FOXpath – an introduction

What is FOXpath?

FOXpath is an expression language for navigating the file system, selecting files and folders, and reporting their contents. A selection of file system contents has the look and feel of an XPath expression selecting XML nodes. Consider the following expression:

```
/programs/wildfly/descendant~::*.bat/parent~::*
```

It is a path consisting of steps, separated by slashes. Yet it is not an ordinary file path. Note the third step which consists of a *navigation axis* (`descendant`) and a *name pattern* (`*.bat`), separated by the characters `~::`. The step describes a downward movement, looking for files and folders in and at any level under the current folder (the `wildfly` folder) and matching the name pattern `*.bat`. The fourth step also consists of an axis (`parent`) and a name pattern (`*`); the step moves one level upward to the containing folder, whatever its name. One should realize that also the first two steps have a navigation axis, which is `child` but has been omitted for brevity's sake, as `child` is the default axis. An equivalent notation of our first FOXpath expression would therefore be:

```
/child~::programs/child~::wildfly/descendant~::*.bat/parent~::*
```

This verbose representation makes it easy to understand precisely how the expression is resolved to a value. In the following description, the term "current location" means the files and folders found so far.

- Start at the root folder, denoted by the leading slash (if there were no leading slash, navigation would start in the current working directory)
- Step #1: look for files and folders found in the current location (the root folder) and matching the name pattern `programs`
- Step #2: look for files and folders found in the current location (the `programs` folder) and matching the name pattern `wildfly`
- Step #3: look for files and folders found in or at any level under the current location (the `wildfly` folder) and matching the name pattern `*.bat`
- Step #4: look for the folders containing the current location (any of the `.bat` files and folders found by the preceding step)

In short, the expression defines a navigation which visits all `.bat` files and folders in or at any level under the `wildfly` folder and then moves up to the containing folders. The expression is intended to select all folders of a wildfly installation which contain `.bat` files. Though unlikely, however, there might also be `.bat` *folders*, in which case the selected folders are not guaranteed to contain `.bat` *files*. This uncertainty can be removed by using a *predicate* (a filtering expression put into square brackets):

```
/child~::programs/child~::wildfly/descendant~::*.bat[is-file()]/parent~::*
```

The same selection can be obtained by a different expression which uses nested predicates:

```
/programs/wildfly/descendant-or-self~::*[*.bat[is-file()]]
```

If the `wildfly` folder is the downloaded application server `wildfly 10.1.0` (renamed to `wildfly` for the sake of brevity), the expression value is in both cases:

```
/programs/wildfly/bin
/programs/wildfly/docs/contrib/scripts/service
```

The `fox` script

The examples shown in this article can be tested using the `fox` script found in the `bin` folder of the FOXpath project. The script expects a single parameter which is a FOXpath expression or a file containing a FOXpath expression (if option –f is used). The script returns the *value* of the FOXpath expression. So, for example, the command-line call

```
fox "/programs/wildfly/descendant-or-self~::*[*.bat]"
```

produces the paths of all `wildfly` folders containing `.bat` files. The double quotes are not part of the expression. They are delimiters which should always be used when passing FOXpath expressions as command-line parameters. When the expression,

```
/programs/wildfly/descendant-or-self~::*[*.bat]
```

is stored in a **file** (`scripts.fox`), the same result is obtained by the call

```
fox –f scripts.fox
```

Further options:

- use option –v `name=value` in order to pass an **input parameter** to the expression
- use option –b if the expression uses the **canonical syntax**
- use option –p if you want to inspect the **expression tree**, rather than evaluate the expression

The use of input parameters will be explained in section *Examples #5 – reusable tools*. The meaning of canonical syntax will be explained in the following section.

