



AGETOR®

AXT Basic V. 2.0.3

What's new?

Content

1	Introduction.....	3
2	Java and XSL	3
3	Setting AXT keys from XSL	4
4	Inlining XSL in AXTConfiguration.....	5
5	Passing parameters to XSL scripts.....	5
6	Overriding default input and output for filters.....	6
6.1	Filter input.....	6
6.1.1	Inlined input.....	6
6.1.2	URL input.....	7
6.1.3	Filter default input	7
6.2	Filter output.....	8
6.3	Input/Output summary.....	8
7	Calling local document entries.....	9
7.1	Coping with recursion.....	9
8	Enhanced error trace information.....	11

This document describes the changes made in AXT version 2.0.3.

1 Introduction

The primary goal of this release has been to support the creation and use of dynamically created information in the transformation process. With the new capabilities you may create information at one stage in the transformation process (e.g. initially summarize data or calculate/select information) and use it in later transformations steps (filters and side transformations).

Rather than inventing a new dedicated AXT language for data manipulation, XSL scripting in combination with Java method calls (see below) is used for creation of information. This information may be assigned to AXT *keys* that have global scope in the transformation process. AXT keys have previously been fixed with the invocation of a transformation and provided solely by the client submitting document data (and to some extent AXT itself). The choice of this model allows us to use the well-known filter abstraction for metadata (keys) manipulation while exploiting the full strength of XSL and Java for the actual manipulation.

AXT transformation execution has hitherto used parallel execution of filters. Since this model fits badly with generation of information at one stage and later use (serial/sequential notion), the default execution mode has been changed to sequential execution of filters in version 2.0.3. Sequential execution has the additional advantage of being less resource consuming (processor and memory-wise) and may result in better performance and response time in some cases.

Since key manipulation requires Java methods to be called from within an XSL-script we shall first look at how it is possible to invoke Java methods from XSL (in the Apache XSLT implementation used by AXT). Then it is shown how keys are actually set using a Java method. Next the new features of inlining XSL code and parameter passing to XSL scripts is described. Finally we look at the new strong feature of input/output stream redirection.

2 Java and XSL

The following example shows how the current date is selected and formatted using Java from within XSL. To use Java classes in XSL you must declare the types as namespaces in the stylesheet tag (line 3-4). All references to classes and instances of classes must be prefixed with the chosen namespace. To create instances of a declared class you use the `new()` operator (6-8) and assign the instance to an XSL variable using the `xsl:variable` expression syntax (6-8). Calling Java methods is done in a different way than in Java. You call a method in a `select` attribute (`variable` or `value-of` constructs) by prefixing the namespace for the class to the method and handing an instance of the class as the first parameter to the call (before other parameters). Line 11 (and 14) shows this. We wish to call the method `format` on the `SimpleDateFormat` instance that was created in line 7 (`myDateFormatter`). To do this we refer to the method (`df:format`) and pass the instance held in the XSL-variable `myDateFormatter` plus additional parameters (the date that should be formatted which is held in the variable `myDate`).

```
1. <xsl:stylesheet version="1.0"
2.   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3.   xmlns:da="xalan://java.util.Date"
4.   xmlns:df="xalan://java.text.SimpleDateFormat">
5. <xsl:template match="/">
6. <xsl:variable name="myDate" select="da:new()"/>
7. <xsl:variable name="myDateFormatter" select="df:new('yyyy-MM-dd')"/>
8. <xsl:variable name="myTimeFormatter" select="df:new('HH:mm:ss')"/>
9. <javatest>
10. <date>
11.   <xsl:value-of select="df:format($myDateFormatter, $myDate)"/>
12. </date>
13. <time>
14.   <xsl:value-of select="df:format($myTimeFormatter, $myDate)"/>
15. </time>
16. </javatest>
17. </xsl:template>
18. </xsl:stylesheet>
```

The output of the above script will look something like:

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <javatest xmlns:df="java.text.SimpleDateFormat"
3.   xmlns:da="xalan://java.util.Date">
4.   <date>2004-01-07</date>
5.     <time>14:01:17</time>
6. </javatest>
```

With this basic knowledge on Java method invocation from XSL we can now set AXT keys from XSL.

3 Setting AXT keys from XSL

The AXT XSLProcess filter automatically provides the set of AXT keys as a parameter to you XSL script. The AXT parameters are held in a subclass of the Java Properties class. Keys are held as name/value pairs and may be referred to by their name. The example below shows how to create an AXT key by the name `mydate` containing the current date formatted as in the previous example.

In line 4 we declare the AXT parameter class since we are going to use the provided instance (to add a key). In line 6 we must declare the name of the parameter instance, which is always `params` for the AXT parameter set. In line 12 the method `put()` on the key set is used for adding a new key `mydate` with the value `$adate`. `adate` is an XSL variable created in line 10 as the result of a Java method invocation returning a date formatted in a string.

Upon execution of this script the new key `mydate` is available in all subsequent AXT filters and side transformations.

```

1. <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
2.   xmlns:da="xalan://java.util.Date"
3.   xmlns:df="java.text.SimpleDateFormat"
4.   xmlns:Params="xalan://dk.bording.axt.core.conf.AXTParameters">
5.   <!-- params is AXT provided instance of AXTParameters -->
6.   <xsl:param name="params"/>
7.   <xsl:template match="/">
8.     <xsl:variable name="myDate" select="da:new()" />
9.     <xsl:variable name="myDateFormatter" select="df:new('yyyy-MM-dd')"/>
10.    <xsl:variable name="adate" select="df:format($myDateFormatter, $myDate)"/>
11.    <!-- add new key to AXTParameters using the put() method -->
12.    <xsl:value-of select="Params:put($params, 'mydate', $adate)"/>
13.  </xsl:template>
14. </xsl:stylesheet>

```

4 Inlining XSL in AXTConfiguration

Since AXT key manipulation is done in XSL but created keys usually are used/referred in the AXT configuration, hiding the key assignments could make the reading of the transformation logic more complicated. To address this problem we provide the optional feature that XSL code for the XSLProcess filter may be directly inlined in the AXT configuration file. In general, however, we do not recommend inlining XSL unless it adds to the understanding of the transformation process since the configuration becomes bigger and harder to grasp.

Since XSL instructions may conflict with the XML syntax of the configuration, inlined code must be escaped using the XML CDATA construct. This construct allows any characters to appear in an XML file without the XML processor parsing it as XML, thus avoiding any syntactical clash. Below the previous example (setting a key) is shown, as it may appear in the AXT configuration using inlining. Use the tag `transformation-instructions` around the XSL code (line 4 and 21) and the CDATA construct to avoid syntactical conflicts (line 5 and 20).

```

1. <doc name="setkey">
2.   <key name="function" value="setkey" />
3.   <filter class="dk.bording.axt.tc.xsl.XSLProcess">
4.     <transformation-instructions>
5.       <![CDATA[
6.         <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
7.           xmlns:da="xalan://java.util.Date"
8.           xmlns:df="java.text.SimpleDateFormat"
9.           xmlns:Params="xalan://dk.bording.axt.core.conf.AXTParameters">
10.          <!-- params is AXT provided instance of AXTParameters -->
11.          <xsl:param name="params"/>
12.          <xsl:template match="/">
13.            <xsl:variable name="myDate" select="da:new()" />
14.            <xsl:variable name="myDateFormatter" select="df:new('yyyy-MM-dd')"/>
15.            <xsl:variable name="adate" select="df:format($myDateFormatter, $myDate)"/>
16.            <!-- add new key to AXTParameters using the put() method -->
17.            <xsl:value-of select="Params:put($params, 'mydate', $adate)"/>
18.          </xsl:template>
19.        </xsl:stylesheet>
20.      ]]>
21.     </transformation-instructions>
22.   </filter>
23. </doc>

```

5 Passing parameters to XSL scripts

In this version of AXT we provide passing of parameters (and thus keys) to XSL scripts. To distinguish between parameters to the XSL process filter itself (e.g. the `xsl-script` parameter) and parameters that should be available *within* the script you must prefix the parameter names with the

keyword `xsl-param`. The example below illustrates how a key `somekey` (e.g. received from an inlet sending a document to AXT) is passed as a key to the XSL script using the `xsl-param` prefix (line 4). In the XSL script itself the key must be declared. This happens in line 8. The parameter becomes an ordinary XSL variable and may now be referred (line 10). Thus the example below generates an XML document containing the value of a key received from an external source. Of course this key value could have been manipulated or used as a parameter to a Java call etc.

