial*).

Discussion

The present work is closely related to RESTXQ ([3]), and it is revealing to study similarities and differences. Both approaches reduce the writing of XQuery applications to the writing of XQuery functions, and in both cases infrastructure takes the responsibility to call the appropriate function. In both cases, infrastructure not only calls the function, but supplies it with information input which represents the information supplied by the triggering event. The approaches are, of course, concerned with different types of events – an HTTP request in the case of RESTXQ, a command line call in the case of topic tools. But both models are essentially models how to map an external event and its information content to (a) the selection of a function to be called, (b) input information accessible to the function called.

Many differences between RESTXQ and topic tools are consequences of the different types of events with which they are concerned. For example, RESTXQ deals with HTTP header fields, path segments and URI parameters, as these are natural parts of an HTTP request. Topic tools, on the other hand, essentially deals with a *single string* - user input without any pre-given structure, so that topic tools has the freedom – and responsibility – to define a structural model into which the string is translated and which transforms the string into structured information.

But there is one meaningful difference between RESTXQ and topic tools, and this is the *model of function input information*. RESTXQ makes an apparently obvious choice: function input is defined by the *function signature*. Each logical input parameter is a function argument, and the model of function input is a sequence of parameters defined in terms of a name and an XQuery sequence type.

Topic tools chooses a different input model: function input is an instance of the *operation interface* as defined by the tool scheme, instantiated as a message element. The logical input parameters are hidden behind message accessor functions, and the model of input is a set of parameters defined in terms of the tool scheme meta model.

This difference –

- “input model = function signature / input parameter = function argument” vs.
- “input model = operation interface / input parameter - supplied by message accessor”

has three significant consequences. The latter approach offers a *richer parameter model*, it implies a *stable function signature* and it enables a *functional abstraction* of parameter values. We inspect these aspects in turn.

The operation interface consists of parameters defined in terms of a name, a sequence type and further properties implied by the tool scheme meta model. These further parameters are above all facets. Facets extend the built-in item types in the same way as user-defined XSD types extend the built-in simple types. It is a key feature of topic tools to enable fine-grained validation controlled by facets independently of imported schemas. More generally speaking, uncoupling the model of an operation parameter from the concept of a function parameter thus enables a *richer parameter model*, partially implemented by code (facet validation), rather than being restricted to native language support.

The operation providing functions of topic tools have a uniform signature, accepting a single input parameter which is a message element. More important than the uniformity across different functions is the uniformity of a given function over time: topic tools imply a *stable function signature*. This facilitates agile development and the stepwise addition of features. Refinement of functionality typically entails the addition of further control parameters. If each input parameter is represented by a function argument, agile development means that the function signatures change often. Apart from that, operations offering rich functionality tend to have an ever growing list of parameters. If this implies an ever growing function signature, things become unwieldy. Who wants to handle functions with 10 or even 20 parameters?

Obtaining input parameters by calling a message function means a *functional abstraction*. This can be used in various ways. For example, it becomes possible to define a parameter as a huge document set without loading the complete set into memory: this can be avoided by using an accessor function which supports the retrieval of a subset. More generally speaking, the message interface enables lazy evaluation and modification of access details at access time.

On first sight, the use of object parameter types also seems to be a case of functional abstraction. But in fact it is not, as RESTXQ might, in principle, define the same mappings from strings to objects (elements) and support the use of such objects in the function signatures. The concept of object parameter types, however, creates a different aspect to be considered when evaluating RESTXQ and topic tools. The definition of object parameter types introduces a mapping of lexical entities to the logical parameters of operation input. So in fact we are dealing with *two* mappings: (a) from lexical input to logical input (aka operation parameters), and (b) from logical input to its representation in XQuery. Speaking about function arguments versus message object we dwelt on the differences concerning mapping (b). Remembering object parameter types, we take notice that topic tools puts also emphasis on mappings (a). It expresses them by the definition of object parameter types.

The tool scheme, however, is not concerned with these mappings. The mappings deal with lexical input, but the tool scheme is composed of logical parameters. For example, the tool scheme states that a certain parameter is a name filter; the tool scheme does not prescribe that it is constructed from a string with a certain syntax, this is only implied by the used typed, `nameFilter`. We arrive at a clear distinction between the tool scheme and the mappings used to supply it with actual values. The distinction opens an interesting vista: although this paper defined and viewed topic tools as a particular kind of command line tools, the core of the concept is not bound to the notion of a command line - it is bound to the logical model of an operation. There is an input source (for example – a command line) which is mapped to an instance of an operation interface. The type of input source may be exchanged for a different type, if we have access to an alternative mapping, translating this other type into the tool scheme. As the original mapping – as well as the hypothetical alternative mapping – is accomplished by infrastructure, the replacement would in no way affect the application code hand-written by the developer. For instance, the topic tools model might be employed as an alternative model for the development of RESTful web services. In order to accomplish this, it would suffice to add a sub model which maps HTTP requests to instances of an operation interface. This would be very easy, as the RESTXQ defined mappings could be reused, redirected from one target, function arguments, to another, logical operation parameters.