Key concepts of navigation

The examples demonstrate how FOXpath reuses the key concepts of XPath, applying them to the contents of the file system:

- selection is modeled as a sequence of **navigation steps**

- a navigation step combines a **navigation axis** (`child`, `descendant`, `parent`, …) with a **name test** (e.g. `*.bat`); note that steps without an explicit axis use the default axis, `child`
- a navigation step may be extended by **predicates** (an expression in square brackets) in order to refine the effected selection

FOXpath also supports the abbreviated syntax defined by XPath. Our examples can be written more succinctly:

```
/programs/wildfly//*.bat/..
/programs/wildfly//*[*.bat]
```

And here comes yet another, slightly more sophisticated example: the expression

```
/programs/wildfly//*[is-file()][not(ancestor~::modules)]
```

selects all files of the wildfly installation, excluding files found in or under the `modules` folder. Any step can have any number of predicates.

FOXpath is a superset of XPath 3.0

There is more to the similarity of FOXpath and XPath: FOXpath is a *superset* of XPath 3.0, which means:

- any XPath 3.0 expression is a valid FOXpath expression
- FOXpath can be used to select folders and files, but also to select XML nodes within files

At this point, an important detail concerning the syntax must be clarified. Our first examples used the forward slash as step separator, and XPath also uses the forward slash as step separator. This seems to create a dilemma. Step separators are not merely syntactic separators, but *operators* whose semantics control the flow of items from one step to the next, and the semantics required for connecting file system steps are different from the semantics required for connecting node tree steps. Being a superset of XPath, FOXpath needs both operators, and the slash cannot be used to denote both. Therefore FOXpath uses *different* step separators, one for connecting file system steps and another for connecting node tree steps: one is the forward slash, the other the backslash. Which is which?

FOXpath has a **canonical** syntax and a **friendly** syntax. The canonical syntax uses the forward slash as the separator of node tree steps, and the backward slash as the separator of file system steps. This means that XPath syntax is retained (everything means in FOXpath what it used to mean in XPath) and merely extended. The *friendly* syntax swaps the separator tokens: the forward slash separates file system steps, and the backslash separates node tree steps. Our initial examples used the friendly syntax and can be turned into canonical syntax simply be replacing all forward slashes by backslashes. If you prefer the canonical syntax, invoke the `fox` script using option –b. Example:

```
fox -b "\programs\wildfly\descendant-or-self~::*[*.bat]"
```

The friendly syntax is meant for the common case where the main interest is in file system navigation, and node navigation constitutes a special operation. As the forward slash is (arguably)

more convenient, the swapping of separator tokens is meant to make the language syntax as a whole more elegant and agreeable.

Now we can see how FOXpath allows a free combination of both types of navigation – file system navigation, and node tree navigation – even within a single path expression. Consider the following expression:

```
/programs/wildfly//standalone.xml\\*:extensions
```

It effects a mixed navigation: the first part is a file system navigation landing at the `standalone.xml` file somewhere under `wildfly`; and then it drills down *into* the file (assuming it is an XML document) and selects the `extensions` element. The value of the expression is therefore not a file path, but the selected XML element:

```
<extensions xmlns="urn:jboss:domain:4.2">
  <extension module="org.jboss.as.clustering.infinispan"/>
  <extension module="org.jboss.as.connector"/>
  <extension module="org.jboss.as.deployment-scanner"/>
   …
</extensions>
```

It is very important to understand that FOXpath is based on **XPath 3.0** – not XPath 1.0. Many developers who have a notion of XPath know only XPath 1.0. But XPath 1.0 – while being a huge achievement, setting up the basic principles of navigation (steps, axes, name tests, predicates) – is very limited compared to XPath 3.0. If you know only XPath 1.0, the formula "FOXpath = XPath + file system navigation" would be downright misleading, as you might ignore the *major* part of the power FOXpath has to offer. So let us briefly compare the main features of XPath1.0, XPath 3.0 and FOXpath and then look at a few examples, which should give you an idea about how to use FOXpath.

Comparison: XPath1 / XPath3 / FOXpath

The main differences between XPath 1.0, XPath 3.0 and FOXpath are summarized by the following table.

