Automating XML Pipelines Through Rules

Rui Lopes and Luís Carriço

LaSIGE and Department of Informatics
University of Lisbon
{rlopes,lmc}@di.fc.ul.pt

Abstract. This paper presents XPR - XML Pipeline Rules, a novel approach on defining, maintaining, and using XML pipelines for multiple-step document format conversion and automated detection. XPR introduces a template-based approach for reusable blocks of processing operations, as well as an abstraction for detecting document formats based either on XPath or schema validation. With XPR, there is no need to hand code different pipelines for each required document conversion. Therefore, it helps developers on maintenance and extensibility scenarios. A use case is presented, illustrating the drawbacks of current XML pipeline technologies, and how XPR can be used complementary to these by overcoming their limitations.

1 Introduction

XML processing with pipelines is an emerging topic in the XML scene. Pipelines have raised the level of abstraction on defining XML-based applications, discarding the complexity from using traditional API-based XML technologies in programming languages. Thus, it has shifted away from a procedural approach towards a purely declarative one. As such, XML pipelines are defined as a flow of operations to be performed over XML documents (e.g., applying an XSLT stylesheet, validating a document against a schema, etc.), according to some given business logic. Furthermore, pipelines improve on the maintenance and reuse of operations, as users typically tend to develop blackboxed solutions for performing required tasks (e.g., one XSLT stylesheet *vs.* several stylesheets plus something to glue them all).

Albeit being an emerging topic, different XML pipeline technologies have been developed in the last years. Such technologies tried to solve different problems, therefore ended on having different capabilities. Cocoon [1] and XPL [2] were defined with a flavour towards creating full-fledged XML-based web applications; MT pipeline [3] and smallx [4] shared a common goal of providing highly efficient XML processing for large datasets; SXPipe [5], XPDL [6], or even Ant scripts [7] provided simpler (therefore less powerful) ways for specifying XML pipelines. XProc [8] is the W3C's proposal on standardizing these technologies, to cope with their different needs. Despite the differences on purposes and specification languages, all these technologies share a common concept: pipelines are defined explicitly, i.e., a developer specifies the flow of operations to be performed over a given input, delivering an expected output. If the logic behind

this process has to be changed or adapted to different circumstances, the developer has to modify the pipeline accordingly (e.g., adding support for a different XML format at the pipeline's input).

However, the level of abstraction introduced with XML pipeline technologies may be insufficient when scaling or maintaining pipeline-based applications. Imagine the following scenario: a developer has to come up with a solution for transforming different document formats into a normalized publishing format (XSL-FO) to be further processed in consonance to a required business logic. Moreover, this process should be fully automated, to discard the manual selection of appropriate transformations based on document formats. Furthermore, after evaluating the problem, the developer realized that direct transformations to XSL-FO were not available to some document formats (such as one-to-one XSLT stylesheets). The cost of developing them is highly prohibitive, but sequences of off-the-shelf transformations were able to perform those tasks. Based on these requirements and available tecnhologies for document format conversion, the following graph of transformations was identified (Figure 1):

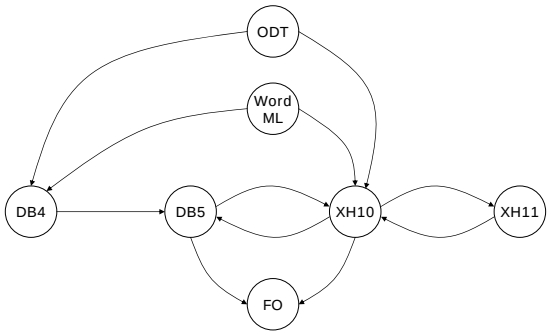

Fig. 1. Graph of transformations

Each node in the graph represents a particular document format that must be supported (Open Document for word processing, *ODT*; Microsoft Word 2003, *WordML*; DocBook v.4, *DB4*; DocBook v.5, *DB5*; XHTML 1.0 Strict, *XH10*; XHTML 1.1, *XH11*; XSL-FO, *FO*), while each arc represents a transformation from its source node to its destination node. In this scenario, any path between a chosen node and the *FO* node is an XML pipeline that can be constructed and used. However, the maintenance cost of creating this type of applications increases as new requirements appear (e.g., supporting a new document format, changing a document transformation operation to a more robust version, support different normalized document formats, etc.), therefore being reflected on pipelines specifications - either by modifying existing ones, fully rewritting them, or even creating new sets of pipelines.

