XQTav Reference Guide

Table of Contents

1. Introduction

XQTav [XQTav] is a software component that extends the Taverna [Tav] Workbench and makes it possible to use XQuery 1.0 [XQuery] inside Taverna workflows. The application introduces a new type of processor that executes XML queries using Saxon [Saxon] XQuery engine.

In an XQTav processor, the XQuery defines almost everything: input and output ports, their types, and of course the computation the processor is going to perform.

XQTav can also convert existing Taverna workflows (possibly including XQTav processors inside, too) to XML queries, performing almost the same computation as the original workflow. Almost the same as there are some limitations that will be covered later in this document.

This document explains some advanced topics concerning XQTav like implicit iterations, limitations of the XQTav XQuery generator and so on. There are also some examples here.

The software is available for download from *xqtav.sourceforge.net*.

The Authors are:

- Jerzy Tyszkiewicz, PhD, DSc, Warsaw University
- Andrzej Kierzek, PhD, Polish Academy of Sciences
- Jacek Sroka, MSc, Warsaw University
- Grzegorz Kaczor, MSc, Warsaw University

The examples presented in this document are also available in the XQTav release, under */examples* subdirectory. It may be necessary to run the examples in Taverna to fully undestand the description.

When an important term is introduced, there is always a bibliography link to that term. Some less important links are also placed directly in the text.

Warning. This document is not an introduction to XQTav. Please refer to the basic tutorial for introduction. Here we assume that you know Taverna well and have some basic experience with XQTav and XQuery.

2. Usage – an example

Let's start with a “real life” example of an *in silico* experiment that will show how XQTav can be used together with other Taverna processors.

--> /examples/filterBlastOutputByGo

2.1. An experiment overview

In the example experiment, we run *BLASTP* [BLAST] on a specified sequence and filter the results list by Gene Ontology [GO] ids specified as a parameter of our experiment.

The experiment has got three parameters described below:

1. *e-value-cutoff* the sequence similarity threshold value to filter out less similar sequences
2. *sequence* the amino acid query sequence that is passed to BLAST
3. *go_list* list of Gene Ontology identifiers we are interested in

The result of the experiment is a list of sequences in FASTA format that have at least one GO identifier specified on the *go_list* list.

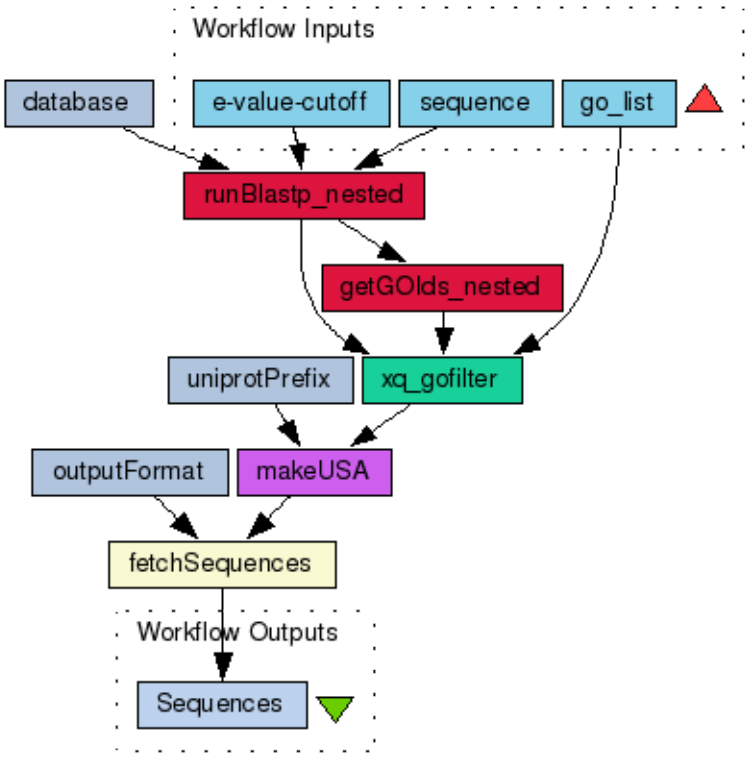
We split the computation into the following major parts:

1. run BLAST and fetch the results' accession numbers; we use DDBJ BLAST [DDBJ-BLAST] to achieve that
2. query the GO Browser [QuickGO] to get Gene Ontology ids associated with each of the result sequences
3. choose only that sequences that have at least one GO identifier on our list of interesting ids (the *go_list* parameter)
4. fetch sequence descriptions for the interesting sequences.

The Taverna workflow overview is shown in picture 1. The most important processors are:

1. **runBlastp_nested** – a nested workflow that wraps the DDBJ BLAST invocation
2. **getGOIds_nested** – a nested workflow that wraps the GO Browser query invocation
3. **xq_gofilter** – the XQTav processor that does the GO id lists filtering
4. **fetchSequences** – the processor that fetches sequences given an accession number

The additional processors (*database*, *uniprotPrefix*, *makeUSA* and *outputFormat*) are local Java processors and their role in the computation will be explained later.



Ilustracja 1. The example experiment - overview

2.2. Description of the workflow

Picture 4 shows more information about the workflow, including processors' input and output port names and data types.

The **runBlastp_nested** processor (that is in fact a nested workflow) populates three input ports:

- *sequence* – a sequence that is to be passed to BLAST – we pass a workflow input here; the input should be a sequence in FASTA format
- *database* – name of the database to query using BLAST – we use Swiss-Prot database so the input is *SWISS* here (saved in a string constant processor **database**)
- *exp* – maximum similarity value we accept – we pass a workflow input here; the input should be a float number, usually less than *1.0*; we can use, for example, *1e-4*, that is *0.0001*

The result of the processor execution is a list of sequence accession numbers. The list is passed to two processors – to **getGOIds_nested** and to **xq_gofilter**.

The **getGOIds_nested** processor (a nested workflow too) reads an accessor number passed to its *uniprot_ac* input and outputs a list of Gene Ontology ids associated with the protein identified by that number. As the values passed to the *uniprot_ac* input will be of type *l('text/plain')* and the formal type of the input is *'text/plain'*, an iterative invocation is needed – the **getGOIds_nested** processor will be called once per each accession number on the list being passed to the input.

The output of the **getGOIds_nested** processor (a list of GO identifiers, or, more strictly, a list of

lists of GO identifiers as the processor will usually be called using implicit iterations) is now passed to the **xq_gofilter**'s input *go_id_lists*. Please pay attention again: as the input type of *go_id_lists* is *l(l('text/plain'))* there will be no iteration here – even if the **getGOIds_nested** processor is called multiple times, the results will be collected into one list of lists – and that is what we expect to happen here.

The other inputs of **xq_gofilter** are *uniprot_acc_list* that is simply a list of accession numbers coming from **runBlastp_nested** and *go_list* holding a list of Gene Ontology ids we are interested in.

The **xq_gofilter** processor itself performs matching between its three inputs. In the *uniprot_acc_list* input port, it has got a list of accession numbers. In the *go_id_lists* input, there are lists of GO ids assigned to corresponding accession numbers on the *uniprot_acc_list* list. For example, the third element on the *go_id_lists* is a list of Gene Ontology ids assigned to the protein identified by the third accession number on the *uniprot_acc_list*. The **xq_gofilter** processor outputs only that accession numbers for which at least one of the assigned Gene Ontology ids is found on the id list being passed to the *go_list* port.