XPath 1.0	**XPath 3.0**	**FOXpath**
Very few kinds of expressions (path, function call, variable reference, literal, arithmetic, and/or, comparison)	> 30 kinds of expression	Added the `foxpath` expression for navigating the file system (and other resource trees)
Few standard functions (27)	Comprehensive library of standard functions (166)	Added about 50 further functions
A path step must be a navigation step (axis + node-test + predicates)	A path step may be a navigation step or any other kind of expression	As in XPath 3.0, pertaining to conventional path expressions as well as `foxpath` expressions
Primitive data model	Elaborate data model (XDM 3.0)	XDM 3.0

FOXpath examples

The final part of this introduction provides examples of simple and more advanced FOXpath expressions. These examples should give first ideas how to put FOXpath to interesting uses. The expressions explore an installation of the Oxygen XML IDE. As of version 18.0, this is a folder tree comprising 2058 folders and 13897 files. It can be downloaded from here:

https://www.oxygenxml.com/

The examples assume the following installation directory:

```
/Program Files/Oxygen XML Editor 18
```

The expressions can be resolved to their value using the `fox` script found in the `bin` folder of the FOXpath project. It expects a single parameter which is a FOXpath expression or a file containing a FOXpath expression (if option –f is used). See above, the section on the `fox` script for more details.

> *N O T E*
>
> *Expressions typed in on the command-line should always be enclosed by quotes.*

Examples #1 – basic navigation

The first group of examples demonstrates the basics of navigation:

- navigation steps consisting of axis, name test and predicates
- parentheses, `comma` and `except` expression.

As a simple start, let's list the `.exe` files found in the root folder of the installation. Expression:

```
/`program files`/`oxygen xml editor 18`/*.exe
```

[When typing in FOXpath expressions on the command-line, remember to enclose them in quotes.]

We obtain a list of 17 files:

```
/Program Files/Oxygen XML Editor 18/diffDirs.exe
/Program Files/Oxygen XML Editor 18/diffFiles.exe
/Program Files/Oxygen XML Editor 18/dotnetValidator.exe
…
/Program Files/Oxygen XML Editor 18/xmlGeneratorGui.exe
/Program Files/Oxygen XML Editor 18/xmllint.exe
/Program Files/Oxygen XML Editor 18/xsltproc.exe
```

Note the backquotes used in navigation steps containing blanks. Using wildcards, we get the same result with an expression that is more convenient to type:

```
/prog*files/ox*18/*.exe
```

Next, we want a list of all `.exe` files found *at any level under* the installation root. This is achieved by replacing the single slash preceding the last step by a double slash:

```
/prog*files/ox*18//*.exe
```

An equivalent expression uses the `descendant` axis explicitly:

```
/prog*files/ox*18/descendant~::*.exe
```

The result is a list of 38 files. Now we want to create a list of all *folders* containing `.exe` files. We just extend the previous expression by a further step moving one level upward, from the files to the containing folder:

```
/prog*files/ox*18//*.exe/..
```

An equivalent expression uses the `parent` axis explicitly:

```
/prog*files/ox*18/descendant~::*.exe/parent~::*
```

Interestingly, the result consists of only three folders:

```
/Program Files/Oxygen XML Editor 18
/Program Files/Oxygen XML Editor 18/.install4j
/Program Files/Oxygen XML Editor 18/jre/bin
```

The installation contains a large number of XML files. Rather than listing them, we would just like to know their number. Wrapping the appropriate path expression in a *function call*:

```
count(/prog*files/ox*18//*.xml)
```

we obtain the result `2263`. An equivalent expression uses the *arrow operator*:

```
/prog*files/ox*18//*.xml => count()
```

Next, we look for files matching a rather specific condition: all XML files which are not well-formed. We use a *predicate* which is only true for those files:

```
/prog*files/ox*18//*.xml[not(doc-available(.))]
```