Two concepts can be leveraged from these type of scenarios: identifying the transformations that are available (and how they can be concatenated), and de-

tecting automatically XML document formats. From these high-level concepts, any XML pipeline can be derived according to required transformation semantics. This paper presents *XPR - XML Pipeline Rules*, a high-level language for the specification of these two concepts, named *routing graphs* and *sensors* for document formats, respectively. These rules are defined and transformed into corresponding XML pipelines based on the syntax proposed by the latest XProc Working Draft. Next, the way routing graphs are specified is presented.

2 Defining a routing graph

A routing graph describes what operations have to be used to transform any XML input format into any XML output format. Therefore, each node of a routing graph is, at least, a source or a destination of one processing operation. This means that, for instance, from the scenario described previously, the routing graph is defined by the set of arcs of Figure 1. Mapping such abstractions into a concrete XML syntax is done through *XPR templates rules*, as seen on Figure 2 (a complete solution is presented in Appendix A).

```
<xpr:rules xmlns:xpr="http://www.rlopes.net/xml/xpr"
           xmlns:p="http://www.w3.org/2006/XProc">

  <xpr:template in="odt" out="xh10">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="odt-to-xh10.xsl"/>
    </p:step>
  </xpr:template>

  <xpr:template in="xh10" out="fo">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="xh10-to-fo.xsl"/>
    </p:step>
  </xpr:template>

</xpr:rules>
```

Fig. 2. XPR templates example

This example presents the basic language constructs usage for defining a routing graph, through the specification of a template that converts *ODT* into *XHTML 1.0*, and another one from *XHTML 1.0* to *XSL-FO*. At the top level, the `xpr:rules` element agreggates templates defined with `xpr:template` elements. Each template defines its input and output identifiers (e.g., document formats) with the `in` and `out` attributes, respectively.

Inside each template, an XProc-based syntax of a flow of operations has to be defined. Each template may be a lot more sofisticated as compared to the example, such as combining several XProc steps and constructs, calling other pipelines, etc. This is typically done when the intermediate results from the operations are not intended to be exposed on the routing graph. In order to link each template when computing XProc-based pipelines, three bindings have to be used inside the templates. These bindings are `$name` (the name of the flow of operations to be referenced inside pipelines), `$from` (the operation upon which the template is dependent from), and `$source` (the specific source to be grabbed from the operation). The presented templates will be used as operations on other pipelines that will be also generated: converting *WordML* and *XHTML 1.1* to *XSL-FO*, both will use the conversion between *XHTML 1.0* and *XSL-FO* defined in the second template. Even further, if a different conversion target is chosen for the same routing graph (i.e., replacing XSL-FO), both templates can be used seamlessly, therefore leveraging extensibility and maintenance scenarios.

After the establishment of the desired routing graph, the abstraction layer for the automatic detection of document formats is defined through a set of *sensors*, as presented in the next section.

3 Attaching sensors

As explained earlier, an XML pipeline is typically defined to support a specific document format as its input. Adding support for other document formats may be implemented through XPath-based testing with `choose/when/otherwise` constructs (similarly to XSLT's). More advanced scenarios may opt for schema validation as the way to detect document formats and trigger appropriate pipelines to perform desired operations, through `try/catch` mechanisms. Both approaches are similar in their semantics, as they perform some type of detection of document formats. However, as they are reflected differently on XML pipeline syntax, it poses increasing difficulties on extensibility and maintenance scenarios (i.e., typically results on deep trees of nested `choose` and `try` blocks). Moreover, when a different conversion target is required to be supported, these mechanisms have to be adapted accordingly in a manual fashion.