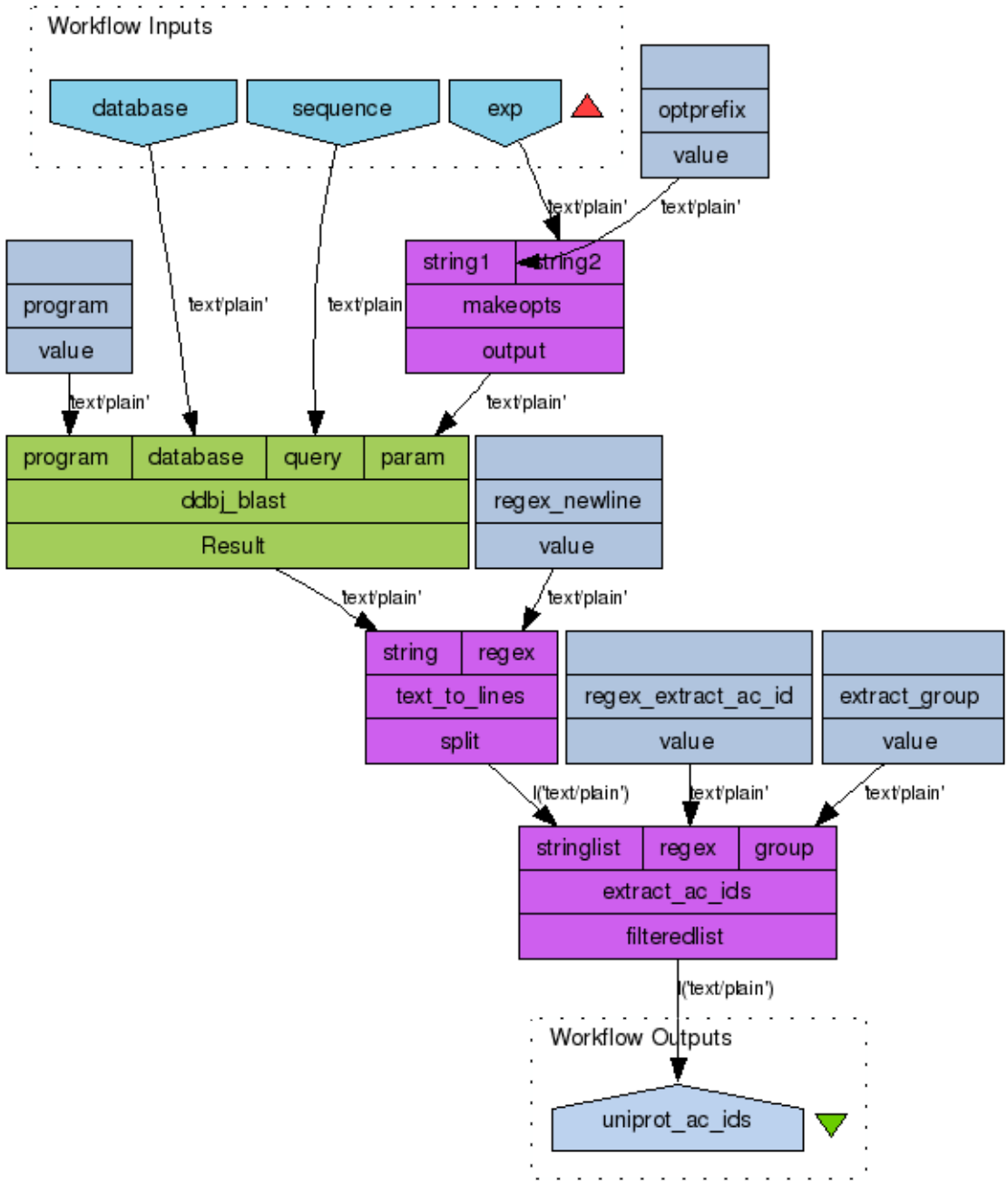
The last step is fetching the amino acid sequence for each of the accepted accession numbers. We do it via the *seqret* [SOAPLAB] **fetchSequences** processor and thus we have to compose a Uniform Sequence Address [usa]. This is done using the **makeUSA** (string concatenation) and **uniprotPrefix** (string constant: *uniprot:*) processors. The last processor, **outputFormat**, is a string constant with value *ncbi* and tells the **fetchSequences** processor that the sequences returned should be in *ncbi* format.

2.3. Description of the runBlastp_nested workflow

The workflow that constitutes the **runBlastp_nested** workflow is shown in picture 2. There is nothing special or difficult in the workflow so we will only have a brief look at its construction.

The core of the computation here is the *ddbj_blast* processor that executes the BLAST query. The output of the processor is a standard, text BLAST output format. So to extract the accession numbers, we have to do some filtering of the output. Two regex-based local Taverna processors are sufficient to do that – *text_to_lines* splits the output to a list of lines and *extract_ac_ids* select only lines that look like result sequences, at the same time capturing the accession numbers.

The rest of the processors are required to pass BLAST execution parameters to the *ddbj_blast* processor. These are *program* (*blastp*) and *makeopts* (*-eEXP*, where *EXP* is the similarity threshold value).



Ilustracja 2. The runBlastp_nested workflow

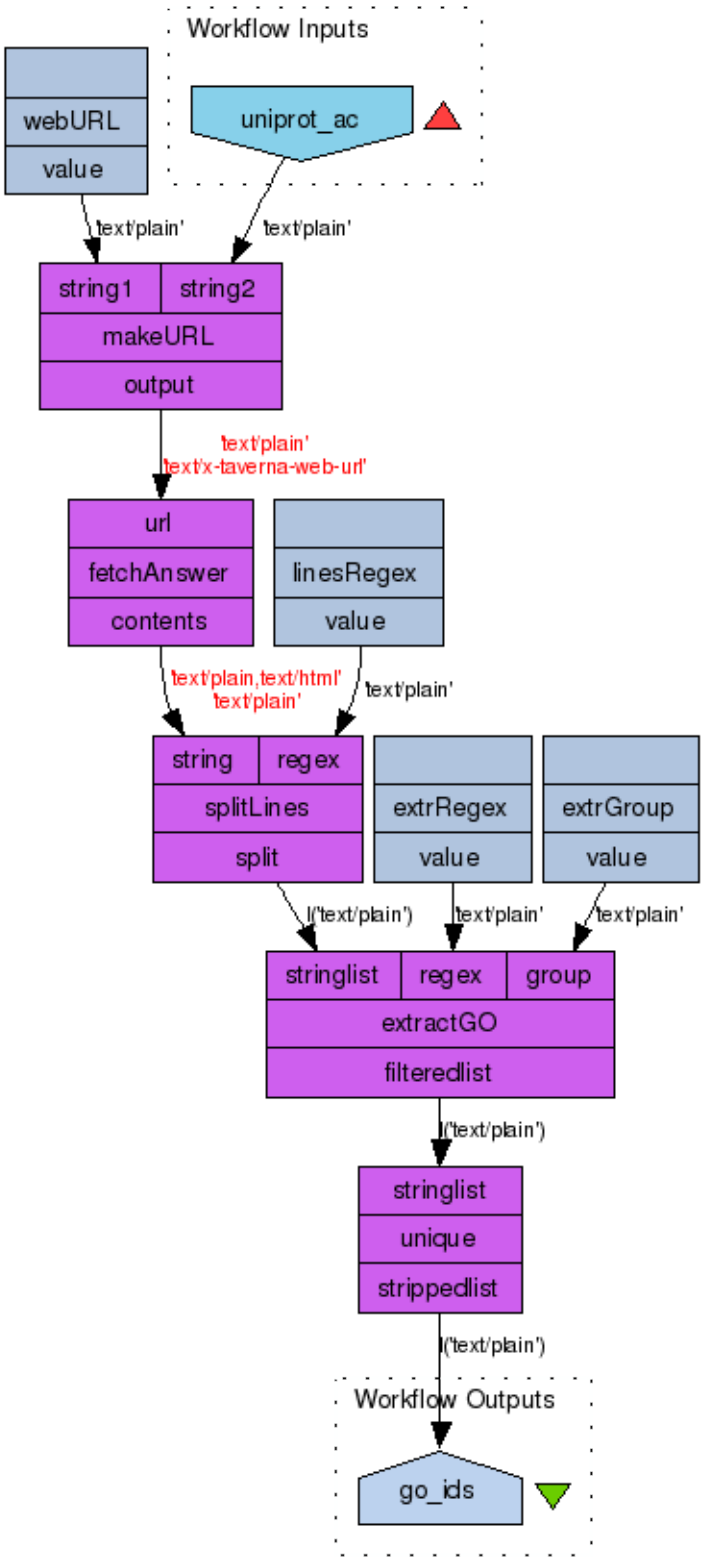
2.4. Description of the getGOIds_nested workflow

Picture 3 shows the **getGOIds_nested** workflow. The goal of the workflow is to get a list of Gene Ontology identifiers for a given accession number.