```
1. <doc name="xslparam">
2. <key name="function" value="xslparam" />
3. <filter class="dk.bording.axt.tc.xsl.XSLProcess">
4.   <param name="xsl-param:myparam" value="{somekey}"/>
5. <transformation-instructions>
6.   <![CDATA[
7.     <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
8.       <xsl:param name="myparam"/>
9.       <xsl:template match="/">
10.        <result>parameter had value: <xsl:value-of select="$myparam"/></result>
11.      </xsl:template>
12.    </xsl:stylesheet>
13.  ]]>
14. </transformation-instructions>
15. </filter>
16. </doc>
```

6 Overriding default input and output for filters

Though we have chosen the filter abstraction for setting AXT keys, the data manipulation may not be a direct part of the document transformation at all. I.e. creating key variables for later use should normally not interfere with the document transformation and could be seen as *metadata* manipulation rather than document transformation. This means that we may not be interested in producing any output from the XSLProcess that creates keys but rather have the input stream copied unmodified to the next filter following the key setting filter. Similarly the input stream at the position where metadata are manipulated may not even be XML and thus not suitable as input for the XSLProcess filter! These observations have led to a new feature in AXT allowing filter input to be copied through a filter unmodified as well as having a filter create its own input on the fly.

6.1 Filter input

The default input to a filter is the output of the previous filter in the configuration sequence. This may be overridden in a number of ways from AXT version 2.0.3.

6.1.1 Inlined input

Input can be specified directly in the AXT configuration using the new `<input>` tag in the filter tag. The example below illustrates this. The input tag contains a (very simple) XML document, which is given as input to the XSL processor rather than the input to the document entry. A CDATA block is used since the `<?xml` prolog would otherwise result in a conflict for the configuration file which is also XML. Had the input been plain text to e.g. a mail filter, the CDATA block could have been omitted. No output is generated in this simple example.

```
1. <doc name="inlineinput">
2.   <key name="function" value="inlineinput" />
3.   <filter class="dk.bording.axt.tc.xsl.XSLProcess">
4.     <input><![CDATA[<?xml version="1.0" encoding="UTF-8"?><root/>]]></input>
5.     <transformation-instructions>
6.       <![CDATA[
7.         <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
8.           <xsl:template match="/">
9.             </xsl:template>
10.          </xsl:stylesheet>
11.        ]]>
12.      </transformation-instructions>
13.    </filter>
14. </doc>
```

6.1.2 URL input

A more useful source of input may be the data at an URL endpoint. To use this feature the filter attribute `input` should be used (rather than the inline tag above). The example below illustrates how a filter may fetch its input from the Internet on invocation. The fetched input in this example is a batch of orders. Usages could include invocation of a counter or calling upon an Internet information service.

```
1. <doc name="urlinput" execmode="ser">
2.   <key name="function" value="urlinput" />
3.   <filter class="dk.bording.axt.tc.xsl.XSLProcess"
4.     <input="https://tools.rushorder.com/xml/xml.spec.xml">
5.     <transformation-instructions>
6.       <![CDATA[
7.         <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
8.           <xsl:template match="/">
9.             <xsl:value-of select="orderdata/@batch"/>
10.          </xsl:template>
11.        </xsl:stylesheet>
12.      ]]>
13.    </transformation-instructions>
14.  </filter>
15. </doc>
```

6.1.3 Filter default input

When using the XSLProcess filter for setting keys we need input to be of XML format in order for the XSL processor to work. However we may not care about the input at all. To handle this situation you may let the value of the input attribute be `'default'`. The AXT will ask the filter in question to provide its own suitable default input adhering to whatever format the filter expects. Default for the XSLProcess filter is the very simple XML document below:

```
<?xml version='1.0' encoding='UTF-8'?><default-xml/>
```

Below the code for setting an AXT key is shown again now with the input attribute set to default allowing *any* output from the previous filter (since it is ignored).