Consequently, XPR introduces the notion of *XPR sensor rules* for document formats to allow the automation of selecting an approriate XML conversion pipeline. Each sensor describes its semantics either through an XPath expression or a validation schema, and links itself to a specific document format. Figure 3 presents an example for the scenario described earlier (a complete solution is presented in Appendix A).

This example shows the syntax for describing sensors, both XPath-based and schema-based. The XPath expression tests loosely for an *XHTML 1.0* document, while a RELAX NG schema identifies an *ODT* document. In addition to `xpr:template` elements, the `xpr:rules` element also wraps `xpr:sensor` elements, the way sensors are described. Each sensor defines its associated document format through the `format` attribute. In the case of XPath-based sensors,

```
<xpr:rules xmlns:xpr="http://www.rlopes.net/xml/xpr"
           xmlns:html="http://www.w3.org/1999/xhtml">

  <xpr:sensor format="xh10" test="/html:html and not(//html:ruby)"/>

  <xpr:sensor format="odt" schema-href="odf.rng" schema-type="relaxng"/>

</xpr:rules>
```

Fig. 3. XPR sensors example

the `test` attribute must be used for defining the document format identification expression. On the other hand, when creating a schema-based sensor, the `schema-href` and `schema-type` have to be used to identify the schema location and its type (e.g., `xmlschema`, `relaxng`, etc.), respectively. In the case of having more than one sensor for the same document format, sensor ordering inside the XPR document decides which one to be applied.

After the rules for the routing graph and sensors have been described, they must be applied in the appropriate way, as explained in the next section.

4 Applying the rules

With the routing graph described, and the sensors defined accordingly, these rules are applied in the following way:

- A node in the routing graph is selected as the target document format;
- Each route between every node and this node is traced;
- For each route, a corresponding pipeline is created;
- The appropriate sensor is attached to each pipeline.

However, this algorithm may yeld more than one valid route between starting and ending nodes. In such cases, the shortest path method must be applied accordingly, as less transformations will be used (typically resulting on a better performance). If, after this, more than one route can still be selected, template ordering inside an XPR document dictates the decision (earlier templates will be chosen over latter ones).

Another feature on applying XPR rules bases itself on the lifespan of sensors. Their main goal is to automatically detect a document format and select an appropriate pipeline. However, they may be used also as validation mechanisms between the operations defined by templates, therefore enforcing that valid documents are fed to and produced by each processing operation.

The concrete application of XPR rules on existing or newly-created XProc-based pipelines can be done in different scenarios, depending on its goals and execution environment, as follows:

1. *Offline rules unwinding:* this type of XPR rules application is based on transforming a set of rules into a concrete XProc library consisting on the set of pipelines for each document format to be supported, and a pipeline that combines the specified sensors with their corresponding pipelines. This type of application is particularly suitable for command line or script-based generation of pipelines;
2. *Double pass pipeline execution:* in this case, an XProc step for triggering XPR is used inside XProc pipelines (like any other XProc step). However, the execution environment preprocesses the XPR step by unwinding it into concrete pipeline libraries (like in the previous case), and replacing the XPR step with a call to the sensors pipeline. This case enables the direct usage of XPR inside XProc pipelines without having to implement and/or adapt an XPR engine to an XProc processor;
3. *XProc component:* syntactically, this case is used exactly in the same way as the double pass pipeline execution case. However, by implementing a native component for XProc, there is no need for pre-processing a pipeline document. Therefore, pipelines can be executed directly. Another benefit from this approach comes from performance optimization issues, as the implementation may pre-compile and cache the generated pipelines.

