yax - an XProc (XML Pipeline) Implementation

Joerg Moebius

description under construction
Release: 0.11.0 / 2008-03-03

Table of Contents

What is yax?

yax is an Java implementation of the XProc Specification, an XML Pipeline Language (XProc: An XML Pipeline Language W3C Working Draft 17 November 2006 [http://www.w3.org/TR/2006/WD-xproc-20061117/]). yax processes XProc Scripts like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT"
      yax:description="transforms 'a*' elements to 'b*' elmenents.">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="trans2" type="xproc:XSLT"
      yax:description="transforms 'b*' elements to 'c*' elmenents.">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
      <xproc:parameter name="transformer" value="Saxon6" />
   </xproc:step>
   <xproc:step name="trans3" type="xproc:XSLT"
      yax:description="transforms 'c*' elements to 'd*' elmenents.">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
      <xproc:parameter name="transformer" value="XSLTC" />
   </xproc:step>
</xproc:pipeline>
```

Quickstart

Prerequisites

The following packages have to be available (installed):

- Java (Version 5 or above)
- Graphviz - Graph Visualization Software [http://www.graphviz.org/],

  if want to use the visualisation features (pipeline trace, port trace).
- further XSLT processors (like saxon, xalan et al),

  if you want to use other processor(s) than the one (XSLTC), which comes with java.

Installation

Download yax from http://yax.sourceforge.net and unzip the file to an arbitrary <*yax-installation-directory*>.

Configuration

For the first run no special configurations are necessary.

If your environment still contains an installation of Graphviz you can set the *automaticOutput* attribute of the <*pipetrace*> and the <*porttrace*> preference within the preference file at <*yax-installation-directory*>/config/preferences.xml to 'yes'. Then you will get files containing the graphical representations of the pipeline.

First Run

On the command line change to the <*yax-installation-directory*> and start yax with:

```
# java -classpath lib/yax-0.7.jar net.sf.yax.Yax examples/example1.xproc examples/example1.xml
```

If all works well you will get a set of messages like the following. Which transformation processor appears in the messages depends on your actual environment configuration.

```
default transformer is set to system transformer.
automaticOutput of pipe trace is set to 'yes'.
automaticOutput of port trace is set to 'yes'.
suppress of process is set to 'no'.
step 'trans1': transformer Apache Software Foundation (Xalan XSLTC)(Version 1.0) is used.
step 'trans2': transformer Apache Software Foundation (Xalan XSLTC)(Version 1.0) is used.
step 'trans3': transformer Apache Software Foundation (Xalan XSLTC)(Version 1.0) is used.
Yax run sucessful completed.
```

This run generates several output file in the <*yax-installation-directory*>/examples directory:

- 1) <*input filename*>.output.xml

  contains the result of processing the pipeline.

- 2) <*pipeline filename*>.config.xml

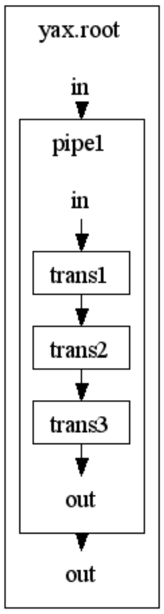
  contains the pipeline with all implicit ports and resulting connections between the ports and all used pipeline-library (the actual implementation is handled as serveral libraries):

```
<yax:configuration xmlns:yax="http://www.opsdesign.eu/yax/1.0">
  <!--configuration description generated by Yax - Do not edit by hand-->
  <pipeline name="pipe1">
    ..
    <step name="trans1">
      <output port="out" sequence="no" yax:creator="implementation.xproc.standard"/>
      <input port="in" sequence="no" yax:creator="implementation.xproc.standard">
        <yax:connection port="in" yax:component="pipe1"/>
      </input>
      <input port="stylesheet" sequence="no" yax:creator="implementation.xproc.standard"/>
      <input href="examples/transformation1.xsl" port="stylesheet" yax:creator="pipeline"/>
    </step>
    ...
  </pipeline>
  <xproc:pipeline-library name="xproc.options" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
    <xproc:declare-step-type type="Rename" yax:description="">
      <xproc:output port="out" select="" sequence="no" yax:bySourceRequired="no"/>
      <xproc:input port="in" select="" sequence="no" yax:bySourceRequired="no"/>
      <xproc:parameter name="name" required="yes" yax:values="{$any}"/>
      <xproc:parameter name="select" required="yes" yax:values="{$xpathExpression}"/>
    </xproc:declare-step-type>
    ...
  </xproc:pipeline-library>
  <xproc:pipeline-library name="xproc.standard" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
    ...
  </xproc:pipeline-library>
  <xproc:pipeline-library name="yax.standard" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
    ...
  </xproc:pipeline-library>
</yax:configuration>
```

- 3) <*pipeline filename*>.pipeTrace.png

  contains the compressed graphical representation of the pipeline. (This file will only generated if the *automaticOutput* attribut of the <*pipetrace*> preference is set to 'yes'.)

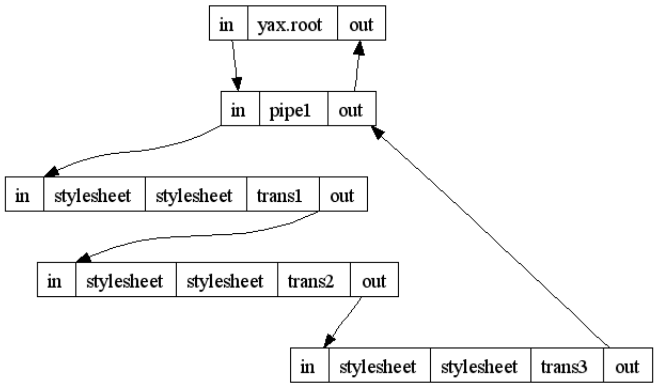
Figure 1. Compressed Graphical Representation of the Pipeline 'example1.xproc'



- 4) <*pipeline filename*>.portTrace.png

  contains the graphical representation of the pipeline which depicts the expected route the process will take over the ports. (This file will only generated if the *automaticOutput* attribut of the <*porttrace*> preference is set to 'yes'.)

Figure 2. Graphical Representation of the Pipeline 'example1.xproc'



Approach

The core idea of yax is to implement the XProc specification in a way that

a) makes it easy to follow the 'evolution' of the specification,

b) makes it easy to implement custom extentions and

c) makes some suggestions to the specification possible.

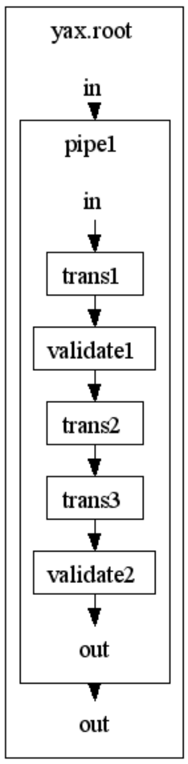
Inputs / Outputs

I assume for the majority of pipeline use cases the position of steps (and constructs) implies the chaining of input and output ports in the sequence of its appearance. That in mind it would be the easiest way to omit the explicit port entries where the intention of chaining the input and output ports is expressed by position of the components. Going this way it is necessary to write port entries only for the situation in which one want deviate the ordinary sequence.

In case of a closed pipeline structure the pipeline processing is quiet obvious:

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="validate1" type="xproc:Validate"/>
   <xproc:step name="trans2" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
   </xproc:step>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
   </xproc:step>
   <xproc:step name="validate2" type="xproc:Validate"/>
</xproc:pipeline>
```