Indeed, there are a few such files:

```
/Program Files/Oxygen XML Editor 18/frameworks/dita/DITA-OT/plugins/
      com.oxygenxml.webhelp/plugin_commons.xml
/Program Files/Oxygen XML Editor 18/frameworks/dita/DITA-OT2.x/plugins/
      com.oxygenxml.webhelp/plugin_commons.xml
/Program Files/Oxygen XML Editor 18/frameworks/dita/reuse/internal_subset_xsd.xml
/Program Files/Oxygen XML Editor 18/frameworks/dita/reuse/reuse_template_dtd.xml
/Program Files/Oxygen XML Editor 18/frameworks/dita/reuse/reuse_template_xsd.xml
/Program Files/Oxygen XML Editor 18/frameworks/docbook/xsl/
      com.oxygenxml.webhelp/plugin_commons.xml
```

Similar expressions produce lists of all empty files (7) and all empty folders (28):

```
/prog*files/ox*18//*[is-file()][file-size() = 0]

/prog*files/ox*18//*[is-dir()][not(*)]
```

Suppose we are interested in all empty folders *except* those located under the framework `dita`. We add a further predicate which yields only true if *upward navigation* to the `dita` folder (which is child of the `frameworks` folder) fails:

```
/prog*files/ox*18//*[is-dir()][not(*)][not(ancestor~::dita/parent~::frameworks)]
```

In order to list all XSD files contained by one of the frameworks: `xinclude`, `xlink`, `xslt`, the *comma* expression is handy:

```
/prog*files/ox*18//frameworks/(xinclude, xlink, xslt)//*.xsd
```

Note that the comma expression is placed in parentheses in order to ensure the correct evaluation order. Without the parentheses, the top-level expression would be a comma expression (with a last operand `xslt//*.xsd`), rather than a path expression! Formally speaking, the `comma` expression is wrapped in a ***parenthesized*** expression.

Using an `except` expression, we discover all XSD files contained by some framework, *excluding* the frameworks `dita`, `docbook`, `tei`:

```
/prog*files/ox*18//frameworks/(* except (dita, docbook, tei))//*.xsd
```

The last two examples demonstrate that a path step is not necessarily a navigation step expression, which consists of axis, name test and optional predicates. Here, we encounter steps which are a parenthesized expression wrapping a `comma` and an `except` expression, respectively.

In summary, the examples of the first group demonstrated some basic approaches how to select folders and files.

Examples #2 – appending a reporting step

All previous examples selected files or folders and produced a list of URIs (file paths). But often we would like particular information *about* the selected resources – not, or not only, their URIs. A simple approach is to append an additional step to the path expression – a "reporting step" – which produces for each selected item the desired information.

In the Oxygen installation, frameworks are represented by child folders of the `frameworks` folder. A list of framework names is thus just a list of folder names, rather than folder paths. The following expression:

```
/prog*files/ox*18//frameworks/*[is-dir()]/file-name()
```

produces the list of names:

```
ant
css_support
daisy
dita
docbook
…
xproc
xslt
xspec
```

Here, the reporting step consists of a *function call* which returns the file or folder name of the items selected by the preceding steps. Suppose we want more information about the frameworks – the name plus the number of XSDs. To achieve this, we just extend our reporting step:

```
/prog*files/ox*18//frameworks/*[is-dir()]/
 concat(file-name(), ' ', count(.//*.xsd))
```

The linefeed was just added for the sake of readability. The `concat` and `count` function are standard functions of XPath 3.0. We get this result:

```
ant 0
css_support 1
daisy 0
dita 568
docbook 41
...
xproc 0
xslt 2
xspec 0
```

Adding more function calls (`rpad` and `lpad`, returning a right/left padded string):

```
/prog*files/ox*18//frameworks/*[is-dir()]/
 concat(rpad(file-name(), 30, '.'), lpad(count(.//*.xsd), 5))
```

we produce a prettier list:

```
ant ..........................    0
css_support ..................    1
daisy ........................    0
dita .........................  568
docbook ......................   41
…
xproc ........................    0
xslt .........................    2
xspec ........................    0
```