To do it, we compose a HTTP GET request to the QuickGO application and filter the output HTML with regular expressions to extract GO identifiers. The structure of the workflow is very similar to that of **runBlastp_nested** and we will not describe it in detail here. In contrary to **runBlastp_nested**, only local Taverna processors are used to get GO identifiers as the QuickGO application is accessed using HTTP only and not SOAP.

The output of the workflow is a list of GO identifiers assigned to the sequence identified by the

accession number passed to the *uniprot_ac* workflow input.



Ilustracja 3. The getGOIds_nested workflow

2.5. Description of the xq_gofilter XQTav processor

The **xq_gofilter** processor is an XQTav processor. This means that the computation it performs and

the port structure are defined using XQuery. Let's have a closer look at it.

We already know what the processor does. It takes three arguments:

- a list of text identifiers named *uniprot_acc_list*
- a list of lists of GO identifiers named *go_id_lists*
- a list of GO identifiers named *go_list*

And there is a correspondence between *uniprot_acc_list* and *go_id_lists*: for each index *i* the element *go_id_lists[i]* is a list of GO identifiers assigned to a protein identified by *uniprot_acc_list[i]*.

Now, the **xq_gofilter** processor first finds that sequences of GO identifiers from *go_id_lists* that have a non-empty intersection with *go_list* and then outputs that sequences from *uniprot_acc_list* that correspond to the found lists of identifiers.

The processor's code makes extensive use of XQuery FLWOR [FLWOR] expressions. Being similar to SQL but applicable to XML elements the FLWOR expressions form a quite convenient way of manipulating and filtering XML data. They seem also prone to optimalisation.

2.5.1. The code

Before any computation is done, there are two functions defined that wrap input lists into XML elements to simplify the further processing.

There is an auxiliary namespace defined, *x*, with URL *http://xqtav.sourceforge.net/ns/temp/x*:

```
declare namespace x = "http://xqtav.sourceforge.net/ns/temp/x";
```

This is obligatory as Saxon requires all functions to have a namespace prefix.

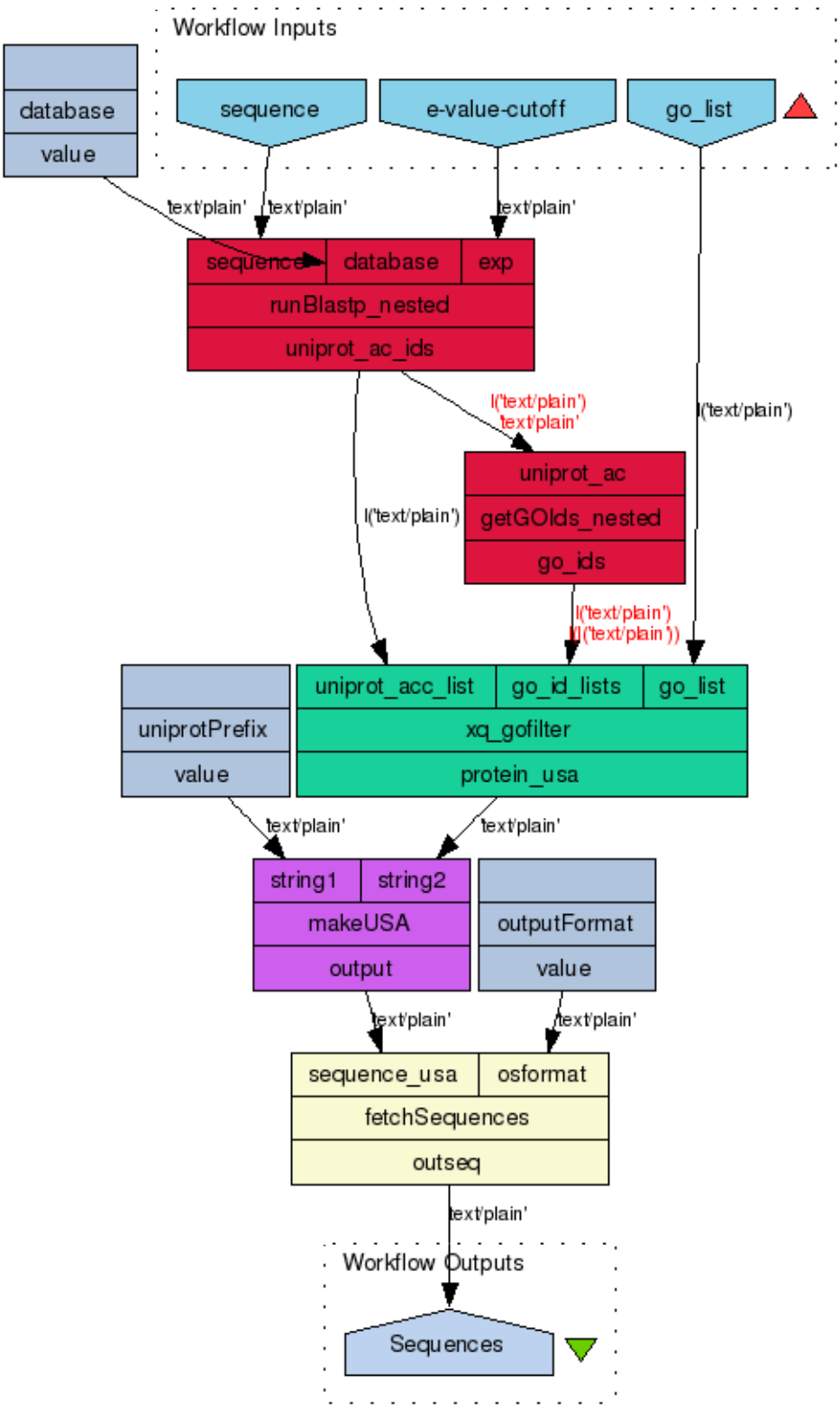
The first function, *x:tagAccSequence*, converts a sequence of strings to a sequence of *x:el* elements. Each such an element contains two subelements, *x:acc* containing the corresponding string from the input sequence and *x:num* containing the consecutive number.

That is, an input sequence

```
“a”,”b”,”c”
```

would be converted to

```
<x:el>
<x:acc>a</x:acc><x:num>1</x:num>
<x:acc>b</x:acc><x:num>2</x:num>
<x:acc>c</x:acc><x:num>3</x:num>
</x:el>
```



Ilustracja 4. The workflow with port names and types

Of course, the iteration here must be done using tail recursion:

```
declare function x:tagAccSequence($seq,$num) {
```

```
   if (fn:count($seq) = 0) then ()
   else (
     <x:el><x:acc>{fn:string($seq[1])}</x:acc><x:num>{$num}</x:num></x:el>,
     x:tagAccSequence(fn:subsequence($seq,2),$num+1)
  )
};
```

The second function numbers each element as well but it differs from the first one in two, very important details. First, its output are standard *IType* elements (for description of *IType* see chapter 3). Second, it replaces the wrapping element from each list item with *itype:p* – and we will use it to number the list of lists of GO identifiers – the *go_id_lists* argument of the processor. Why? Because we do not want to iterate over each element of *go_id_lists* to wrap it into *x* namespace element as it is a list of lists and can contain many elements.

```
declare function x:tagGoListSequence($seq,$num) {
  if (fn:count($seq) = 0) then ()
  else (
    <itype:p>
       <itype:num>{$num}</itype:num>
       {
            $seq[1]/*
       }
    </itype:p>
    ,
    x:tagGoListSequence(fn:subsequence($seq,2),$num+1)
  )
};
```