Figure 3. Simple 'closed' pipeline

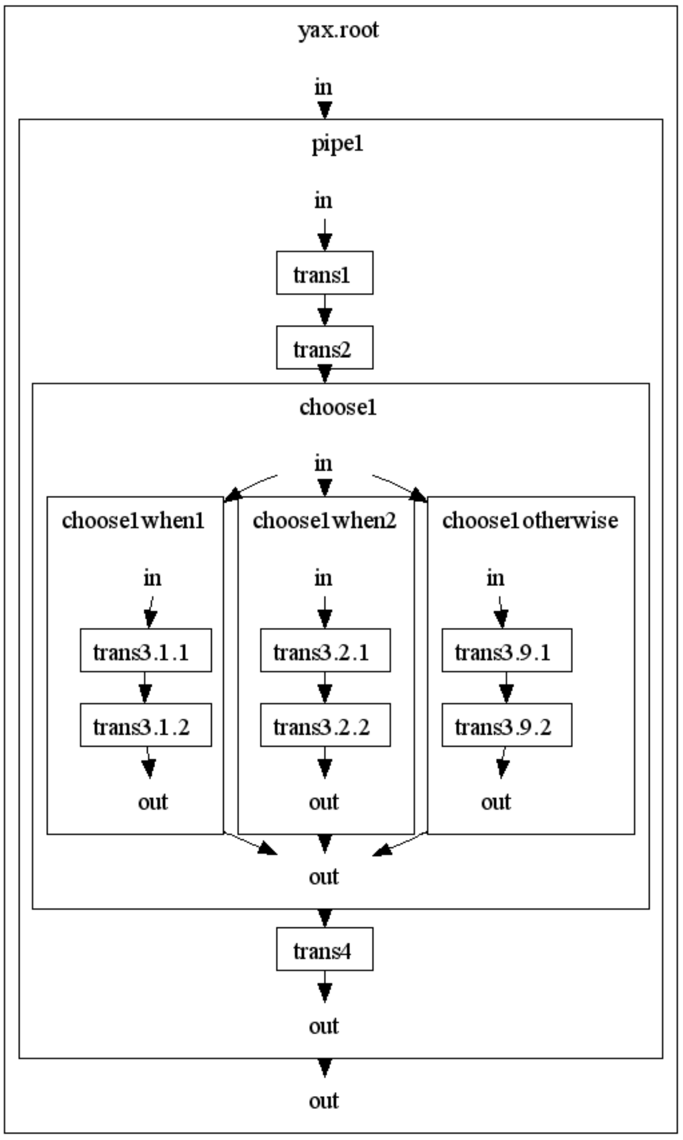


In case of using switching constructs like <*Choose*> <*Try/Catch*> the pipeline keeps closed and the pipeline processing as in the previous case:

```
   <xproc:pipeline name="pipe1"
      xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
      xmlns:yax="http://opsdesign.eu/yax/1.0">
      <xproc:step name="trans1" type="xproc:XSLT">
```

```
         <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
      </xproc:step>
      <xproc:step name="trans2" type="xproc:XSLT">
         <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
      </xproc:step>
      <xproc:choose name="choose1">
         <xproc:when name="choose1when1">
            <xproc:step name="trans3.1.1" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
            <xproc:step name="trans3.1.2" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
         </xproc:when>
         <xproc:when name="choose1when2">
            <xproc:step name="trans3.2.1" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
            <xproc:step name="trans3.2.2" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
         </xproc:when>
         <xproc:otherwise name="choose1otherwise">
            <xproc:step name="trans3.9.1" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
            <xproc:step name="trans3.9.2" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
            </xproc:step>
         </xproc:otherwise>
      </xproc:choose>
      <xproc:step name="trans4" type="xproc:XSLT">
         <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
      </xproc:step>
   </xproc:pipeline>
```

Figure 4. 'Closed' Pipeline with Switching Construct

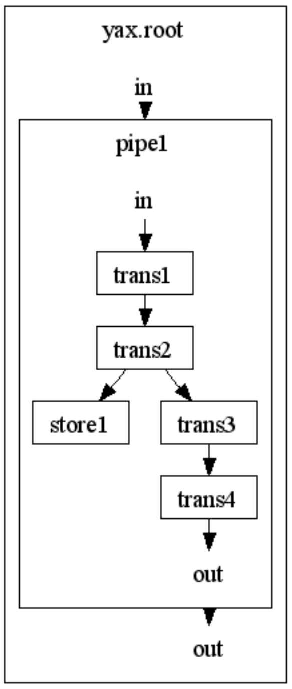


Components like <*Store*> and <*Load*> opens a pipeline.

<*Store*> (and all other output components) opens the pipeline backward by transfering the current result to somewhere, but the main trunk of the pipeline will be kept untouched.

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="trans2" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
   </xproc:step>
   <xproc:step name="store1" type="xproc:Store"/>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
   </xproc:step>
   <xproc:step name="trans4" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
   </xproc:step>
</xproc:pipeline>
```

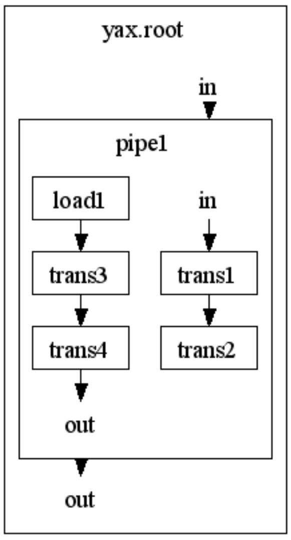
Figure 5. 'Backward Open' Pipeline with Output Component



Only <*Load*> (and all other input components) cuts the main trunk and and a decision how to continue is necessary. For instant yax simply cuts the trunk of the pipeline. The effect is - assuming c.p. - the result of the previous pipeline is simply discarded.

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="trans2" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
   </xproc:step>
   <xproc:step name="load1" type="xproc:Load"/>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
   </xproc:step>
   <xproc:step name="trans4" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
   </xproc:step>
</xproc:pipeline>
```

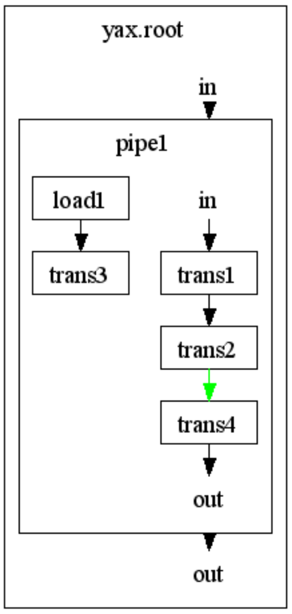
Figure 6. 'Foreward Open' Pipeline with Input Component



So using input components only makes sense if one uses expicit references to the output port of the preceding part of an input component.

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="trans2" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
   </xproc:step>
   <xproc:step name="load1" type="xproc:Load"/>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
   </xproc:step>
   <xproc:step name="trans4" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation4.xsl"/>
      <xproc:input port="in" step="trans2" source="out"/>
   </xproc:step>
</xproc:pipeline>
```

Figure 7. Foreward Open' Pipeline with Explicit Connection



Extension Mechanism

All component type classes have to reside in a package called 'net.sf.yax.types'. The built-in components are physically located in the yax library file *yax-<version>.jar*. They are distributed on three pipeline-libraries:

- xproc-standard
- xproc-options
- yax-standard

yax depicts all existant components grouped by libraries in the pipeline config file.

you can extend yax with step implemented by yourown by simply extend the class net.sf.yax.types.StepType and place the class file(s) of this implementation into the *custom* directory. For further details please refer to 'references/custom steps'.

Processing and Error Handling

ToDo

Issues concerning the Specification

Issues concerning the Specification

- Associating Documents with Ports / Specified by source

  Since constructs also have ports, their ports should also be reachable by referencing via 'by source'. So

  ```
  <p:step name="expand" type="p:xinclude">
    <p:input port="document" step="otherstep" source="result"/>
  </p:step>
  ```

  should become something like (better with a more handsome term for 'component'):