Suppose we want to add more statistics – the counts of schematron (.sch), XML and XSLT files. To achieve this, we simply extend the last step accordingly:

```
/prog*files/ox*18//frameworks/*[is-dir()]/
 concat(rpad(file-name(), 30, '.'),
 lpad(count(.//*.xsd), 5),
 lpad(count(.//*.sch), 5),
 lpad(count(.//*.xml), 5),
 lpad(count(.//*.xslt), 5))
```

Here is the result:

```
ant ..........................    0    0    0    0
css_support ..................    1    0    1    0
daisy ........................    0    0    4    0
dita .........................  568    7  941    2
docbook ......................   41    5  987    0
…
xproc ........................    0    0    0    0
xslt .........................    2    0    4    0
xspec ........................    0    0    1    0
```

Examples #3 – creating reports

Up to now, the values of our expressions were a sequence of lines, where each line described one of the selected folders and files. But we may require different kinds of report, which are not one-by-one descriptions of selected resources. For example, desiring some kind of overview, we may be interested in a list of all file extensions. The following expression yields a sorted, de-duplicated list of all file extensions encountered in the Oxygen installation:

```
/prog*files/ox*18//*[is-file()]/file-ext() => distinct-values() => sort()
```

Result:

```
.1
.20020917
.JPG
.LIBXSLT
.LICENSE
.TXT
```

```
.access
.aff
.ai
.bat
.bfc
.bin
.bmml
.bmp
…
.xspec
.yml
.zip
```

Next, we would like to know for each file extension the number of respective files. To achieve this, we use a *FLWOR* expression. FLWOR expressions enable the assignment of intermediate values to variables (`let` clauses for a single assignment, and `for` clauses for iterative assignment), which are available in the expression producing the final value (`return` clause). Consider the following expression:

```
let $root := /prog*files/ox*18
let $files := $root//*[is-file()]
let $extensions := $files/file-ext() => distinct-values() => sort()
for $ext in $extensions
let $count := $files[file-ext() eq $ext] => count()
return
    concat(rpad($ext, 10, '.'), ': ', $count)
```

Certainly you would not like to enter such an expression on the command-line. Remember that you can store the expression in a file (e.g. `ext.fox`) and pass the file name to the `fox` script, using option option –f. Example call:

```
fox –f ext.fox
```

The call produces this result:

```
.1 .......: 1
.20020917 : 1
.JPG .....: 1
.LIBXSLT .: 1
.LICENSE .: 1
.TXT .....: 5
.access ..: 1
.aff .....: 5
.ai ......: 1
.bat .....: 23
.bfc .....: 1
.bin .....: 10
.bmml ....: 12
.bmp .....: 6
…
.xspec ...: 9
.yml .....: 1
.zip .....: 3
```

As FLWOR expressions can be nested to any depth, they enable very complex reports. The next example reports each framework in terms of …

- its name
- the number of its files
- all file extensions and the respective numbers of files

Here is the expression:

```
let $root := /prog*files/ox*18
for $framework in $root//frameworks/*[is-dir()]
let $name := file-name($framework)
let $files := $framework//*[is-file()]
let $headline :=
  concat('Framework: ', $name, '   (', count($files), ' files)')
let $extensionInfo :=
  let $extensions := $files/file-ext() => distinct-values() => sort()
  for $ext in $extensions
  let $count := $files[file-ext() eq $ext] => count()
  return
    concat('  ', rpad($ext, 11, '.'), ': ', $count)
return (
  $headline,
  $extensionInfo
)
```

It produces this report:

```
Framework: ant  (4 files)
  .dtd ......: 1
  .framework : 1
  .html .....: 1
  .jar ......: 1
Framework: css_support  (2 files)
  .xml ......: 1
  .xsd ......: 1
Framework: daisy  (20 files)
  .css ......: 2
  .dtd ......: 10
  .ent ......: 1
  .framework : 1
  .properties: 2
  .xml ......: 4
…
Framework: xspec   (30 files)
  .css ......: 1
  .framework : 1
  .html .....: 1
  .png ......: 3
  .rnc ......: 3
  .txt ......: 1
  .xml ......: 1
  .xproc ....: 6
  .xql......: 1
  .xsl ......: 12
```