Now we can describe the main code of the processor:

```
(
  let (: *1* :)
    $go_list := fn:doc("taverna://itype/go_list")/itype:p
    ,
    $go_id_lists := fn:doc("taverna://itype/go_id_lists")/itype:p
    ,
    $uniprot_acc_list :=
      <x:taggedacc>{ (: *2* :)
        x:tagAccSequence(
          fn:doc("taverna://itype/uniprot_acc_list")/itype:p/itype:p/itype:i
          ,
          1
        )
      }</x:taggedacc>
  return
  (
    <itype:p name="protein_usa">{ (: *3* :)
      for $go_id_lists in $go_id_lists/itype:p
      return
        let $go_id_lists :=
          <itype:p>{ (: *4* :)
            x:tagGoListSequence(
              $go_id_lists/itype:p
              ,
              1
            )
          }</itype:p>
        return
```

```
            let $lists := $go_id_lists/itype:p (: *5* :)
              [
              for
                $current_goid in itype:i/fn:string(.)
                ,
                $allowed_goid in $go_list/itype:p/itype:i/fn:string(.)
              where
                $current_goid = $allowed_goid
              return "a"
              ]
            return
            (
              for (: *6* :)
                $acc in $uniprot_acc_list/x:el
                ,
                $p in $lists
              where
                fn:number($p/itype:num) = fn:number($acc/x:num)
              return
                <itype:i smode="none">{ (: *7* :)
                  fn:string($acc/x:acc)
                }</itype:i>
            )
      }</itype:p>
    )
)
```

The marks like that: `(: *1* :)` denote points in code we will have a closer look at.

1. `(: *1* :)` The processor inputs *go_list* and *go_id_lists* are read in a standard XQTav way, to *IType* format.
2. `(: *2* :)` The *uniprot_acc_list* input is tagged (numbered) just after being read, using *x:tagAccSequence*.
3. `(: *3* :)` A processor output value is constructed – it will be populated to the processor output port *protein_usa*.
4. `(: *4* :)` The list of lists of Gene Ontology identifiers from *go_id_lists* is tagged using *x:tagGoListSequence*; the output is a sequence of *itype:p* elements – each of them contains a number stored in an *itype:num* element and a list of *itype:i* elements containing GO identifiers.
5. `(: *5* :)` The main filtering is done in the condition brackets **[** and **]** using a FLWOR expression; trying to express it in human language, the expression says: *let $lists contain all such lists of GO identifiers (itype:p elements) from $go_id_lists that meet the following condition: if you take all elements from the list and all elements from the $go_list list, these two sets will have at least one common element*. The XQuery optimizer should be able to express it more cleverly.
6. `(: *6* :)` It is time to "join" the found GO lists with the *uniprot_acc_list* as we want to output sequence identifiers and not GO id lists. Now the tagging proves useful.
7. `(: *7* :)` Eventually, we output the sequence identifiers. Please note that the output type of the *protein_usa* port is *'text/plain'*. With little effort, the output type (and the processing code) could be changed to *l('text/plain')* to show statically that the iteration may take place. Please refer to chapters 3 and 4 for further information about typing and iterations.

2.6. Example summary

XQuery has been designed to facilitate the needs for convenient XML processing. The most convenient way of using XQTav is when there is XML data to be processed. You can save some time and simplify your workflow layout then. XQTav can sometimes prove useful when processing non-XML data but this happens only when that non-XML data can be at a relatively low price converted to XML format. If no, XQTav will probably be of no use.

You can sometimes use the generation mechanism described in the next chapter to generate the XQuery that performs a certain, complex processing, and try to further modify it to meet your needs. But the XQuery produced by the generator tries to preserve many properties of the source workflow and therefore may be hard to read or modify. And perhaps it would be easier and wiser to try to extract the parts of the functionality that cannot be done using Taverna and implement only those ones using XQTav.

3. Intermediate typing system (IType)

IType is an interface that defines the way the temporary data is passed from Taverna to an XQTav processor inputs and from the XQTav processor outputs back to Taverna. You can say *IType* defines a kind of serialization of Taverna data types to XML.

In XQTav, all temporary data is represented as *IType* XML elements.

3.1. Data types in Taverna

In general, Taverna data type system (called Baclava Data Model [BDM]) include simple types and lists of types. We can show this in a simple grammar-like definition[1]:

```
TYPE = PRIMITIVE | COLLECTION
PRIMITIVE = "'"<MIMETYPE>[","<MIMETYPE>]*"'"
COLLECTION = s|l|p|t"("PRIMITIVE|COLLECTION")"
```

The description above can be understood as follows.

- a Taverna data type can be a *primitive data type* or a *collection*, the primitive types are atomic units of Taverna data typing; the collections include sets, lists, partial orders or trees of any Taverna data type,
- a *primitive data type* is denoted by a string, enclosed in single quotes, composed of one or more MIME type names; a list of MIME types can be found at www.iana.org/assignments/media-types/,
- a *collection* is denoted by a letter *s*, *l*, *p* or *t* in front of an expression enclosed in brackets that defines the internal data type; the collections include sets (*s*), lists (*l*), partial orders (*p*) and trees (*t*); in fact, only lists and sets have been implemented.

Of course, most often used types are *'text/plain'* and *l('text/plain')*. You can also encounter *'application/octet-stream'*, *'text/xml'*, *'text/html'*, *'image/x-graphviz'*, *'image/jpeg'* and lists of that types. Every Taverna input and output port has a data type defined.

For a more in-detail description of Taverna data types please refer to a document that is available on Taverna website: taverna.sourceforge.net/index.php?doc=usingbaclava.html.

3.2. Mapping of Taverna data types to XML

IType defines a mapping between the Taverna data types and XML elements belonging to the *itype* namespace[2]. The mapping is performed as follows:

- a value of a primitive data type is converted to a single XML element named *itype:i*,
- a collection is converted to a single *itype:p* element with contents being a list of XML

1 http://taverna.sourceforge.net/index.php?doc=usingbaclava.html
2 the namespace URI is java:pl.edu.mimuw.xqtav.itype.TypeConvert

elements obtained by converting each of the collection items to *IType*.

The Taverna-*IType* conversion is performed in two stages. The first stage is a conversion from a Taverna *DataThing* to a Java object using the *DataThing.getDataObject()* method. During the second stage the XML elements are built.

In the other direction, the conversion is performed from XML elements to Java objects. The objects are then converted to Taverna data using the *DataThingFactory.bake()* method.

3.3. Encoding of primitive values

When it comes to primitive values, they can be simple text data as well as complex XML elements or raw, binary data. Thus *IType* must sometimes perform some kind of data encoding inside XML elements.

The type of encoding used is indicated by content of the *smode* attribute of an *itype:i* element. The four supported encoding types are:

- *none* (default encoding) – no encoding is used, to obtain the item data it is sufficient to extract text contents of the *itype:i* element,
- *binhex* (binary-to-hex simple mapping) – *used mainly to transfer wrapped Java objects*[3]; encoding: the object to be encoded is serialized using Java serialization to a byte array and each byte in this array is converted to a two-letter hexadecimal representation; decoding: this type of encoding should not appear when handling XQTav processor input or output values; you can use it when you use XQTav with Java extensions of Saxon[4],
- *dtxml-binhex* (Taverna serialization and *binhex*) – *used to transfer Taverna BDM Java objects*; encoding: the object to be encoded (which is a *DataThing*) is serialized using *DataThingSerializer.marshal()* method and the result byte array is converted to text just as in *binhex* encoding; decoding: should not appear during normal usage of Saxon (see *binhex* above),
- *xml* (XML element inside) – used only when returning XML elements as a result of XQTav processor computation; if you want to return a value of type *'text/xml'*, simply set the *smode* attribute of the proper *itype:i* element to *xml* – the element will be passed back to Taverna with all the namespace declarations properly included, all special chars properly XML-encoded and so on; encoding: the text value of the node is simply the XML, decoding: you will typically not need to decode such a value – however, if you do, use *pl.edu.mimuw.xqtav.itype.JDomConvert*. ***Note:*** *this encoding will not be used when reading value of an input port which type is 'text/xml' – see next sections for more details.*

As XQTav has been designed to work well with XML it will hardly ever become necessary for you to understand all the encoding described above. Even if you define an extension function and have to get *IType* Java objects as function arguments, the *TypeConvert.Saxon2IType()* method will convert the arguments to a more usable, abstract *IType* representation[5].