As said, for the second and third cases, XPR is used directly inside XProc pipeline documents. This is done seamlessly, through the definition of the a special purpose processing component, according to the XProc step signature presented in Figure 4 (namespaces omitted for simplification):

```
<p:declare-step-type type="xpr:rules">
  <p:input port="document"/>
  <p:input port="rules"/>
  <p:output port="result"/>
  <p:parameter name="format" required="yes" />
  <p:parameter name="sensors-everywhere" />
</p:declare-step-type>
```

Fig. 4. XProc step signature for XPR

This step declaration for XPR requires two inputs (the `document` to be processed, and the `rules` document), produces one `result` document, and requires two parameters, the document `format` that will be chosen to be delivered by generated pipelines, and an optional `sensors-everywhere` flag to signal the insertion of sensors after every routing graph nodes (increasing their lifespan during pipeline execution).

From the scenario described earlier in this paper, a real XProc pipeline that uses XPR according to this step signature is presented in Figure 5.

Next section discusses briefly the way an XPR processor prototype was implemented.

```
<p:pipeline xmlns:p="http://www.w3.org/2006/XProc"
            xmlns:xpr="http://www.rlopes.net/xml/xpr"
            name="publisher">

  <p:input port="document"/>
  <p:output port="result" step="branding" source="result"/>

  <p:step name="normalize" type="xpr:rules">
    <p:input name="document" step="publisher" source="document"/>
    <p:input name="rules" href="rules.xml"/>
    <p:parameter name="format" value="fo"/>
  </p:step>

  <p:step name="branding" type="p:xslt">
    <p:input name="document" step="normalize" source="result"/>
    <p:input name="stylesheet" href="branding.xsl"/>
  </p:step>

</p:pipeline>
```

Fig. 5. XProc pipeline example using an XPR-based step

5 Implementation details

As a proof-of-concept, an XPR processor was implemented in XSLT 2.0 in consonance with the first and second scenarios described on the previous section. This implementation transformed both sensor and routing rules into concrete XProc pipelines, according to XPR semantics described earlier. Three main aspects can be distinguished on implementing XPR: detecting which routes are valid according to the selected target document format; generate the sensor pipeline; and generate each concrete route's pipeline.

The first step on implementing XPR in XSLT is to detect which routes are valid, i.e., which input document formats have a route to the target document format. Due to XSLT's functional nature, the prototype XPR processor implemented route detection through a naïve recursive breadth-first algorithm. First, an initial route set is chosen based on all input document formats (i.e., available templates's `in` and `out` attributes). Afterwards, each route's ending node is expanded to all available subsequent nodes in the routing graph. This operation is repeated for each route, until one of two conditions is verified: a loop has been detected, or no more expansion can be performed. After all valid routes have been found, the shortest ones are filtered for each available input document format, therefore forming the routing set that will be used subsequently.

Based on the routing set calculated previously, the sensor pipeline can now be created. Each sensor is transformed into specific XProc constructs, depending on its type (XPath or schema-based). For succeding sensor tests, a corresponding

transformation pipeline is called. Each sensor failure is recusively transformed into the next route's sensor detection.

More concretely, XPR sensor based on XPath expressions are transformed into `choose/when/otherwise` XProc constructs (see Figure 6 for an example): `when` the testing for the sensor's XPath expression succeeds, it means that a document has been fed that matches the rules specified by the sensor, therefore the appropriate pipeline is called; `otherwise`, iterate to the next sensor in the list.

```
<p:choose name="generated-sensor-name">

  <p:when test="sensor's XPath expression"/>
    <!-- call the corresponding pipeline -->
  </p:when>

  <p:otherwise>
    <!-- iterate to the next sensor -->
  </p:otherwise>

</p:choose>
```

Fig. 6. XProc skeleton for an XPR XPath-based sensor

In the case of schema-based sensors, their semantics are transformed into `try/catch` XProc constructs (see Figure 7 for an example): first, `try` to validate the input document according to the specified schema and execute the corresponding pipeline; if something goes wrong, `catch` the validation error and iterate to the next sensor in the list.

These XProc skeletons for XPR sensors further emphasize the (probable) inherent complexity of specifying directly a sensor pipeline in XProc, as the pipeline's XML tree deepness grows proportionally to the number of sensors that must be used. Once again, this issue may pose severe problems on scaling and maintaining this type of pipelines in a manual fashion, therefore enforcing the need for the abstraction layer provided by XPR.