Examples #4 – mixed navigation

As FOXpath is a superset of XPath, it supports navigation of node trees, along with navigation of the file system. Both kinds of navigation can be *combined* very effectively, even within a single path expression:

- steps selecting resources may have predicates navigating their contents
- leading steps selecting resources may be followed by trailing steps selecting resource contents

As an example of the first scenario, let us select schematron files containing quick fixes. Quick fixes are represented by elements with the local name `fix`. In the expression:

```
/prog*files/ox*18//*.sch[\\*:fix]
```

the predicate is false unless the resource is an XML document containing one or more `fix` elements. As FOXpath supports namespace declarations, we might also make sure that the `fix` elements belong to the correct namespace, like this:

```
declare namespace qf='http://www.schematron-quickfix.com/validator/process';
/prog*files/ox*18//*.sch[\\qf:fix]
```

How to get a sorted list of all quick fix titles? The following expression combines the selection of schematron files within the Oxygen installation with the selection of quick fix `title` elements within those files:

```
/prog*files/ox*18//*.sch\\*:fix\*:description\*:title
  => distinct-values()
  => sort((), lower-case#1)
```

and yields this result:

```
Add "http://" before the link
Add "https://" before the link
Add @format attribute
Add a parameter to the referred QuickFix (do not use in oxygen 17)
Add a reference to the first of the assert/report elements.
Add a reference to the second of the assert/report elements.
...
The fix title
Unwrap the sqf:fixes element
Wrap theelement into a sqf:fixes container.
```

An interesting possibility is to *extract nodes* from selected resources and *collect* them into a single document. Using the FOXpath function `xwrap`, we collect selected quick fix elements into a document, which we might perhaps use as a repository of reusable quick fixes:

```
/prog*files/ox*18//*.sch\\*:fix
  => xwrap('quick-fixes', 'b')
```

[The second parameter ('b') requests the addition of `xml:base` attributes to the extracted elements, identifying the source document.] Of course we might get more selective, e.g. collecting only quick fix elements whose title starts with "Move"). We just add a predicate:

```
/prog*files/ox*18//*.sch\\*:fix[.\\*:title\starts-with(., 'Move')]
  => xwrap('quick-fixes-move', 'b')
```

We obtain this document (abridged listing):

```
<quick-fixes-move countItems="6">
  <sqf:fix xmlns:sqf="http://www.schematron-quickfix.com/validator/process" ...>
    <sqf:description xmlns="http://purl.oclc.org/dsdl/schematron">
      <sqf:title>Move text in a shortdesc element</sqf:title>
      <sqf:p>Create a shortdesc element with the current paragraph text. And remove
the topic body.</sqf:p>
    </sqf:description>
    <sqf:add match="*[contains(@class, ' topic/body ')]" .../>
    <sqf:delete match="*[contains(@class, ' topic/body ')]" .../>
  </sqf:fix>
  <sqf:fix ...>...</sqf:fix>
  <sqf:fix ...>...</sqf:fix>
  <sqf:fix ...>...</sqf:fix>
  <sqf:fix ...>...</sqf:fix>
  <sqf:fix ...>...</sqf:fix>
</quick-fixes-move>
```

Examples #5 – reusable tools

FOXpath supports *external variables*. We may thus write reusable tools which can be applied to actual input. The following expression counts the folders and files found in and at any level under a root folder specified by the user:

```
declare variable $dir external;
concat('#dirs:  ', $dir/descendant~::*[is-dir()] => count()),
concat('#files: ', $dir/descendant~::*[is-file()] => count())
```

Using the `fox` script, external variables can be specified using option –v. After storing the expression in a script `counts.fox`, the call

```
fox -v "dir=/program files/oxygen xml editor 18" -f counts.fox
```

yields the result:

```
#dirs:  2058
#files: 13897
```

Note that the value of `$dir` is used as is – it is not resolved as a FOXpath expression. Therefore the value must be the exact path of a folder, including whitespace etc. This is inconvenient, and besides we are constrained to evaluate a *single* folder. But sometimes we might want to apply the counting

to a set of folders. We may significantly improve our script by treating the parameter value as a FOXpath expression which is resolved to a value. This can be achieved using the `fox` function:

```
declare variable $dir external;
let $dirVal := fox($dir)
return (
  concat('#dirs:  ', $dirVal/descendant~::*[is-dir()] => count()),
  concat('#files: ', $dirVal/descendant~::*[is-file()] => count())
)
```