3 the objects to be serialized need to implement the *Serializable* interface

4 then use *pl.mimuw.edu.xqtav.itype.TypeConvert* to convert values between IType, Java and Taverna BDM

5 see package *pl.edu.mimuw.xqtav.itype*, classes: *SingleValue, ValueList*

3.4. Reading XQTav processor input values from XQuery

To read a value from an XQTav processor input port, you should use a default XQuery document import function, *fn:doc*. The protocol to use here is *taverna* and the host name is *itype* (for backward compatibility it can also be *workflow*) or *xml*. The path is the name of the port. All the rest of the URL is ignored.

So, an example call to get the value of an input port named *input* would look like that:

```
fn:doc("taverna://itype/input")
```

But here we have got a little problem. The result of the *fn:doc* function is a document node. We would like to get an element node instead. As all the values passed from Taverna to XQTav are lists (even if the value is not a list it is wrapped into a list to maintain type safety) we can simply select the *itype:p* element:

```
fn:doc("taverna://itype/input")/itype:p
```

And now everything is as expected – we have an element that contains a list of input values (most often there will be only one element in the list – the actual input value; but this is not always the case – see chapter 4). The elements of the list are *IType* elements that correspond to actual port input values.

Notice: *when you use the XQuery generator you might see in the result XQuery some very long expressions including constructions like that:*

```
sxl:iter_cross((sxl:makeArg(sxl:mrg(($inval_input))))))/sxl:call
```

This is to ensure proper query behaviour when implicit iterations must be performed. You do not have to use the sxl in your queries.

3.4.1. XML input values

When you read processor input using the method described above, even if the port's content-type is *'text/xml'*, you will get a plain text content. If you want to get the input data as an XML element that can be easily used within XQuery, you should change the processor input type to *'text/plain'* and use *xml* instead of *itype* in the call:

```
fn:doc("taverna://xml/input")/itype:p
```

In the element obtained as the result of the expression below you will get a typical *IType* element hierarchy with one exception: all the *itype:i* elements that had their *smode* set to *xml* will have been converted to real XML elements – they can be used in XQuery just like any other XML element defined inside the query.

This has one important implication – when you pass a not-well formed XML document to an input

that expects *'text/xml'* and use the expression above, an XML exception will be thrown.

3.5. Outputting results back to Taverna

You may want to pass an XML element to a XQTav processor output port so that Taverna received it as a correct *'text/xml'* data. To do that you should define an *itype:i* element into which you put the XML element you want to output. The *smode* attribute of the *itype:i* element should be set to *xml*.

3.6. Examples

In the *examples/itype* subdirectory of the XQTav distribution you will find a collection of simple examples concerning *IType*. We will discuss them in brief here.

3.6.1. Copying input to output

--> /examples/itype/simple_io/simple_io.xml

This example shows the simplest way the input port value can be copied to the output port using XQTav.

The query code looks like that:

```
(
   <itype:p name="output">{
     for $item in fn:doc("taverna://itype/input")/itype:p/*
     return $item
     }</itype:p>
)
```

What's happening here? The result of the query will be a document containing only one *itype:p* element. The element's *name* attribute specifies the output port name. The content of the element is the content of the *input* input port. No advanced transformations are performed here.

Let's run the workflow using the *simple_io_input.xml* file as input value.

If we load into Taverna the following example:

--> /examples/itype/simple_io/simple_io_dump.xml

we may see how does the output look like. Let's run the workflow with the same input (file *simple_io_input.xml*). The output looks like that:

```
<?xml version="1.0" ?>
<itype:p xmlns:itype="java:pl.edu.mimuw.xqtav.itype.TypeConvert">
  <itype:i class="java.lang.String" smode="none">a</itype:i>
</itype:p>
```

As we can see, the result is a list of items containing only one element. The element is a simple text and the contents are not encoded. If this was the real output of the previous workflow then the result

would be a one-element list while we had seen that only one single element had been returned. This is because the output port type is *'text/plain'* and not *l('text/plain')* – in such a case one-element lists are converted to single values. So the real output of the previous worklow was:

```
<itype:i class="java.lang.String" smode="none">a</itype:i>
```

The main reason for returning lists instead of single values is that some implicit iterations can occur during query execution. See chapter 4 for more details.

3.6.2. XML input-output

--> /examples/itype/xml_io/xml_copy.xml

This example copies its input to output, too. Additionally, it assumes the input is a well-formed XML file.

Let's have a look at XQuery:

```
 <itype:p name="output">{
    for $elem in fn:doc("taverna://xml/input")/itype:p/*
    return <itype:i smode="xml">{$elem}</itype:i>
}</itype:p>
```

Here we also define a single output *IType* element – but the contents now slightly differ from the previous examples. First, the input is read using *xml* host part of the URL and not *itype* or *workflow*. Second, the output *itype:i* element has got the *smode* attribute set to *xml*.

The *for* loop here ensures that in case of an explicit iteration everything goes well but it serves a second purpose too. When no input is passed to the processor the result will be an empty list.

If we used such a construction here:

```
 <itype:p name="output">{
    <itype:i smode="xml">{
          fn:doc("taverna://xml/input")/itype:p/*[1]
    }</itype:i>
}</itype:p>
```

the processor would fail with a runtime exception.

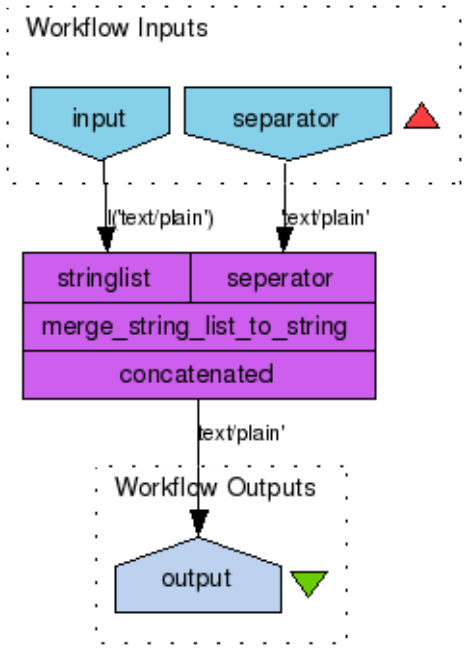
4. Input type mismatch – implicit iterations

--> /examples/type_matching/merge.xml

This chapter describes one of the most interesting Taverna mechanisms that is responsible for handling situations when the data type of a value being passed to processor's input does not match the input's data type. And when such a situation occurs for more than one input.

4.1. When is type-matching needed

Let's show on an example the basic concepts of that type-matching mechanism. Picture 5 shows a very simple workflow, performing string list concatenation using a given string separator.



Ilustracja 5. A simple workflow - concatenate a list of strings

Let's consider the following situations – for now, for a single input port:

- *The types match*; the situation occurs when you use file *input_match.xml* as an input – no type-matching action is needed, the *merge_string_list_to_string* processor is called only once and the result is as expected,
- *The input type is some kind of a structured collection of expected input type*; such a “structured collection” may be a list, a set, a list of lists of expected input types, and so on; use *input_struct.xml* to see an example; in such a situation some kind of an *iteration* should be performed; in general, Taverna calls the processor many times, and constructs the output result that will resemble structure of the input type; in our example, the expected input type for port *stringlist* was *l('text/plain')* while the actual one was *l(l(l('text/plain')))*; now, as the result type of the *merge_string_list_to_string* processor is *'text/plain'*, the output type after an iterative call will be *l(l('text/plain'))* – run the example to see the details.