```
<p:step name="expand" type="p:xinclude">
  <p:input port="document" component="otherComponent" source="result"/>
</p:step>
```

- Associating Documents with Output Ports

  Are there use cases this kind of association makes sense?

- context of choose/when/otherwise

  the context should also be provided by a (e.g. reference) port.

- are there important reasons to differenciate between steps and constructs?

- Should a 'declare-construct-type' exists?

  In my understanding the only difference between constructs and steps is that steps treats the result while constructs only passes the result without any treatment. Otherwise both have to follow the rule for components.

  One can define custom 'step's. So should one also be able to define constructs.

- what expression language(s) (el) should be used in XProc scripts?

  For compatibility purposes an el should be defined by the specification. Otherwise the scripts runs only on processors which understands a certain el.

- Try/Catch

  Group within Try necessary?

  multiple Catch clauses would make sense.

- namespace uri for XProc

- base URI for XInclude and Store

  A pipeline might has more than one input document simultaniously via the root input and loads. How to determine a single base URI?

- Load / Store

  As Load should provide an unambiguous content on its *result* port it must not have other inputs than the source referenced by href.

  Store provides (or better delivers) an unambigous content to the external realm via the href. If Store would provide this content simulantiously an its *result* the pipeline chain could be kept closed at a Store step point.

Features

Table 1. Features

Feature	**Description**
common features	
logging	You can use either log4j or another logging subsystem. Is no logging system is determined the log will be written to the system ports.
Implementation of Constructs	

Feature	Description
<p:pipeline>	The <*Pipeline*> element should be implemented in accordance with the specification. The input port provides the documents to the contained components. The output port receives the result and provides it to the external realm.
<p:choose>/<p:when>/<p:choose>	The *<p:choose>/<p:when>/<p:choose>* element should be implemented in accordance with the specification. Beside the *source* and the *result* port *<p:choose>/<p:when>/<p:choose>* uses the *reference* port. You can define a *reference* port either for each *<p:when>* or a general *reference* port at the *<p:choose>* element (or a mix from both). The input port provides the documents to the contained components. *<p:choose>/<p:when>/<p:choose>* communicates only via the ports of the *<p:choose>* element with the external realm.
<p:try>/<p:group>/<p:catch>	The *<p:try>/<p:group>/<p:catch>* element should be implemented in accordance with the specification. The *<p:try>* element traps all exception which appears while processing the *<p:group>* elements content. If an exception appears the *<p:catch>* gets the control to continue the processing. *<p:try>/<p:group>/<p:catch>* communicates only via the ports of the *<p:try>* element with the external realm.
Implementation of Steps	
<p:XSLT>	The <*XSLT*> step should be implemented in accordance with the specification. The step receives the document, which is to treat via the input port and provides the transformation result at the output port.
<p:XInclude>	The <*XInclude*> step should be implemented in accordance with the specification. The step receives the document, which is to treat via the input port and provides the transformation result at the output port.
<p:Load>	The <*Load*> step should be implemented in accordance with the specification. The step loads an XML document from somewhere accessed via the URI expressed in the href attribute and provides this document at the *result* port. The URI might also reference to an database server or any other service, which is able to return XML documents. (yax will provide a

Feature	Description
	(experimental?) data source interface in one of the next releases.)
<p:Store>	The *<Store>* step should be implemented in accordance with the specification. The step store the content of a context to somewhere accessed via the URI expressed in the href attribute. The URI might also reference to an database server or any other service, which is able to return XML documents. (yax will provide a (experimental?) data source interface in one of the next releases.)

Status

yax currently provides a command line interface (see Quickstart and Reference). Other interfaces will follow in short.

All language components will be implemented step by step. Currently the most important components

- *<p:pipeline>*
- *<p:XSLT>*
- *<p:XInclude>*
- *<p:Load>*
- *<p:Store>*
- *<p:choose>/<p:when>/<p:otherwise>*
- *<p:try>/<p:group>/<p:catch>*

are usable. The other components still exist but have got an interims implementation, which passes a document through the pipeline process without treating it in the specification conformant way. Such an interims implementation is indicated by a warning message. You can omit the output of these messages by setting the preference standardImplementationWarnings to "no".

The interims implementation of language constructs resp. containers (controlling parts like *choose*, *try/ catch*, etc.) acts like the *pipeline* construct. It passes the treated document to the contained components, receives the result and provides it for further treatment.

The interims implementation of steps (active pipeline parst, which treat the documents like *XInclude*, *XSLT*, etc.), acts like the *Identity* step. It receives the treated document and provides it for further treatment without doing something by its own.

The first releases of yax will be published continuously in short intervals with the hope of vital feedback.

Table 2. Status of Implementation

Feature	Status	Annotation
Common Issues		
inputs and outputs	in process	implicit ports realized.
property / property files	done	

Feature	**Status**	**Annotation**
preferences / preference file	done	
Core Language Constructs		
Pipeline	done	
Group	done	same behaviour as Pipeline.
For-Each	planned	
Viewport	planned	
Choose/When/Otherwise	done	first usage of explicit reference ports: if exists reference ports of when clauses can refer to the reference port of the choose construct (or obviously th another reference port)
Try/Group/Catch	done	extended with differentiating catch clauses.
Import	planned	
Pipeline-Library	done	yax builds libraries based on the current implementation (see pipeline config file). CRs expected.
Core Steps		
Identity	done	
XSLT	done	CRs expected.
XInclude	done	
Serialize	in process	
Parse	in process	
Load	done	will be extended with access to database server and other data services.
Store	done	will be extended with access to database server and other data services.
Components Subelements		
input	(done)/in process	refer to *inputs / outputs* in this table.
output	(done)/in process	refer to *inputs / outputs* in this table.
Parameter	done	
input-parameter	??	didn't understood.
Mirco Operations Components (normal steps?)		
Rename	planned	
Wrap	planned	
Insert	in process	
Set-Attributes	planned	

Reference

Overview

Description ToDo

Prerequisites

The following packages have to be available (installed) on an arbitrary location:

Table 3. Required Packages

Package	Usage
Java (Version 5 or above)	Since Version 1.4 Java contains a default transformation processor (XSLTC). This processor will be used if no other processors are visible (in the classpath). yax assumes the presence of this processor. So at least Java 1.4 is necessary for using yax. As yas is developed under Java 5 it is the recommended Java version.
Ant(Version 1.7.0 or above)	yax comes with an ant task, so you can use yax also under ant. An ant installation is only needed if you want to use the yax ant task.
Log4j(Version 1.2.14 or above)	log4j is the recommended logging system. If log4j is not installed (resp. not reachable) yax uses its own ConsoleLogger. If you yax only under ant from yax perspective a log4j installation is not necessary because all log messages will be redirected to the ant logger.
Saxon [http://www.saxonica.com/](xls1: Version 6.5.3 or above, xls2: Version 8.6. or above)	yax provides the usage of different XSL processors. The saxon processors builds the preferred test environment for the yax development. (Obviously you can use any xsl processor which is reachable vie JAXP). The preferences file still includes the Saxon processors. If you want to use it follow Saxonias instruction how to obtain and install them. If no xsl processor is installed yax uses the system default transformer which comes with JRE (XSLTC).
Xalan /(XSLTC) [http://xml.apache.org/xalan-j/](Version 2.7.0 or above)	yax provides the usage of different XSL processors. Xalan comes from Apache. The preferences file still includes the Xalan and the XSLTC processor. If you want to use them follow Apaches instruction how to obtain and install them. If no xsl processor is installed yax uses the system default transformer which comes with JRE (XSLTC).
Graphviz - Graph Visualization Software [http://www.graphviz.org/]	yax provides the graphical visualisation of a pipeline in two ways. The *port trace* depicts the flow of the input(s) by drawing the expected usage of the components ports. The *pipeline trace* is a more compressed mode of the same matter.

Package	Usage
	For these features yax generates dot scripts and executes dot.exe of Graphviz to generate the pictures of the graphs. If you would like to use these features dot.exe must be installed and reachable (via system path).

Installation

Download yax from http://yax.sourceforge.net and unzip the file to an arbitrary <*yax-installation-directory*>.

```
<yax-installation-directory>/lib
|
+--- lib
     |
     +--- yax-<n.m>.jar (<n.m> = Version Number)
```

Configuration

Overview

You can have influence on the behaviour of yax over several ways:

- alter preferences within the preferences file at <*yax-installation-directory*>/config/preferences.xml
- pass particular properties at command line
- pass a name of a property file at command line
- use the standard property file 'yax.properties' at the current directory

Properties

Table 4. Property

Property	Description
common properties	
baseURI	The baseURI represents the base directory which is used by a relative path to become an absolute path.

Property Files

Preferences

Logging

yax preferred logging system is log4j [http://logging.apache.org/log4j/docs/]. yax comes with a log4j configuration containing a console appender and a rolling file appender which writes a log file each day.

To simplify the beginning the recent Log4j library comes with yax. You can find the library in YAX_HOME/lib.

In the root of YAX_HOME you can find the two bat-files 'runExample1.console.bat' and 'runExample1.log4j.bat. The two batches demonstrates who to use and how to avoid using log4j. Wether yax uses log4j or not depends simply on the fact whether yax finds log4j on the classpath or not.

While yax runs in ant mode yax redirects all log messages to ant's logger and all other log configuraiton take no effect.

classpath

For using yax in commandline mode 'yax-n.m.jar' (n.m = version number) must be reachable. if you want to run yax under ant the libraries 'yax-n.m.jar' and 'ant-yax.jar' must be reachable. Both of the libraries resides by default in YAX_HOME/lib but can be placed at any other location.

If you want to use any other xslt processor than the system default transformer you have to place their libraries into the classpath. In commandline mode you can either place the libraries into the system classpath or pass it via classpath option (-classpath resp. -cp) during the program start. When running yax under ant all necessary libraries must be assigned to the classpath BEFORE starting ant. Passing the transformer libraries via ant's classpath features (either within taskdef or with classpathref attribute) do not work.

XML Catalog

An XML catalog maps (usually remote) URIs to other (usually local) URIs.

yax uses Norman Walsh's resolver [http://xml.apache.org/commons/components/resolver/]. You can configure the resolver - especially the link to your catalog file - by editing the config/CatalogManager.properties file. How to configure the the resolver is explained in an excellent manner on the resolver's site [http://xml.apache.org/commons/components/resolver/]

For your confinience the yax distribution contains the recent resolver library (lib/resolver.jar). If you want to use the xml catalog feature the config directory and the resolver.jar have to be part of your the classpath. Start yax with the '-noCatalog' option If you want to suppress the xml catalog usage.

baseURI

Expression Language

At the present time the property reference mechanism known from http://ant.apache.org/manual/index.html is implemented.

You can use parameter references with the form *${<parameter name>}* within *href* attribute (or parameter). This reference will be replaced by the value of parameter with the same name passed either by property file or by start parameter ('-D<parameter name>').

With a combination of start parameter and parameter references within the xproc script you can build your scripts in a more flexible way. Example:

```
# java -classpath lib/yax-0.7.jar
       net.sf.yax.Yax
       -Dexample.dir=examples
       -Dtest.dir=test
       -DoutputFilename=output1.xml
       example1.xproc
```

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:step name="Load1" type="xproc:Load" href="${examples.dir}/${inputFilename}"/>
    <xproc:step name="Store1" type="xproc:Store" href="${test.dir}/${outputFilename}"/>
</xproc:pipeline>
```

Program Startparameter

Table 5. Features

Attribute	Description	Required
script	name of the XProc script to use - given either relative to the baseURI or as an absolute path.	Yes
in	specifies a single XML document to be treated. Should be used with the out attribute. for multiple input use a fileset.	No
out	specifies the output name for the treated result from the in attribute.	No. Must not be present, when 'in' is not present.
force	recreates output files, even if they are newer than their corresponding input files or the XProc script.	No
outDir	use in case of multiple input (via fileset). specifies the directory where the output files are to write to.	No.
outPattern	use in case of multiple input (via fileset). specifies how to build the output filename.	No.
noOutputfile	suppresses generation of an outputfile (overrides all other output parameters). The usage of this parameter makes sense when you only want to use inner output.	No.
noCatalog	suppresses the usage of an xml catalog.	No.
propertyfile	the properties this file contains will be passed to yax for further usage.	No.
baseURI	the base directory for all containing steps and containers.	No.
verbose	yax logs verbosely. For getting these log messages it is required that ant is also started in verbose mode.	No.

Attribute	Description	Required
quiet	yax only logs errors, warnings and very important informations.	No.

Usage

The Ways to Use Input and Generate Output

Overview

Input . There a (currently) two ways to access input data:

- use start parameters to pass input location(s)
- use the *<p:Load>* step.

Output . You can determine the output location(s) using the following solutions:

- **Omit any determination concerning output.**

  Then yax generates the output location(s) based on the input location. yax inserts *.output* into the input file name. Example:

  input location

  ```
  path/to/input/resource/inputname.xml
  ```

  output location

  ```
  path/to/input/resource/inputname.output.xml
  ```

- **Use the start parameters -out, -outDir- outPattern**

  yax generates the output location(s) based on these parameters. For further details see the sections 'Start Parameters', 'Commandline', 'Ant Interface'.

- **use the <p:Store> step.**

  This step stores the current state of the treated content. For further details see the section 'Using the Step *<p:Load>*'.

  By using the start parameter *-noOutputfile* you can restrict the output to the 'inner output'.

Outer I/O

All I/O effects by start parameters (or other interface input) is called outer I/O.