A. The `items` module as generated by `ttools`

The following listing shows the `items` module generated by `ttools` in response to the following call:

```
basex -b "request=add?dir=/projects/xclear, mod=items, ops=names paths values" /
```

```
(:
 : ------------------------------------------------------------------------
 :
 : items.mod.xq - Document me!
 :
 : ------------------------------------------------------------------------
 :)

(:~@interface
<interface>
   <operations>
      <operation name="names" type="node()" func="names">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="paths" type="node()" func="paths">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="values" type="node()" func="values">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
    </operations>
</interface>
:)

module namespace f="http://www.ttools.org/xclear/xquery-functions";
import module namespace tt="http://www.ttools.org/xquery-functions" at
    "_request.mod.xq",
    "_reportAssistent.mod.xq",
    "_nameFilter.mod.xq";

declare namespace z="http://www.ttools.org/xclear/structure";

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
```

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    return
        <z:names countDocs="{count($docs)}">{
           ()
        }</z:names>
};

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:paths($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    return
        <z:paths countDocs="{count($docs)}">{
           ()
        }</z:paths>
};

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:values($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    return
        <z:values countDocs="{count($docs)}">{
           ()
        }</z:values>
};
```

B. The `items` module after editing it in the tutorial

The following listing shows the `items` module as it looks after editing the generated prototype as guided by the tutorial. The listing is an example of a first version of a topic tool, as it can be developed within a couple of hours and yet be useful enough to be handed over to users.

```
(:
 : -------------------------------------------------------------------------
 :
 : items.mod.xq - a module for inspecting item names, paths and data values
 :
 : -------------------------------------------------------------------------
 :)

(:~@interface
```

```
<interface>
   <operations>
      <operation name="names" func="names" type="node()">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="scope" type="xs:string" fct_values="att, elem, all" defau
         <param name="names" type="nameFilter?"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="paths" func="paths" type="node()">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="paths" type="pathFilter?"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="values" func="values" type="node()">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep=";" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="nvalues" type="nameFilterMap(xs:integer)?"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
    </operations>
</interface>
:)

module namespace f="http://www.ttools.org/xclear/xquery-functions";
import module namespace tt="http://www.ttools.org/xquery-functions" at
    "_request.mod.xq",
    "_reportAssistent.mod.xq",
    "_nameFilter.mod.xq";

declare namespace z="http://www.ttools.org/xclear/structure";

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $scope as xs:string := tt:getParams($request, 'scope')
    let $filter as element(nameFilter)? := tt:getParams($request, 'names')

    let $atts :=
        for $n in distinct-values($docs//@*
            [not($filter) or tt:matchesNameFilter(local-name(.), $filter)]/loca
        order by lower-case($n) return <z:a n="{$n}"/>
    let $elems :=
        for $n in distinct-values($docs//*
            [not($filter) or tt:matchesNameFilter(local-name(.), $filter)]/loca
        order by lower-case($n) return <z:e n="{$n}"/>
    let $infoAtts := $filter/@source/attribute nameFilter {.}
    return
```

```
        <z:names countDocs="{count($docs)}">{
            $infoAtts,
            if ($scope eq 'elem') then () else
                <z:atts count="{count($atts)}">{$atts}</z:atts>,
            if ($scope eq 'att') then () else
                <z:elems count="{count($elems)}">{$elems}</z:elems>
        }</z:names>
};

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:paths($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $filter as element(pathFilter)? := trace( tt:getParams($request, 'paths
    let $paths :=
        for $p in distinct-values($docs//(*, @*)/f:path(.))[not($filter) or tt:
        order by lower-case($p)
        return <z:path p="{$p}"/>
    let $infoAtts := $filter/@source/attribute pathFilter {.}
    return
        <z:paths countDocs="{count($docs)}" countPaths="{count($paths)}">{
           $infoAtts,
           $paths
        }</z:paths>
};

(:~
 : Document me!
 :
 : @param request the operation request
 : @return a report describing ...
 :)
declare function f:values($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat')
    let $fmap as element(nameFilterMap)? := tt:getParams($request, 'nvalues')

    let $defaultLimit := 10
    let $atts := $docs//@*
    let $elems := $docs//*[not(*)][text()]
    let $attNames := distinct-values($atts/local-name(.))
    let $elemNames := distinct-values($elems/local-name(.))
    let $attValues :=
        for $name in $attNames
        let $limit := tt:nameFilterMapValue($name, $fmap, $defaultLimit)
        let $values := distinct-values($atts[local-name(.) eq $name])
        let $show := for $v in $values[position() le $limit]
                     order by lower-case($v) return <z:v v="{$v}"/>
        order by lower-case($name) return
            <z:a n="{$name}" count="{concat(count($show), '/', count($values))}
    let $elemValues :=
        for $name in $elemNames
        let $limit := tt:nameFilterMapValue($name, $fmap, $defaultLimit)
```

```
            let $values := distinct-values($elems[local-name(.) eq $name])
            let $show := for $v in $values[position() le $limit]
                         order by lower-case($v) return <z:v v="{$v}"/>
            order by lower-case($name) return
                <z:e n="{$name}" count="{concat(count($show), '/', count($values))}
        return
            <z:values countDocs="{count($docs)}">{
                <z:atts countAtts="{count($atts)}">{$attValues}</z:atts>,
                <z:elems countAtts="{count($elems)}">{$elemValues}</z:elems>
            }</z:values>
    };

    declare function f:path($n as node())
            as xs:string {
        string-join($n/ancestor-or-self::node()/concat(self::attribute()/'@', local
    };
```

Bibliography

[1] BaseX GmbH, Germany. Homepage.http://www.otds.de

[2] OTDS e.V., Schicklerstrasse 5-7, 10179 Berlin, Germany. Homepage. http://www.otds.de

[3] Retter, Adam. RESTful XQuery. XML Prague 2012, Conference Proceedings, p. 91-124. http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf.

[4] Robie, Jonathan et al, eds. XQuery 3.0: An XML Query Language (Second Edition). W3C Recommendation 8 April 2014. http://www.w3.org/TR/xquery-30/