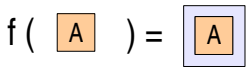
- *The expected input type is a structured collection of the actual input type*; you can see that using the *input_simpler.xml* input file; in such a situation Taverna wraps the input type to match the expected one.
- *The actual and expected types do not match*; then an error occurs; there is nothing interesting in an error, but you may see it in the *sample_mismatch.xml* workflow, by passing an input from *input_mismatch.xml*.

When the types match or when a wrapping into a list or set or even more complex type is required, it is clear how such a transformation could be done. The problems appear when it comes to the iterations.

4.2. Iteration methods in Taverna

The situation is relatively simple when there is only one argument that needs iterating. The workflow engine simply copies the input type structure to the output, replacing the leaf elements with matching type with results of multiple processor calls.

Let's show it in a picture. We will express calling a processor as calling a function that fills a blue frame with an argument – an orange square with a letter. So a simple processor call can be shown in picture below:



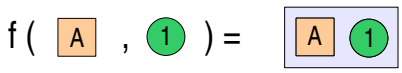
Now, expressing lists of items as white and yellow boxes, calling the same processor with a simple iteration on one argument will look like that:

As we can see, the result of calling a processor to a list of lists of inputs is like replacing each input in the list results of a function call on that input.

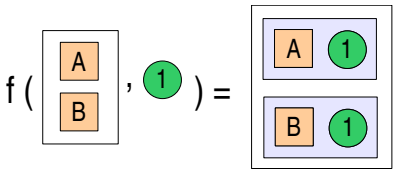
--> /examples/iteration/simpleExample1

The problem arises when there is more than one input that needs iteration. In such a situation, Taverna defines two kinds of basic iteration methods: the *cross* iteration and the *dot* iteration.

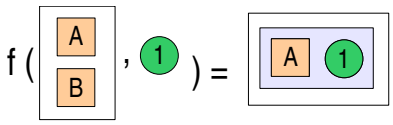
Let's assume that there are two inputs – two arguments of a function. If none of them needs type matching, the results is as in the picture:



The types matched – the result is as expected. When one argument only needs iterating, the result depends of the iteration method that is to be used. If the iteration method is *cross* then a function is called multiple times with the output type structure resembling the input type of the argument being iterated:



So, here the output is a list, as was the first input. When the iteration type is *dot*, a first element from the iterated input type is extracted and used for the only function call:

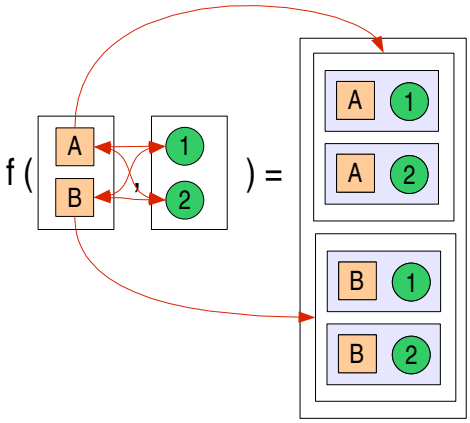


Opposite to no-iteration example, the output here is a list, just like in *cross* iteration. What's more – the output type will resemble the input type of the argument being iterated but, which is different from *cross* iteration, only the first element of the structure will be used in a function call.

The statements above are generally true when it comes to iterating over more than one argument.

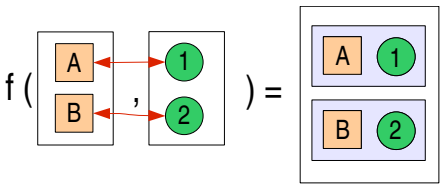
When we use a *cross* iteration, the all-to-all mapping is done and the output structure gets more complicated and deeper – one level per one argument being iterated. Things get even worse when we iterate over three and more arguments – use input *input_4l.xml* on the example */examples/iteration/strategyExample2/iteration4.xml* to see results of a *cross* iteration over four arguments.

An example of *cross* iteration over two arguments is shown in the picture:



The input structures had depth of one, while the result structure is one level deeper, with results grouped by the first argument of the function call.

When we use *dot* iteration the structure does not get deeper. The first elements of all structures being iterated are taken and a function call is performed. Now the second elements are used, the third and so on. An example is shown in the picture:



But be careful – both the cross and dot iterations preserve the most complex structure of the arguments. See example */examples/iteration/strategyExample2/iteration4dot.xml* with input *input_4l_h.xml* to see what happens if one of the lists being iterated has got a complex structure.

4.3. Iteration methods in XQTav

Due to iteration methods in Taverna being so complex the XQTav does not implement the iterations

the same way. We found it impossible to implement that without breaking the two main XQTav assumptions: (i) that as many of the processing as possible is done inside XQuery and not inside Java classes that form XQTav, (ii) the XQuery generated by XQTav is to be human-readable. See chapter 5 for a detailed explanation of these limitations.

Two invariants about iteration methods in XQTav are:

1. the iteration occurs when the actual type is a list or set of formal type
2. the iteration produces a list of formal type

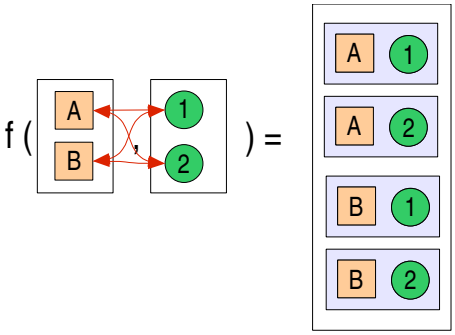
And there are very important implications of that. When types do not match, Taverna reacts by constructing an iteration. And it does it during runtime. We tried to reduce the decisions taken by XQTav during runtime to minimum as they cannot be easily expressed in a static XQuery script that forms the processor's code. So we cannot construct iterations in runtime and must create them in XQuery. But at the same time we do not know what will be the actual data types passed to the inputs. So we perform some “magic” conversions in the Java code. If the actual data type is a complex structure containing the formal data type then it is automatically flattened to match the type being a list of the formal data type. This has been hidden in Java and is not visible in XQuery.

Just the same applies to the situation when “wrapping” is needed – it is performed silently in Java code.

It would be possible to express the wrapping and flattening in XQuery but we think it would make the XQuery script much less readable.

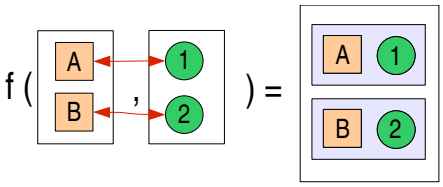
The simplification of the iteration model makes the *dot* and *cross* iterations return the same data types for the same input types. That makes them easy to implement in XQuery – the */sxl* tab in the XQuery editor contains two functions, *sxl:iter_cross* and *sxl:iter_dot*, performing the two types of iterations.

The *cross* iteration differs from the one in Taverna with that it does not wrap the parts of the all-to-all mapping in separate lists:



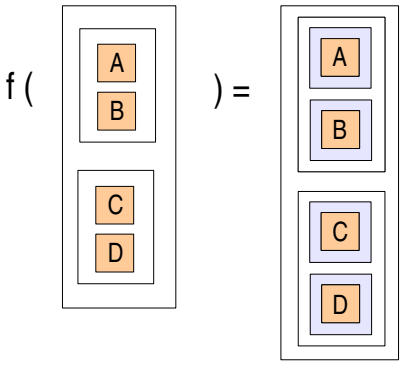
Instead, one list is outputted containing all the possible permutations of the arguments.