Passing only input file location start parameter . The most simple variant of an outer I/O is to use an input location parameter beside the mandatory xproc script location parameter:

```
java -classpath lib/yax-0.8.jar net.sf.yax.Yax examples/example1.xproc examples/example1.xml
```

yax completes the necessary information based on the input location. Assuming you pass the input file

```
path/to/input/resource/inputname.xml
```

derived from that input location yax generates the output location:

```
path/to/input/resource/inputname.output.xml
```

Passing input and output file location start parameter . The straight forward way of outer I/O is to pass an input location parameter and a corresponding output location parameter:

```
java -classpath lib/yax-0.8.jar net.sf.yax.Yax examples/example1.xproc
input/inputname.xml
output/outputname.xml
```

In that case the processing is obvious. yax takes the input document, passes it through the pipeline and writes the result to the ouput document.

Applying the XProc script on multiple input documents. . yax can also process multiple input documents in one run. For applying this feature it is recommended to use ant because in ant you can use the full flexibility of the fileset feature. Example:

```
...
  <target
    name="pipeline1"
    description="uses different multiple input files"
    >
    <yax
      verbose="yes"
      script="test/pipeline00.xproc"
      outDir="test"
      outPattern="${inputName}.out.${inputExtension}"
      >

      <fileset
        dir="examples"
        includes="
        example*.xml
        "
        excludes="
        example2.xml
        "
      />

    </yax>
  </target>
...
```

Alltough you can you this feature in commmandline mode. You pass a colon separated input file list with the -*inList* parameter:

```
java -classpath lib/yax-0.8.jar net.sf.yax.Yax examples/example1.xproc
-inList={input/inputname1.xml;input/inputname1.xml}
outDir=test outPattern="${inputName}.out.${inputExtension}
```

In case of processing multiple input documents it is not possible (or better makes no sense) to pass an output location for each input document. Therefore yax provides to other parameter for determining the output location(s).

With -*outDir* you can choose an arbitrary output directory>. If you this parameter as the only output parameter the processed documents will be written to the output directory with its input filename.

The other parameter is the -*outPattern* parameter. With -*outPattern* you describes the pattern for building the output file name. Example:

Assuming you are using the pattern:

```
outPattern="${inputName}.out.${inputExtension}
```

and you are processing the input files example1.xml and example3.xml by using ant's fileset

```
...
<fileset
  dir="examples"
  includes="
  example*.xml
  "
  excludes="
  example2.xml
  "
/>
...
```

yax will created the two file:

```
...
 examples
 |
 +-- example1.out.xml
     example2.out.xml
```

If you combine this parameter with the *-outDir* parameter determining the 'test' directory as output directory:

```
...

outDir="test"
outPattern="${inputName}.out.${inputExtension}"

...
```

yax writes the two files to the 'test' directory:

```
...

 test

 |
 +-- example1.out.xml
     example2.out.xml
```

Inner I/O

Beside the outer I/O which is effected by the usage of start parameters you can use the steps *p:Load* and *p:Store* to read resp. write xml documents during the pipeline process.

Although they are alternative instruments outer I/O and inner I/O they work simultaniously. If you process for example a pipeline which includes inner I/O without any output start parameter:

```
...
<target
  name="runExample5.1"
  description="uses inner and outer I/O"
  >
  <yax
    script="examples/example5.xproc"
    >
    ...
  </yax>
</target>
```

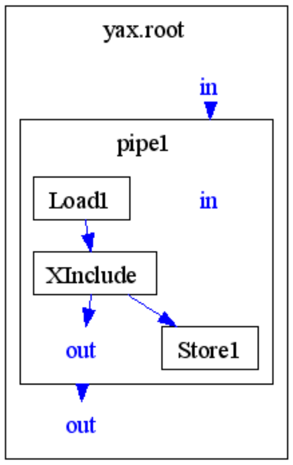
```
...
```

You will get two times the same output in different files:

```
...
INFO [Load:Load1] Reading input file '...\examples\xincludeArticle.xml' into Context '2'.
...
INFO [Store:Store1] Writing output file '...\test\example5.output.cmd.xml' from Context '2'.
...
INFO [root:yax.root] Writing output file '...\yaxOutput.xml' from Context '2'.
...
```

This is triggered by a)*p:Store* step and b) the automatic ouput determination of the *root* element. The following graphic shows this circumstance:

Figure 8. Outer and Inner I/O



The *p:Store* step creates a side branch but yax ports the content through the hole pipeline to the *root* container which is manages the outer I/O.

To avoid such undesired output behaviour (assuming in this case the outer output is undesired) you can use the -*noOutputfile* parameter:

```
...
<target
  name="runExample5.2"
  description="uses only inner I/O"
  >
  <yax
    noOutputfile="yes"
    script="examples/example5.xproc"
    >
    ...
  </yax>
</target>
...
```

In this case the content on the output port of *root* will be discarded.

Command Line Interface

On the command line change to the <*yax-installation-directory*> and start yax with:

```
# java -classpath lib/yax-0.7.jar net.sf.yax.Yax [options]pipeline file location [XML input data file loca
```

If all works well you will get a set of messages like the following. Which transformation processor appears in the messages depends on your actual environment configuration.

```
element 'yax:transformers' has ...something about transformer preference
...
step 'step name': something about transformer which will be used
...
messages from each step
...
Yax run sucessful completed.
```

If you start yax with the option *-h* or without any start parameter you will get a message about the usage alternatives like:

```
Usage: java yax [options] pipelineURI [inputfileURI [outputfileURI]]

Options:

-h, -help               print this message (and exit)
-version                print version information (and exit)
-quiet, -q              be extra quiet
-verbose, -v            be extra verbose (quiet takes precedence)
-debug, -d              print debugging information
-baseURI=<value>        set the baseURI to value
-noOutputfile           suppress generation of an outputfile (overrides all other output parameters)
                        (make sense if output is created within the pipeline)
-D<property>=<value>    provide property for using within the xproc script as '${property}'
-propertyfile <name>    load properties from file (-D<property>s take precedence)

(for further usage alternatives see http://yax.sourceforge.net/)
```

Warning

debug effects that some steps (especially XSLT) logs the hole content of a context. So use thit option with care. At best use only small inputs when you use the debug option.

This run generates several output file within the *pipeline file directory* and within the *XML input data file directory*:

- 1) <*input filename*>.output.xml

  contains the result of processing the pipeline.

- 2) <*pipeline filename*>.config.xml

  contains the pipeline with all implicit ports and resulting connections between the ports and all used pipeline-library (the actual implementation is handled as serveral libraries).

- 3) <*pipeline filename*>.pipeTrace.png

  contains the compressed graphical representation of the pipeline. (This file will only generated if the *automaticOutput* attribut of the <*pipetrace*> preference is set to 'yes'.)

- 4) <*pipeline filename*>.portTrace.png

  contains the graphical representation of the pipeline which depicts the expected route the process will take over the ports. (This file will only generated if the *automaticOutput* attribut of the <*porttrace*> preference is set to 'yes'.)

Yax Ant Task

Overview

the usage of the yax ant task is designed with the XSLT task [http://ant.apache.org/manual/CoreTasks/style.html] in mind. if you are familiar with the XSLT task you will find some similarities. For the determination of single input and output files the attributes *in* and *out* are used. The attributes which holds the script file location is called in the XSLT task *style*. The corresponding attribute in yax is called *script*.

As the XSLT task do not provide multiple input (and output) yax provides an own solution using ant's well known fileset feature in collaboration with the *outDir* and/or *outPattern* attributes.

Corresponding to the XSLT task yax also provides passing parameters to the pipeline processor and the used subsystems (as transformers). While these parameters are not exclusively passed for the usage in transformers to yax the different name 'property' was used. But these 'properties' will also be passed as parameters to the transformers.

If you intend to use other xslt processors than the system defualt processor please consider that all necessary libraries must be assigned to the classpath BEFORE starting ant. Passing the transformer libraries via ant's classpath features (either within taskdef or with classpathref attribute) do not work.

```
...
<target
  name="runExample1"
  description="transforms an input file with concatenated transformation steps"
  >
  <yax
    in="examples/example1.xml"
    out="test/example1.output.ant.xml"
    script="examples/example1.xproc"
    >
  </yax>
</target>
...
```

Description

Parameters

Table 6. Features

Attribute	Description	Required
script	name of the XProc script to use - given either relative to the baseURI or as an absolute path.	Yes
in	specifies a single XML document to be treated. Should be used with the out attribute. for multiple input use a fileset.	No
out	specifies the output name for the treated result from the in attribute.	No. Must not be present, when 'in' is not present.
force	recreates output files, even if they are newer than their corresponding input files or the XProc script.	No

Attribute	Description	Required
outDir	use in case of multiple input (via fileset). specifies the directory where the output files are to write to.	No.
outPattern	use in case of multiple input (via fileset). specifies how to build the output filename.	No.
noOutputfile	suppresses generation of an outputfile (overrides all other output parameters). The usage of this parameter makes sense when output is created within the pipeline.	No.
propertyfile	the properties this file contains will be passed to yax for further usage.	No.
baseURI	the base directory for all containing steps and containers.	No.
verbose	yax logs verbosely. For getting these log messages it is required that ant is also started in verbose mode.	No.
quiet	yax only logs errors, warnings and very important informations.	No.