```

1. <doc name="setkey">
2. <key name="function" value="setkey" />
3. <filter class="dk.bording.axt.tc.xml.XSLProcess" input="default">
4. <transformation-instructions>
5. <![CDATA[
6. <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
7. xmlns:da="xalan://java.util.Date"
8. xmlns:df="java.text.SimpleDateFormat"
9. xmlns:Params="xalan://dk.bording.axt.core.conf.AXTParameters">
10. <!-- params is AXT provided instance of AXTParameters -->
11. <xsl:param name="params"/>
12. <xsl:template match="/">
13. <xsl:variable name="myDate" select="da:new()"/>
14. <xsl:variable name="myDateFormatter" select="df:new('yyyy-MM-dd')"/>
15. <xsl:variable name="adate" select="df:format($myDateFormatter, $myDate)"/>
16. <!-- add new key to AXTParameters using the put() method -->
17. <xsl:value-of select="Params:put($params,'mydate',$adate)"/>
18. </xsl:template>
19. </xsl:stylesheet>
20. ]]>
21. </transformation-instructions>
22. </filter>
23. </doc>

```

6.2 Filter output

The default output of a filter is the result of the filters data processing. However in some cases (as when setting keys) you may not wish to produce an output and thus interfere with the input stream. To this end AXT let you specify that output should be (i) the output from the previous filter or (ii) the input to the filter. You may wonder when the input to the filter is not the same as the output from the previous filter, but this was exactly the case when e.g. input is read from an URL or the filter default input is used as explained previously. Thus to use an XSLProcess filter for setting AXT keys without interfering with the stream you should use the following attribute values for the filter:

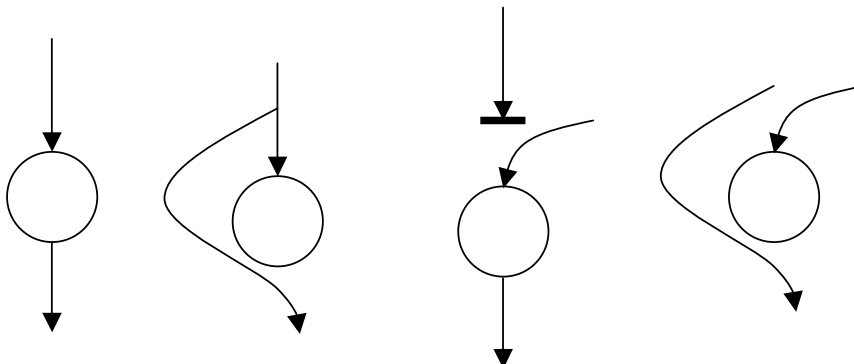
```
<filter class="dk.bording.axt.tc.xml.XSLProcess" input="default" output="prev-filter">
```

To output the *input* (e.g. the URL data in the example above) you should set the value of the output attribute to input:

```
<filter class="dk.bording.axt.tc.xml.XSLProcess" input="default" output="input">
```

6.3 Input/Output summary

Different combinations of the input/output declaration result in a number of options for the data stream routing:



(i) Default streaming: filter receives output from previous filter and outputs filter result

(ii) Sending output of previous filter to next filter

(iii) Obtaining input from external source (URL) or inlining it

(iv) Obtaining input from external source while forwarding output from previous filter

7 Calling local document entries

A new optimized method for calling local document entries is now provided through the filter `dk.bording.axt.flow.LocalCall`. Parameters to the filter are interpreted as keys selecting a local transformation (document entry). The input to the filter is routed to whatever document entry matching the provided parameters best. The output of the filter is the output resulting from executing the document entry. Note that this call abstraction accepts all input formats opposed to the `DocumentFlowFilter`. `LocalCall` does not support extraction of keys as the `DocumentFlowFilter`, but we recommend using the new call mechanism rather than the `DocumentFlowFilter` filter since it is much more efficient and robust to large data streams. To use information from the input stream as keys in document entry selection you should use the new key setting facilities.

The entry below illustrates how another entry may be called. The entry does some processing before invoking an order conversion entry matching the given `provider` key.