The *dot* iteration is same as the Taverna *dot* iteration as long as we remember that the iterations in XQTav cannot get arguments of any type:

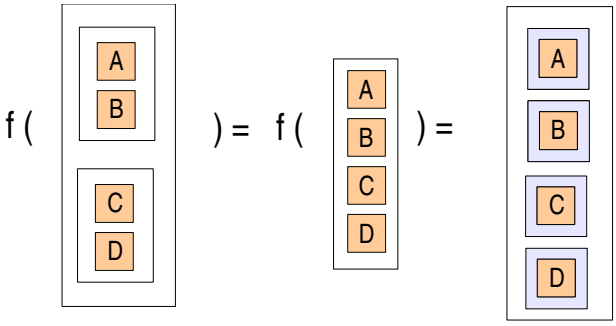


To illustrate that, let's compare the results of the Taverna and XQTav iteration on single argument:

Taverna keeps the type structure on the input type:



while XQTav will first flatten the argument to be of a proper type and then behave just as Taverna would in that situation:



Please note that the differences described above play the role only when it comes to workflows transformed to XQuery by the XQTav generator. If you use XQTav processor in a Taverna workflow and enact the workflow using Taverna, the differences are of no matter as all the iterations are done by the Taverna engine.

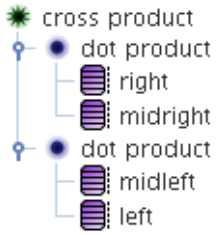
4.4. Iteration strategies

Taverna allows you to define more advanced iteration schemes for a processor than only: *apply cross iteration to all inputs* or *apply dot iteration to all inputs*. Such iteration schemes are called *iteration strategies.*

The introduction to iteration strategies is available in the Taverna manual (*taverna.sourceforge.net/usermanual/docs.word.html)*. An iteration strategy for a processor can be configured in the *Metadata* tab of the Taverna *Advanced Model Explorer*.

--> /examples/iteration/strategyExample2

Generally speaking, an iteration strategy is a tree. The internal nodes of the tree are labeled with *cross* or *dot* while the leafs are all inputs of the processor. An example iteration strategy is shown in picture 6.



Ilustracja 6. An iteration strategy

Such an iteration strategy says: when the cardinalities of inputs do not match, perform the following operations:

1. run a *dot* iteration on inputs *right* and *midright*
2. run a *dot* iteration on inputs *midleft* and *left*
3. run *cross* iteration on the results
4. for each leaf of the result structure, replace the leaf with the result of a processor call on the leaf (the leaf will be a tuple of arguments)

When no iteration strategy is defined for a processor, it is assumed that the iteration strategy is: *run cross on all arguments*.

4.5. Chapter summary

Iterations are a powerful mechanism and allow a really advanced processing control if you know how to use them. Complex iteration strategies are hardly seen in real life while simple ones, like iterating over one or two arguments, prove really useful and are often applied.

XQTav implements simplified iterations and this should be taken under consideration when generating XQuery from complex workflows. For more details, see chapter 5.

5. XQuery generator – limitations

The XQTav XQuery generator produces XML queries that, when executed, perform the same computation as the source workflow does. Currently, XQTav can properly convert almost any workflow to XQuery. However, there are some limitations: alternate processors, error handling, and advanced iteration usage.

When it comes to alternate processors and error handling, it is possible that such a functionality will be included in XQTav if there is such a need. Advanced iteration usage problems probably will not be solved.

5.1. Alternate processors

--> /examples/generator/alternate/alternate_xqtav.xml

As the alternate processor definition is located inside the processor's XML definition the XQTav processor does not remove them. But it does not handle the alternate processors, either. This would not be very hard to implement but might complicate the generated XQuery significantly.

Therefore generating XQuery from a workflow that contains processors with alternates is a bit risky. If no processor fails, everything will work well. But if any of the processors that contain alternates fails, none of the alternates will be called and that can lead to two possible situations: (i) the processing will bail out with an error message, (ii) some empty values will appear in the temporary results (see the example) that can lead to strange error messages and make the workflow hard to debug.

In the examples concerning alternate processors and error handling we use the 'echo with occassional failure' processor to show situations when is is necessary to use either of the mechanisms. The processor fails every subsequent four invocations.

5.2. Processor error handling

--> /examples/generator/errors/retry_xqtav.xml

Taverna allows you to define additional error-handling parameters of the processors. For example, you can set up maximum number of retries before the enactor will give up calling a non-responsive or badly working processor.

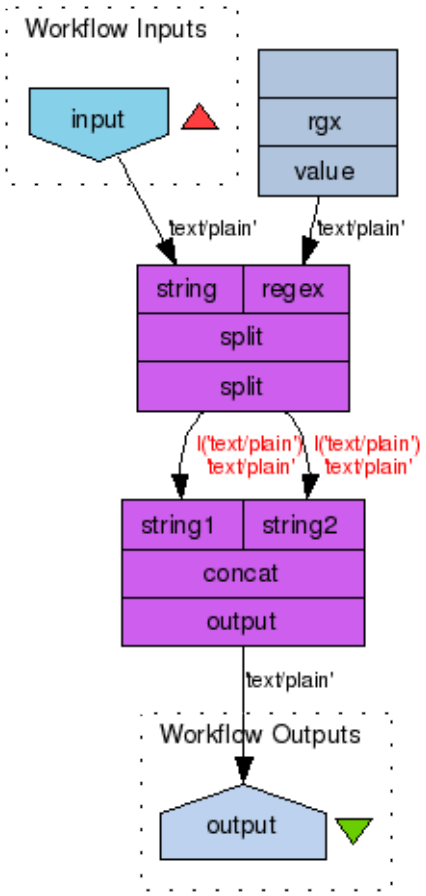
Again, because XQTav assumes human-readability of generated queries, these error handling parameters are ignored during generation. Handling them would require additional loops around processor invocations and, as failure to execute processor produces an exception, there would have to be some internal (in Java code) support for such methods of invocation. Of course, it is possible to implement error handling simply in Java code, but that is in contrary to one of the main XQTav assumptions – compute as much as possible in pure XQuery.

5.3. Advanced iteration usage problems

As it has been said in chapter 4, XQTav does not fully support Taverna iterations and has its own interpretation of *cross* iteration being simply a cross product of inputs. Due to the fact some sophisticated workflows that rely on the Taverna iteration mechanisms may behave differently when transformed to XQuery.

--> /examples/generator/iterations/proc2proc

Let's have a look at example */examples/generator/iterations/proc2proc/proc2proc_taverna.xml*. The workflow is shown in picture 7.



Ilustracja 7. A workflow that relies on processor-to-processor iteration

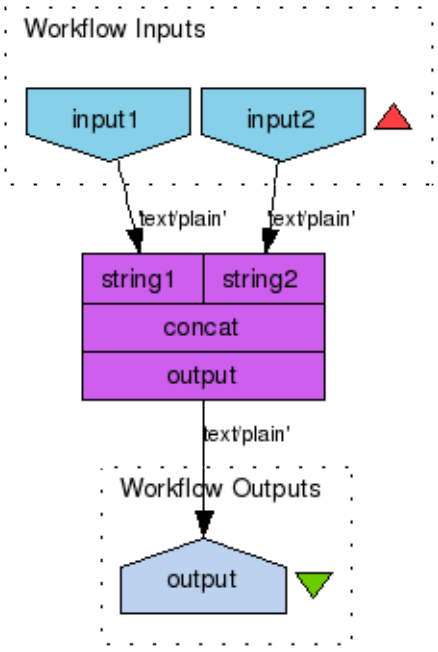
Let's execute the workflow with input from *input.xml*. In this example the type of the output is a list of lists (*l(l('text/plain'))*) in spite of formal type being simply *'text/plain'*. This is of course due to default *cross* iteration being applied on inputs of *concat* processor.

Of course, when you generate XQuery from this workflow and execute it with the same input (the workflow is available in file */examples/generator/iterations/proc2proc/proc2proc_xqtav.xml*), the output will be a simple list (*l('text/plain')*) as the iteration is now performed by XQTav-generated XQuery functions and not by the Taverna enactor.

Also, when you use input-to-processor iterations in your workflow, you should be extremely careful.

--> /examples/generator/iterations/input2proc

Here we have an extremely simple example. The workflow is shown in picture 8.