Parameters specified as nested elements

Examples

see section 'Examples'

SOAP Interface

ToDo

The Pipeline Configuration File

The *<pipeline filename>*.config.xml contains the pipeline with all implicit ports and resulting connections between the ports and all used pipeline-library (the actual implementation is handled as serveral libraries):

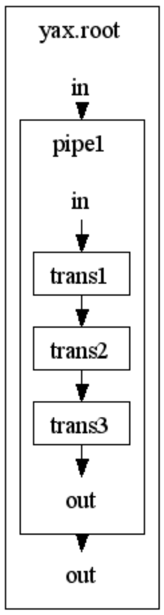
```
<yax:configuration xmlns:yax="http://www.opsdesign.eu/yax/1.0">
<!--configuration description generated by Yax - Do not edit by hand-->
<pipeline name="pipe1">
..
<step name="trans1">
<output port="out" sequence="no" yax:creator="implementation.xproc.standard"/>
<input port="in" sequence="no" yax:creator="implementation.xproc.standard">
```

```
<yax:connection port="in" yax:component="pipe1"/>
</input>
<input port="stylesheet" sequence="no" yax:creator="implementation.xproc.standard"/>
<input href="examples/transformation1.xsl" port="stylesheet" yax:creator="pipeline"/>
</step>
...
</pipeline>
<xproc:pipeline-library name="xproc.options" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
<xproc:declare-step-type type="Rename" yax:description="">
<xproc:output port="out" select="" sequence="no" yax:bySourceRequired="no"/>
<xproc:input port="in" select="" sequence="no" yax:bySourceRequired="no"/>
<xproc:parameter name="name" required="yes" yax:values="{$any}"/>
<xproc:parameter name="select" required="yes" yax:values="{$xpathExpression}"/>
</xproc:declare-step-type>
...
</xproc:pipeline-library>
<xproc:pipeline-library name="xproc.standard" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
...
</xproc:pipeline-library>
<xproc:pipeline-library name="yax.standard" xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0">
...
</xproc:pipeline-library>
</yax:configuration>
```

Graphical Visualization of a Pipeline

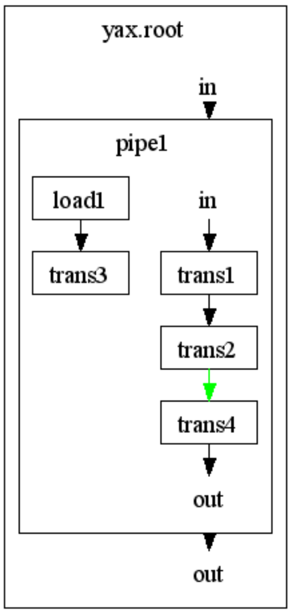
The file <*pipeline filename*>.pipeTrace.png contains the compressed graphical representation of the pipeline. (This file will only generated if the *automaticOutput* attribut of the <*pipetrace*> preference is set to 'yes'.)

Figure 9. Compressed Graphical Representation of the Pipeline 'example1.xproc'



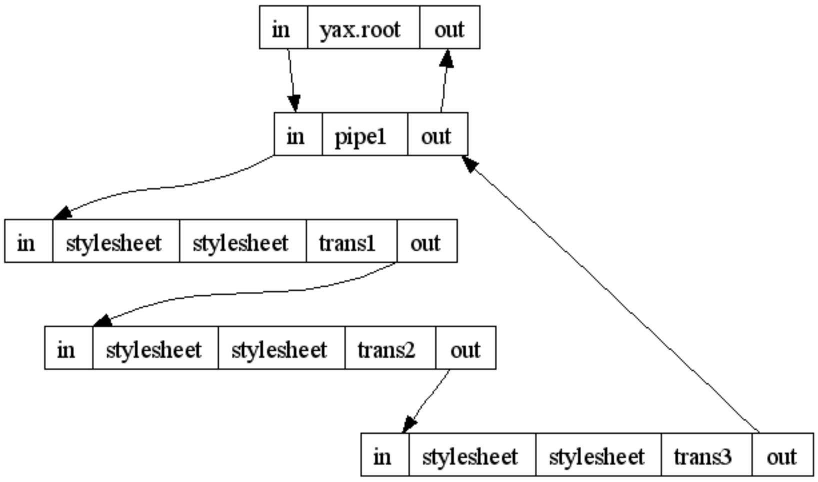
By default implicit connections apears in black or in blue color, explicit connections appears in green color. You can change the appearance according your needs by setting the appropriate preferences. (By change colors keep in mind that default color for errors is red)

Figure 10. Pipeline with implicit and explicit connections



The file <*pipeline filename*>.portTrace.png contains the graphical representation of the pipeline which depicts the expected route the process will take over the ports. (This file will only generated if the *automaticOutput* attribut of the <*porttrace*> preference is set to 'yes'.)

Figure 11. Compressed Graphical Representation of a Pipeline


in | yax.root | out
in | pipe1 | out
in | stylesheet | stylesheet | trans1 | out
in | stylesheet | stylesheet | trans2 | out
in | stylesheet | stylesheet | trans3 | out

Design and Setup a Pipeline

During this phase you can configure what an output you would like to generate when you apply yax to your pipeline by setting the appropriate running parameters.

Use *automaticOutput* for switch on/off the output of pipeTrace resp. portTrace.

Use *suppress* for switch on/off the processing of the pipeline.

run a Pipeline

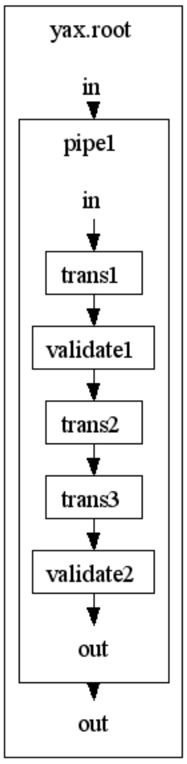
Analogous to the configuration possibilities depicts in *Design and Setup a Pipeline* you can switch off all outputs you don't need.

Using the Construct <*p:pipeline*>

The <*p:pipeline*> construct is a container for steps and other construct.

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:step name="validate1" type="xproc:Validate"/>
   <xproc:step name="trans2" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
   </xproc:step>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
   </xproc:step>
   <xproc:step name="validate2" type="xproc:Validate"/>
</xproc:pipeline>
```

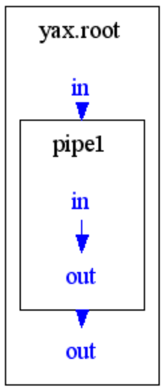
Figure 12. Pipeline



The most simple pipeline is an empty pipeline:

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
</xproc:pipeline>
```

Figure 13. Choose/When/Otherwise construct with exlicit ports at *choose* element



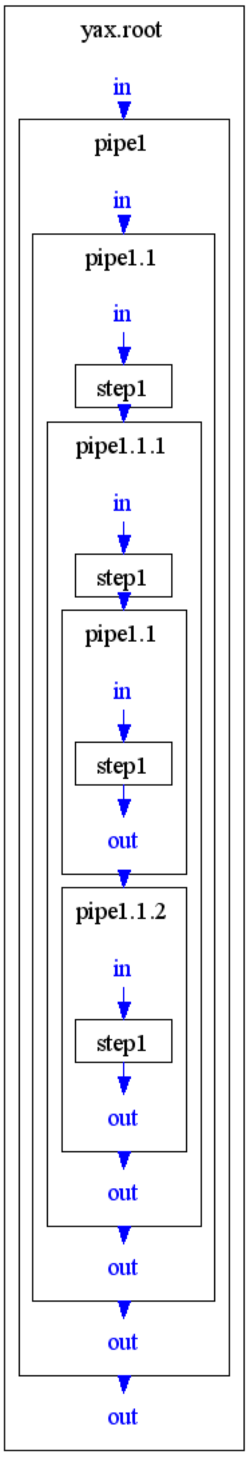
When you run such an empty pipeline you get a message that the pipeline will be bridged. In that case the pipeline construct has the same behaviour as the identity step.

```
...
Construct 'pipe1' is empty and will be bridged.
...
```

pipelines can contain nested components in an arbitrary (??) depth:

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:pipeline name="pipe1.1">
        <xproc:step name="step1" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="examples/transformation1.xsl"/>
        </xproc:step>
        <xproc:pipeline name="pipe1.1.1">
            <xproc:step name="step1" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation2.xsl"/>
            </xproc:step>
            <xproc:pipeline name="pipe1.1.1.1">
                <xproc:step name="step1" type="xproc:XSLT">
                    <xproc:input port="stylesheet" href="examples/transformation3.xsl"/>
                </xproc:step>
            </xproc:pipeline>
        </xproc:pipeline>
    </xproc:pipeline>
</xproc:pipeline>
```

Figure 14. Pipelline with nested Components



Using the Step *<p:XSLT>*

The usual form of the *xproc:XSLT* step consists of the *<xproc:step>* element itself which contains an *<input>* port element. This port refers to the *stylesheet* port.