```
1. <doc name="call1">
2.   <key name="function" value="call1"/>
3.   <key name="provider" value="*/>
4.   . . .
5.   <filter class="dk.bording.axt.flow.LocalCall">
6.     <param name="function" value="orderconversion"/>
7.     <param name="src" value="{provider}"/>
8.   </filter>
9.   . . .
10. </doc>
```

7.1 Coping with recursion

Since the `LocalCall` filter may result in an invocation of the document entry in which the `LocalCall` is used, recursion is possible. This may or may not be intended and if not handled properly result in an eternal loop consuming all system resources thus effectively laying down the machine on which AXT is running. To avoid unintended recursion, recursion is by default intercepted by the AXT engine and will result in a runtime exception if attempted (see excerpt below).

```

1. <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2. <exception>
3.   <type>dk.bording.axt.flow.LocalCallException</type>
4.   <message>Local call aborted due to attempted recursion</message>
5.   . . .

```

If recursion is actually intended, you may override the default behavior by explicitly specifying that recursion is allowed. This is done per document entry with the boolean attribute `allow-recursion`. The example below illustrates this. A main recursion entry `recursiontest` accepts a count key with any value. The entry uses XSL to decrease the received by one, calling itself again with the new reduced value. Match is achieved until count becomes zero where a match with the preceding (more specialized) document entry occurs. Note that changing the order of the two entries would result in infinite recursion since the main entry would match zero before the specialized entry!

```

1. <!-- end of recursion when count is 0 -->
2. <doc name="recursiontest">
3.   <key name="function" value="recursiontest"/>
4.   <key name="count" value="0"/>
5.   <filter class="dk.bording.axt.core.FunctionFilter" output="input">
6.     <param name="func" value="copy"/>
7.     <input><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
8.       <text>recursion ended here!</text>
9.     ]]>
10.   </input>
11. </filter>
12. </doc>
13.
14. <!--Main recursion entry -->
15. <doc name="recursiontest" allow-recursion="true">
16.   <key name="function" value="recursiontest"/>
17.   <key name="count" value="*"/>
18.   <filter class="dk.bording.axt.core.FunctionFilter" output="input">
19.     <param name="func" value="log"/>
20.     <param name="text" value="recursiontest called with count={count}"/>
21.   </filter>
22.   <filter class="dk.bording.axt.tc.xsl.XSLProcess" input="default">
23.     <transformation-instructions>
24.       <![CDATA[
25.         <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
26.           xmlns:Params="xalan://dk.bording.axt.core.conf.AXTParameters">
27.           <!-- params is AXT provided instance of AXTParameters -->
28.           <xsl:param name="params"/>
29.           <xsl:template match="/">
30.             <xsl:variable name="count">
31.               <!--update key using the get()/put() methods -->
32.               <xsl:value-of select="Params:get($params,'count')-1"/>
33.             </xsl:variable>
34.             <xsl:value-of select="Params:put($params,'count', $count)"/>
35.           </xsl:template>
36.         </xsl:stylesheet>
37.       ]]>
38.     </transformation-instructions>
39.   </filter>
40.   <filter class="dk.bording.axt.flow.LocalCall">
41.     <param name="function" value="recursiontest"/>
42.     <param name="count" value="{count}"/>
43.   </filter>
44. </doc>

```

Note that recursion may be very resource consuming and should usually be avoided unless the recursion degree (number of recursive calls) is known to be quite limited.

8 Enhanced error trace information

With the new `LocalCall` filter the need for tracing deeply nested errors has been met by enhancing the error output from AXT. In case of an error, the complete trace of document entry invocations is returned with information on actual parameters.