Ilustracja 8. Processor that can rely on input-to-processor iteration

The workflow is available in */examples/generator/iterations/input2proc/input2proc_taverna.xml*. Although there is nothing special in this workflow, there are two things the workflow diagram does not show – the iteration strategy for the *concat* processor is *run dot on all inputs* and the inputs that are saved in *input.xml* are two lists of strings. Therefore, a dot-iteration will be run on *concat's* inputs. And the output is, as expected, a list of two strings.

And now take look at the workflow that contains XQTav processor with XQuery generated from the workflow above. It is available in */examples/generator/iterations/input2proc/input2proc_xqtav.xml*. When we execute the workflow with input from *input.xml*, the result will be a list of two lists! That resembles the Taverna cross-iteration... because it is cross-iteration. The fact that the iteration strategy for *concat* processor is *dot* does not mean that the iteration strategy of the XQTav processor representing the whole workflow will be the same. It will be *cross*, a default iteration strategy. So the iteration between the old workflow's inputs and the *concat* processor inputs will never take place – it has been replaced with the iteration between the new workflow inputs and the *xqscript* processor inputs.

The problem above can be solved by simply changing the iteration strategy for *xqscript* to *dot on all inputs* but this may not always be so easy.

5.4. Chapter summary

The generator is quite powerful. It can handle complex workflows as well as nested ones and the ones that contain other XQTav-generated workflows. Thanks to some XQuery tricks it can also handle time synchronizations.

It cannot handle three things:

- alternate processors

- error handling parameters
- complex iteration-based workflows relying on a way Taverna performs iterations

Apart from that, the generator usage is limited by the complexity of generated XQuery. We tried hard to make it as simple as possible but it takes some time to get used to.

6. Executing XQTav-generated queries from command line

After you generate XQuery from an existing Taverna workflow using XQTav there are two possible ways of executing such a workflow. The first and the simpler one is to launch it from within Taverna – the XQuery is simply saved in the *xqscript* processor properties. The inputs and outputs of the processor correspond directly to the original workflow's inputs and outputs. The second way is to run the XQuery alone, without the supervision of Taverna.

In this chapter we cover the second way of executing generated queries.

6.1. Usage

-->/examples/showGeneOntologyContext/gene_ontology_xquery.xq

To launch a given workflow from command line using XQTav, you have to perform the following steps:

1. *convert the workflow to XQuery* – to do that, simply save the workflow you want to launch; reset Taverna editor, and add an *xqscript* processor to the model; now, right click on the processor, choose *Edit XQuery...* and use *Generator->Get XQuery from SCUFL*; as input select the workflow you have just closed; then save the generated XQuery to file (*XQuery -> Save as XQuery*);
2. *launch the XQTav command line tool* – change the directory to the *binary* subdirectory of the XQTav distribution; edit the file named *run_saxon_xqtav.sh*, set the *TAVERNA_HOME* directory so that it pointed to your Taverna installation path. Launch the workflow execution using the following syntax:

   ```
   ./run_saxon_xqtav.sh <xquery> <output> (<name> <value>)*
   ```

   where <*xquery*> denotes the path to the query saved in step 1, <*output*> is the output file name, and <*name*> and <*value*> stand for workflow input name and the value we want to pass to that workflow input. You may specify any number of such <*name*> <*value*> pairs.

   For example, if you want to run the example query, issue the command:

   ```
   ./run_saxon_xqtav.sh ../examples/showGeneOntology/gene_ontology_xquery.xq\
   /tmp/out.xml termID GO:0007601
   ```

   The result document will be saved in */tmp/out.xml*.

Note. You have to change directory to *binary* before calling *run_saxon_xqtav.sh*. The *gencp.pl* script is located in that directory. If you want to call *run_saxon_xqtav.sh* from anywhere in the system, you have to copy gencp.pl to a place somewhere in the system PATH and ensure that the TAVERNA_HOME variable in *run_saxon_xqtav.sh* is an absolute path.

If the workflow you are going to launch has got some inputs, XQTav will try to use the values you specified in command line. If you do not specify an input value for some input port, XQTav will show a message and try to get the input value from standard input.

Note. See *Limitations* below.

6.2. Semantics and limitations; type checking problem

So you can treat launching XQTav in *batch mode* as launching Saxon with a query being the XQTav-

generated XQuery representing a Taverna workflow. Of course, some initializations must be done, and they are performed in the *XQTavCommandLine* class. But just after XQTav initialization Saxon takes control and all the execution from that point depends on Saxon. There is no Taverna workbench behind, and XQTav does not know anything about the workflow that is being executed. This is different from the case when the XQuery is executed from within an *xqscript* processor, inside a running Taverna.

And from that difference comes the major limitation of command line execution. To understand it let's see what happens when you execute the generated XQuery from a running Taverna. When you generate XQuery from a workflow you save the generated XQuery in some *xqscript* processor. The processor then changes its input and output ports' names and types to match the inputs and outputs of the original workflow. During processor execution, the values passed to the *xqscript* processor input ports are passed as workflow inputs to the internal workflow represented as XQuery. We can be sure that the types passed to the workflow will be correct because XQTav performs type checking while reading of processor inputs.

In the command-line execution, the internal workflow is not saved in any *xqscript* processor. An input provider that handles the calls like *fn:doc("taverna://itype/input1")* knows only the URL passed to *fn:doc* and the TLS data that include the running XQTav engine, command line arguments and Saxon configuration. Therefore the input provider cannot tell the correct syntactic type for the workflow input identified by URL *taverna://itype/input1*.

The input provider (*ConsoleInputProvider.java*) wraps all non-list values into a list and does not perform any change on inputs whose type is a list. This does not save the user from type errors but is required as all input values should have list types (this restriction is about iteration, you may see that it is mandatory if we want to perform iterations inside XQuery).

There is a possibility to solve the limitation described above but for a price of breaking the rule that only static initialization is performed in XQTav and the rest is under custody of Saxon. Such a solution would be to scan the workflow XQuery before passing it to Saxon and extract the *XQTAV-IOINFO* structure comment (if it is present; if no, nothing can be done). Then all the workflow input port types would be known. That would have another drawback too as the user would have to get familiar with XQTAV-IOINFO and update the structure when *fn:doc* calls change. This would reduce the usability of the code and make it more error-prone.

One more restriction is that *ConsoleInputProvider* allows only strings to be passed as input values. This is not a technological limitation and can be changed if a more advanced input method is needed.

The reason for creating a separate class for command-line launch and not doing it on-the-fly (during the first processor call for example; such approach would allow us to launch Saxon directly from the console) is that to perform some internal type conversions XQTav needs the current Saxon dynamic configuration object. So XQTav must start Saxon and not vice versa.

Pictures

Bibliography

[BDM] - Baclava Data Model, *taverna.sourceforge.net/index.php?doc=usingbaclava.html*
[BLAST] - Basic Local Alignment Search Tool,
www.incogen.com/public_documents/vibe/details/blastp.html
[DDBJ-BLAST] - DDBJ-SOAP Project, *xml.nig.ac.jp/soapp.html*
[FLWOR] - FLWOR expressions, *www.stylusstudio.com/xquery_flwor.html*
[GO] - Gene Ontology Home, *www.geneontology.org*
[QuickGO] - EMBL-EBI QuickGO - GO Browser, *www.ebi.ac.uk/ego/*
[Saxon] - SAXON. The XSLT and XQuery Processor, *saxon.sourceforge.net*
[SOAPLAB] - SOAP-based Analysis Web Service, *www.ebi.ac.uk/soaplab/*
[Tav] - Taverna Workbench, *taverna.sourceforge.net*
[usa] - Uniform Sequence Address,
emboss.sourceforge.net/docs/themes/UniformSequenceAddress.html#usa
[XQTav] - XQTav - an XQuery extension to Taverna, *xqtav.sourceforge.net*
[XQuery] - XQuery 1.0: An XML Query Language, *www.w3.org/TR/xquery/*