```
<xproc:step name="trans1" type="xproc:XSLT">
   <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
</xproc:step>
```

This step uses the default transformer determined by the mechanism described below in this chapter. If you would like to use a particular transformer for a step, you can place a parameters:

```
<xproc:step name="trans1" type="xproc:XSLT">
  <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
               <xproc:parameter name="transformer" value="Saxon8"/>
</xproc:step>
```

Currently only the *href* reference to the source is supported.

Table 7. Exceptions of step <*XSLT*>

Exception	Annotation
trapped during processing	
EmptyResultException	
FileNotFoundException	If href is used to reach file and this file can not be found.
TransformerConfiguration	
Transformer	
DOM	
trapped during transformer determination	
TransformerConfiguration	
Transformer	
DOM	

Determine the desired Transformer

The step receives the document, which is to treat via the input port and provides the result of the transformation at the output port.

It determins the processor according the following mechanism:

Determination of the appropriate transformation processor

There are three ways to determine the processor. The ways overrides the effects of their ancestors in the sequence of its appearance. (The numbers 1) and 2) are part of the processor selection mechanism of JAXP. For further information concerning this mechanism please refer to Suns JAXP documentation. The numbers 3 to 5) all uses the system property mechanism).

- 1) Do nothing

  Since Version 1.4 Java contains a default transformation processor (XSLTC). This processor will be used if no other processors are visible (in the classpath).

- 2) Set the desired processor in the classpath

  JAXP uses the first first processor (precise: TransformerFactory) found in the classpath.

- 3) Set the desired processor with the system property 'javax.xml.transform.TransformerFactory' JAXP uses the transformer implementation mentioned in this system property.

- 4) Set a default transformer within the yax preferences

  Within the yax preferences file <yax-installation-directory>/config/preferences.xml you will find an element called <transformers>. It contains a (customizable) list of elements for each tranformer which shall be reachable by yax. You can insert a 'default' attribute into the <transformers> element with the name of the transformer which should become default processor. The value has to point to an existant <transformer> element.

- 5) Set the tranformer individually for a step

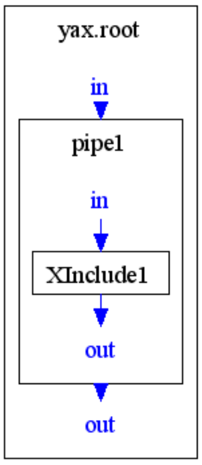
  You can choose a particual transformer for each step by using the parameter: <xproc:parameter name="transformer" value="saxon8" /> The value of this parameter has to point to an existant <transformer> element in the yax preferences file.

Using the Step *<p:XInclude>*

XInclude resolves the *<xi:include>* entry of the treated document. You simply post the *<p:XInclude>* step somewhere in the processing stream and *<xi:include>* entries contained in content will be replaced by the staff the *<xi:include>* entry is referring to.

```
...
<xproc:step name="XInclude1" type="xproc:XInclude"/>
...
```

Figure 15. ***<p:XInclude>***

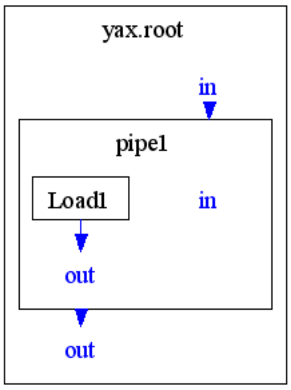


Using the Step *<p:Load>*

<p:Load> loads an XML Document from an external data source. *<p:Load>* references the source via an URI which is placed either in an *href* attribute or (as requested from the specification) parameter with the name *href*.

```
...
<xproc:step name="Load1" type="xproc:Load" href="examples/example1.xml"/>
...
```

Figure 16. ***<p:Load>***



In comparision to other steps at *<p:Load>* the *input* port is replaced by an external content source. The consequence is a break of the pipeline chain at this point. That means that the content at the *result* port of the preceding component of of the *<p:Load>* step will be discarded except other components refer to it explicitly.

<p:Load> can be used to replace the initial content load via the input file start paramter. So with *<p:Load>* you can place the reference to input data into the pipeline and can access yax with a single reference to location of the pipeline document.

```
# java -classpath lib/yax-0.7.jar net.sf.yax.Yax [options]pipeline file location
```

you can use parameter references with the form *${<parameter name>}* within *href* attribute (or parameter). This reference will be replaced by the value of parameter with the same name passed either by property file or by start parameter ('-D<parameter name>').

With a combination of start parameter and parameter references within the xproc script you can build your scripts in a more flexible way. Example:

```
# java -classpath lib/yax-0.7.jar
       net.sf.yax.Yax
       -Dexample.dir=examples
       -Dtest.dir=test
       -DoutputFilename=output1.xml
       example1.xproc
```

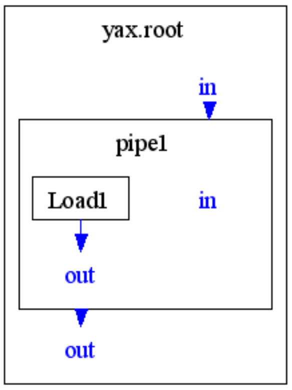
```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:step name="Load1" type="xproc:Load" href="${examples.dir}/${inputFilename}"/>
    <xproc:step name="Store1" type="xproc:Store" href="${test.dir}/${outputFilename}"/>
</xproc:pipeline>
```

Using the Step *<p:Store>*

<p:Store> stores an XML Document at external data target location. *<p:Store>* references the target location via an URI which is placed either in an *href* attribute or (as requested from the specification) parameter with the name *href*.

```
...
<xproc:step name="Store1" type="xproc:Store" href="examples/output.xml"/>
...
```

Figure 17. ***<p:Store>***



In comparision to other steps at *<p:Load>* the *input* port is replaced by an external content source. The consequence is a break of the pipeline chain at this point. That means that the content at the *result* port of the preceding component of of the *<p:Load>* step will be discarded except other components refer to it explicitly.

<p:Load> can be used to replace the initial content load via the input file start paramter. So with *<p:Load>* you can place the reference to input data into the pipeline and can access yax with a single reference to location of the pipeline document.

```
# java -classpath lib/yax-0.7.jar net.sf.yax.Yax [options]pipeline file location
```

you can use parameter references with the form *${<parameter name>}* within *href* attribute (or parameter). This reference will be replaced by the value of parameter with the same name passed either by property file or by start parameter ('-D<parameter name>').

With a combination of start parameter and parameter references within the xproc script you can build your scripts in a more flexible way. Example:

```
# java -classpath lib/yax-0.7.jar
       net.sf.yax.Yax
       -Dexample.dir=examples
       -Dtest.dir=test
       -DoutputFilename=output1.xml
       example1.xproc
```

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:step name="Load1" type="xproc:Load" href="${examples.dir}/${inputFilename}"/>
    <xproc:step name="Store1" type="xproc:Store" href="${test.dir}/${outputFilename}"/>
</xproc:pipeline>
```

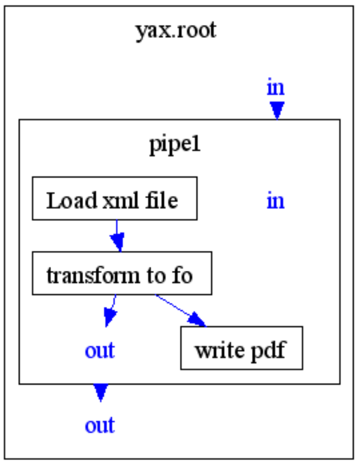
Using the Step *<p:Fop>* for creating PDF files

For using FOP you need FOP is installed on your system and the FOP libraries are part of your classpath.

At the moment *<p:Fop>* and *<p:PDF>* can be used either without any difference. *<p:PDF>* is previewed for future time when other fo processors are accessable by yax. In the meantime *<p:Fop>* and *<p:PDF>* are used as synomyms.

```
...
<xproc:step name="write pdf" type="xproc:Fop"/>
...
```

Figure 18. *<p:XInclude>*



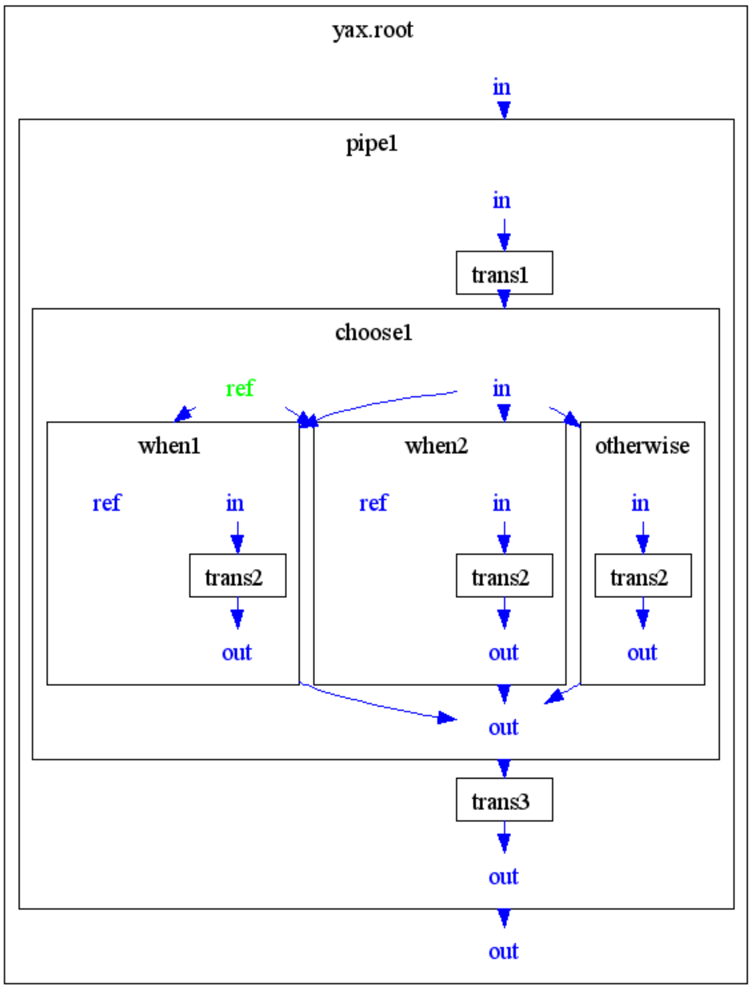
As you can see in the graphic the fop step requires XML-FO as input.