Now we can apply the counting, for example, to a set of frameworks:

```
fox -v "dir=/prog*files/o*18/samples/*" -f counts.fox
```

We obtain:

```
#dirs:  83
#files: 530
```

Let us extend the script by introducing a second external variable, `$ext`, which can be used in order to specify file name extensions for which a separate file count is desired. Example call:

```
fox -v "dir=/prog*files/o*18/samples" -v "ext=xml xsd sch" -f counts.fox
```

Result:

```
#dirs:  108
#files: 538
#.xml:  93
#.xsd:  7
#.sch:  12
```

Here is the new version of our expression with its two external variables:

```
declare variable $dir external;
declare variable $ext external := ();

let $dirVal := fox($dir)
let $dirs := $dirVal/descendant~::*[is-dir()]
let $files := $dirVal/descendant~::*[is-file()]
return (
  concat('#dirs:  ', count($dirs)),
  concat('#files: ', count($files)),

  for $e in tokenize(normalize-space($ext), ' ')
  let $extension := concat('.', $e)
  let $count := count($files[file-ext() eq $extension])
  return
    concat('#', $extension, ':  ', $count)
)
```

Note the default value assigned to `$ext`, which is here the empty sequence. As a consequence, the expression can be invoked without specifying the variable, in which case no additional counts are output.

Examples #6 – foxlib

The final section of examples shows how to build and use libraries of FOXpath expressions. A *foxlib* is an XML file with a <foxlib> root element which contains <foxpath> elements providing FOXpath expressions. A <foxpath> element has a @name attribute specifying the expression name, a @doc attribute summarizing the functionality, and text content which is the FOXpath expression. Using the `fox` script, a particular expression from a foxlib is evoked by appending to the file name of the foxlib a # character followed by the expression name. Here comes an example of a little foxlib providing expressions for analyzing XSDs:

```
<foxlib>

  <foxpath name="cat"
           doc="Create a catalog of XSDs">
declare variable $xsds external;
fox($xsds)/descendant-or-self~::*.xsd
=> distinct-values() => dcat()
  </foxpath>

  <foxpath name="tns"
           doc="List target namespaces">
declare variable $xsds external;
fox($xsds)/descendant-or-self~::*.xsd\*\@targetNamespace
=> distinct-values() => sort()
  </foxpath>

  <foxpath name="topelem"
           doc="List top level element names">
declare variable $xsds external;
fox($xsds)/descendant-or-self~::*.xsd\*\xs:element\@name
=> distinct-values() => sort()
  </foxpath>

  <foxpath name="elem"
           doc="List element names">
declare variable $xsds external;
fox($xsds)/descendant-or-self~::*.xsd\*\\xs:element\@name
=> distinct-values() => sort()
  </foxpath>

</foxlib>
```

The foxlib defines …

- expression `cat` creating a catalog of all XSDs
- expression `tns` giving a list of target namespaces
- expression `topelem` giving a list of top-level element names
- expression `elem` giving a list of all element names

After storing the library in a file `xsd.foxl`, a list of target namespaces defined within the Oxygen installation, for example, is obtained by the following call:

```
fox -v "xsds=/prog*files/ox*18" -f xsd.foxl#tns
```

Note that the `$xsds` parameter is resolved to a value which is expected to consist of XSD files and/or folders containing XSDs.

Examples – afterword

Some of the examples illustrate that FOXpath enables an integrated view of distributed resources and their contents. A single expression may achieve a two-level selection – the level of resources and the level of resource contents. Combined selections on both levels grant the experience of selecting information within a single space of information, contributed to by distributed resources.