Using the Construct *<p:choose>/<p:when>/<p:otherwise>*

The usual case using a *choose/when/otherwise* construct will be that you define only one reference source (at the *choose* element) and applies the test clause of all <*when*> elements on it:

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:choose name="choose1" type="xproc:XSLT">
      <xproc:input port="ref" href="test/chooseInput0.xml"/>
      <xproc:when name="when1" test="/test1">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:when name="when2" test="/test2">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:otherwise name="otherwise">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:otherwise>
   </xproc:choose>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
   </xproc:step>
</xproc:pipeline>
```

Figure 19. Choose/When/Otherwise construct with exlicit ports at *choose* element

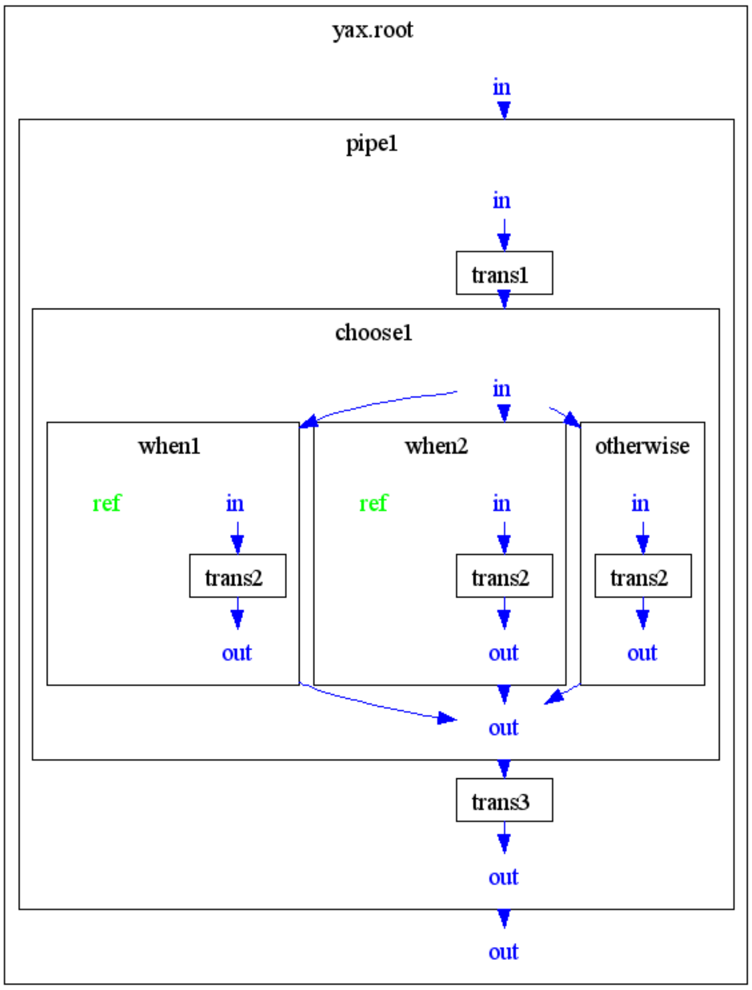


Also it is possible to define a reference source for each *when* clause:

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:choose name="choose1" type="xproc:XSLT">
      <xproc:when name="when1" test="/test1">
         <xproc:input port="ref" href="test/chooseInput1.xml"/>
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:when name="when2" test="/test2">
         <xproc:input port="ref" href="test/chooseInput2.xml"/>
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
```

```
         <xproc:otherwise name="otherwise">
            <xproc:step name="trans2" type="xproc:XSLT">
               <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
            </xproc:step>
         </xproc:otherwise>
      </xproc:choose>
      <xproc:step name="trans3" type="xproc:XSLT">
         <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
      </xproc:step>
   </xproc:pipeline>
```

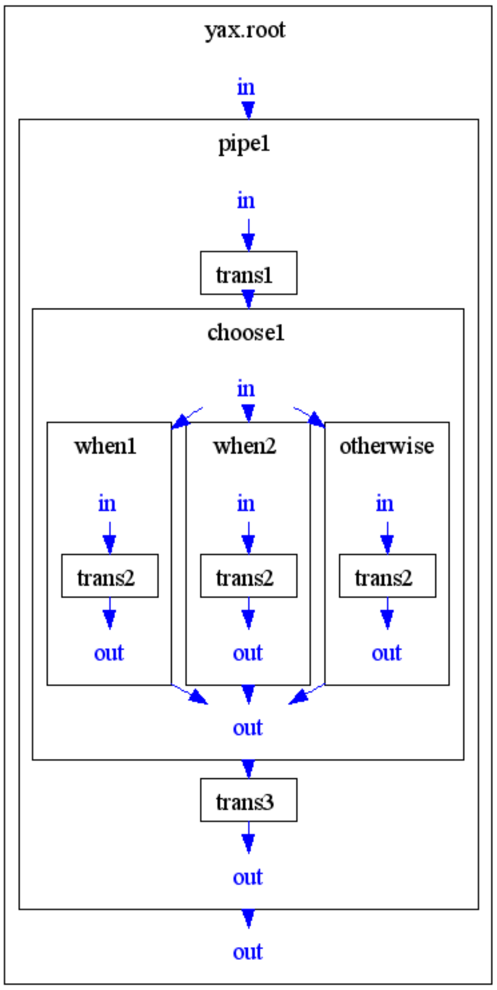
Figure 20. Choose/When/Otherwise construct with exlicit ports at all *when* elements



If you do not define at least one reference port:

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
   </xproc:step>
   <xproc:choose name="choose1" type="xproc:XSLT">
      <xproc:input port="ref" href="test/chooseInput0.xml"/>
      <xproc:when name="when1" test="/test1">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:when name="when2" test="/test2">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:otherwise name="otherwise">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:otherwise>
   </xproc:choose>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
   </xproc:step>
</xproc:pipeline>
```

Figure 21. Choose/When/Otherwise construct with exlicit ports at all elements



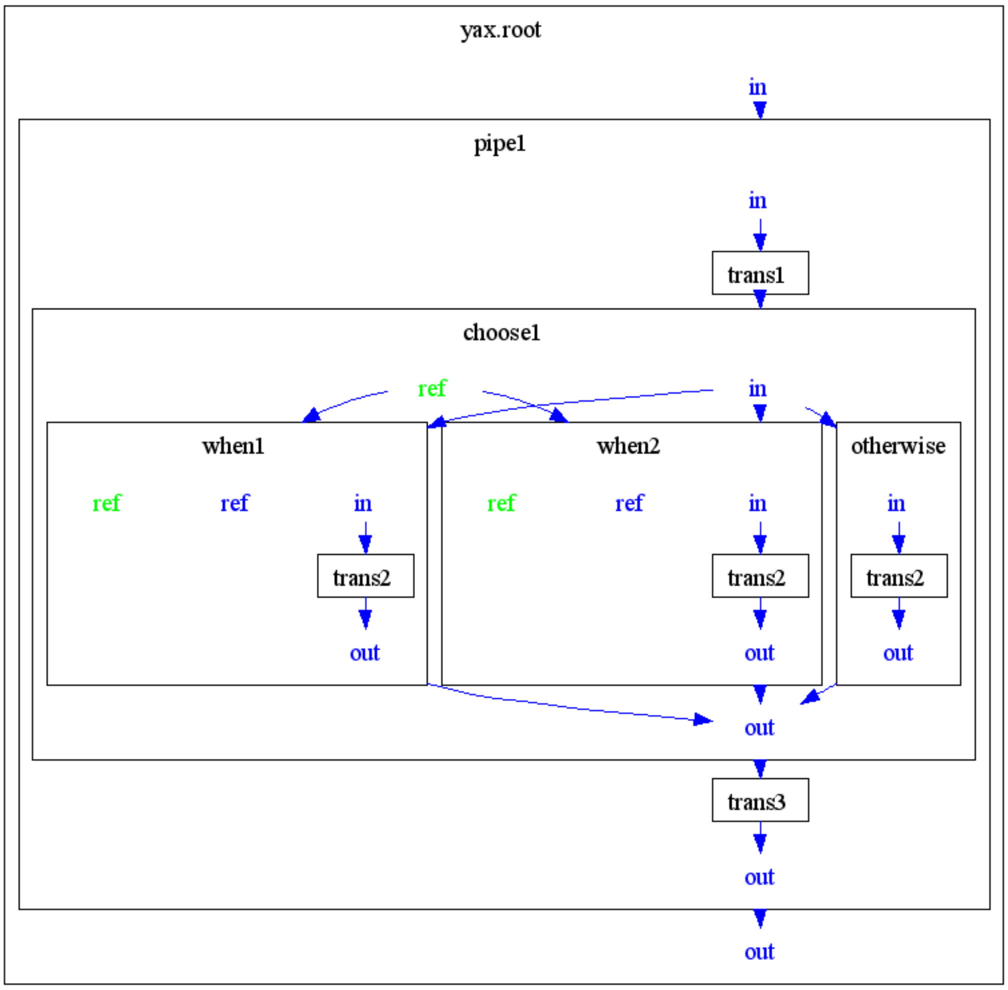
triggers an error message at the first *when* element:

```
construct 'when1': reference port not found.
```

You can also equip all elements of the *choose/when/otherwise* construct simultaniously with a reference port. In that case the reference port of the *choose* element triggers the generation of an implicit reference port at each *when* element. But the implicit reference ports take no effect because explicit ports always have priority. They regain effect when the explicit ports will be (also partitially) erased.

```
<xproc:pipeline name="pipe1"
   xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
   xmlns:yax="http://opsdesign.eu/yax/1.0">
   <xproc:step name="trans1" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation1.xsl"/>
   </xproc:step>
   <xproc:choose name="choose1" type="xproc:XSLT">
      <xproc:input port="ref" href="test/chooseInput0.xml"/>
      <xproc:when name="when1" test="/test1">
         <xproc:input port="ref" href="test/chooseInput1.xml"/>
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:when name="when2" test="/test2">
         <xproc:input port="ref" href="test/chooseInput2.xml"/>
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:when>
      <xproc:otherwise name="otherwise">
         <xproc:step name="trans2" type="xproc:XSLT">
            <xproc:input port="stylesheet" href="test/transformation2.xsl"/>
         </xproc:step>
      </xproc:otherwise>
   </xproc:choose>
   <xproc:step name="trans3" type="xproc:XSLT">
      <xproc:input port="stylesheet" href="test/transformation3.xsl"/>
   </xproc:step>
</xproc:pipeline>
```

Figure 22. Choose/When/Otherwise construct with exlicit ports at all elements

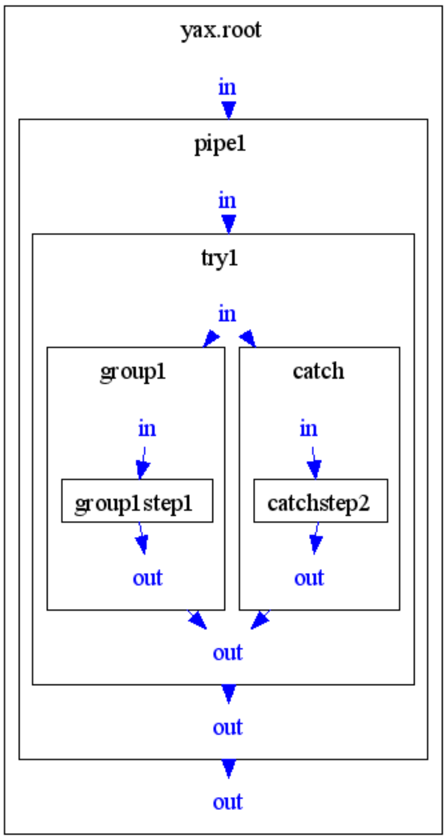


Using the Construct *<p:try>/<p:catch>*

You can use the *<p:try>/<p:catch>* construct either with only a general catch clause:

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:try name="try1">
        <xproc:group name="group1">
            <xproc:step name="group1step1" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation2.xsl"/>
            </xproc:step>
        </xproc:group>
        <xproc:catch name="catch">
            <xproc:step name="catchstep2" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation1.xsl"/>
            </xproc:step>
        </xproc:catch>
    </xproc:try>
</xproc:pipeline>
```

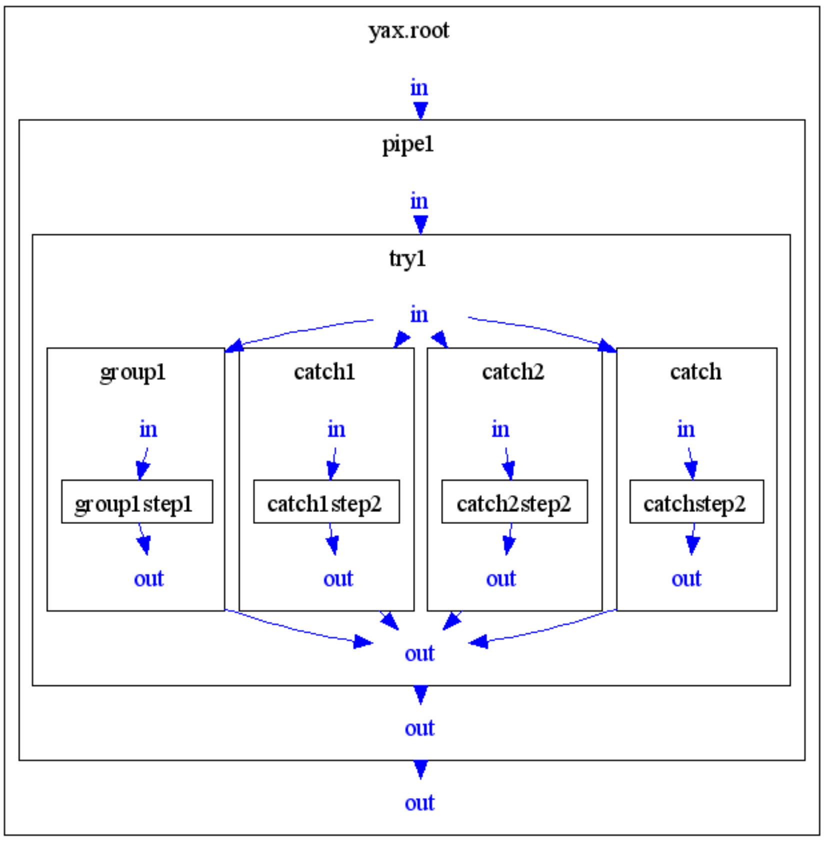
Figure 23. Try/Catch construct with a single catch clause*choose* element



or more sophicated with additional differenciating catch clauses:

```
<xproc:pipeline name="pipe1"
    xmlns:xproc="http://www.w3.org/TR/2006/xproc/1.0"
    xmlns:yax="http://opsdesign.eu/yax/1.0">
    <xproc:try name="try1">
        <xproc:group name="group1">
            <xproc:step name="group1step1" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation2.xsl"/>
            </xproc:step>
        </xproc:group>
        <xproc:catch name="catch1" exception="EmptyResult">
            <xproc:step name="catch1step2" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation1.xsl"/>
            </xproc:step>
        </xproc:catch>
        <xproc:catch name="catch2" exception="FileNotFound">
            <xproc:step name="catch2step2" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation1.xsl"/>
            </xproc:step>
        </xproc:catch>
        <xproc:catch name="catch">
            <xproc:step name="catchstep2" type="xproc:XSLT">
                <xproc:input port="stylesheet" href="examples/transformation1.xsl"/>
            </xproc:step>
        </xproc:catch>
    </xproc:try>
</xproc:pipeline>
```

Figure 24. Try/Catch construct with a single catch clause*choose* element



How ever, each *<p:try>*/*<p:catch>* construct has to contain at least the general catch clause.

Roadmap

Description ToDo

Examples

Overview

In the following you find several examples which depicts what features yax provides and how to achieve to make them run. For your confinience the start of all examples are assembled in the ant script 'runExamplesViaAnt.xml'. You find this file in the root of YAX_HOME. If you prefer to run yax from the command line you can find some 'runExample*.bat' files also at root of YAX_HOME.

Example 1.1 - Concatenated Transformations

Demonstrates a usual xml pipeline consisting of some concatenated transformation steps.

Example 1.2 - Concatenated Transformations (using an xml catalog)

Demonstrates alike Example 1.1. a usual xml pipeline consisting of some concatenated transformation. Supplementally resolves all URIs against an xml catalog.

Example 2.1. - *<p:XInclude>* in Action (resolves Xinclude entries)

This is an example with real data coming from the docbook.sml project [http://docbooksml/sourceforge.net/]. It collects all sections distributed over several files into the output file. Start yax with:

```
# java -classpath lib/yax-0.7.jar
       net.sf.yax.Yax
       -Dparam1=passedFromProgramStart
       example7.xproc
       examples/xincludeArticle.xml
       test/output1.xml
```

Example 2.1. - *<p:XInclude>* in Action (resolves Xinclude entries using an xml catalog))

Demonstrates alike Example 2.1. the usage of *<p:XInclude>*. Supplementally resolves all URIs against an xml catalog.

Example 3.1 - *<p:try>/<p:group>/<p:catch>* in Action ('choose' finds and processes a valid when clause)

'choose' finds and processes a valid when clause. (when clauses overrides with own ref ports the super ref port of choose.)

Example 3.2 - *<p:try>/<p:group>/<p:catch>* in Action ('choose' processes otherwise clause due to no when clause is valid)

'choose' processes otherwise clause due to no when clause is valid. (when clauses have no own ref ports, so uses the super ref port of choose.)

Example 4.1 - *<p:Load>/<p:XInclude>/<p:Store>* in Action (processing a general catch clause)

'try' detectes an exception and processes the (general) catch clause

Example 4.2 - *<p:Load>/<p:XInclude>/<p:Store>* in Action (processong a particular catch clause)

'try' detectes an exception and processes a particular catch clause

Example 5.1 - Using Inner and Outer I/O together

ToDo

Example 5.2 - Using only Inner I/O

ToDo

Example 6 - Passing (substituted) Transformation Parameter to *<p:XSLT>*

This examples shows how to

- pass a parameter from the program start through the xproc script to the xsl script (param1)
- pass a parameter from xproc script to the xsl script (param2)
- use parameters default value because nothing passed to (param3)

Start yax with:

```
# java -classpath lib/yax-0.7.jar
       net.sf.yax.Yax
       -Dparam1=passedFromProgramStart
       example6.xproc
```

```
        examples/example1.xml
        test/output1.xml
```

The output file test/output1.xml will consist of:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
    <param1>passedFromProgramStart</param1>
    <param2>${param2}</param2>
    <param3>notPassedToScript</param3>
</result>
```

Consider that param1 comes from program start, param2 comes (staticly) from xproc script and param3 is the default value from the xsl script.

Example 7 - processing multiple input documents

applies an xproc script on multiple input documents

Example 8 - *<p:PDF>/<p:FOP>* in Action (creating a PDF file)

creates a PDF-File from an XML-FO file.

Tutorial

A simple Pipeline

ToDo

A Pipeline with additional Output

ToDo

A Pipeline with additional Input

ToDo

A Pipeline with Choose/When/Otherwise and Try/Catch

ToDo

Links

Description ToDo

Legal Notice

Software and documentation is released under the terms of the GNU LGPL license (see http://www.gnu.org/copyleft/lesser.html) and comes without a warranty of any kind.

Copyright © 2006 - 2008 joerg.moebius@opsdesign.de [mailto:joerg.moebius@opsdesign.de]

Powered by

[http://sourceforge.net]

[http://docbook.sourceforge.net/]

FOP [http://xmlgraphics.apache.org/fop//]

SAXONICA XSLT AND XQUERY PROCESSING [http://www.saxonica.com/index.html]