The last step on transforming XPR into concrete XProc pipelines relates to the creation of the transformation pipelines themselves. As in the previous case, the routing set calculated initially serves as the ground basis for pipeline construction. Therefore, each route arc is transformed into the corresponding step defined inside XPR template rules, and all bindings (`$name`, `$from`, and `$source`) are adapted accordingly.

The result from all these transformations is an XProc library consisting of a sensor pipeline, and a set of pipelines based on the calculated routes. From the scenarios presented in the previous section, this prototype can be used standalone (unwinding mode) or used in conjunction with a translator of XProc-based XPR

```
<p:try name="generated-sensor-name">

  <p:group name="generated-group-name">
    <p:step name="generated-step-name" type="p:schema-type">
      <p:input port="document" ... />
      <p:input port="schema" href="schema-location" />
    </p:step>
    <!-- call the corresponding pipeline -->
  </p:group>

  <p:catch>
    <!-- iterate to the next sensor -->
  </p:catch>

</p:try>
```

Fig. 7. XProc skeleton for an XPR schema-based sensor

steps into concrete pipeline calls. Both cases are supported in the prototype through simple Ant scripts, thus out-of-the-box functionality is available.

6 Concluding remarks

This paper presented XPR - XML Pipeline Rules, a solution for simplifying the creation of XML pipelines for document format conversion based on partitioned flows of operations, and a way to ease the automation on document format detection based either on XPath expressions or full schema validation. By defining a set of template rules and a target document format, no pipelines have to be specified by hand, reducing human errors and maintenance costs on improving, modifying, and extending pipeline-based document transformations. Moreover, focusing solely on specifying rules for document format detection, developers do not have to tinker complex document detection blocks to be able to apply the same pipeline-based business logic to different document formats automatically.

As the XML pipeline syntax followed by XPR is XProc, some details on XPR's syntax and implementation may be modified in the future, side-by-side with the evolution of XProc until reaching *Technical Recommendation* status. Moreover, XPR will evolve also as a native XProc processing component to be used out-of-the-box in XProc implementations (according to the third scenario described on the rules application section).

Lastly, the current way XPR template rules are specified and used is based solely on matching in and out document formats. This idea can be further extended to more complex annotations (e.g., semantic-based) upon which sofisticated inference engines can select more appropriate routes and generate corresponding pipelines. Such annotations can be used to, for instance, identify priorities on choosing which route is more appropriate for transforming a given

document format to the desired target document format (e.g., based on the sofistication/accuracy of the transformations specified inside each XPR template), or even composing several annotations as a way to describe each template's acceptable XML data (i.e, using logic-based operators).

References

1. Apache Foundation: Apache Cocoon (1999-2006) http://cocoon.apache.org.
2. Bruchez, E., Vernet, A.: XML Pipeline Language (XPL) Version 1.0 (Draft). W3C Member Submission (2005) http://www.w3.org/Submission/xpl.
3. Markup Technologies: Mt pipeline (2006) http://www.markup.co.uk.
4. Milowski, A.: smallx (2006) https://smallx.dev.java.net.
5. Walsh, N.: SXPipe - Simple XML Pipelines (2004) https://sxpipe.dev.java.net/nonav/specs/sxpipe.html.
6. Walsh, N., Maler, E.: XML Pipeline Definition Language Version 1.0. W3C Note (2002) http://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228.
7. Apache Foundation: Apache Ant (2000-2006) http://ant.apache.org.
8. Walsh, N., Milowski, A.: XProc: An XML Pipeline Language. W3C Working Draft (2006) http://www.w3.org/TR/xproc.

A Scenario solution

For purely informative purposes, the scenario described in the paper's introductory section is presented in its full form, as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<xpr:rules xmlns:xpr="http://www.rlopes.net/xml/xpr"
           xmlns:p="http://www.w3.org/2006/XProc"
           xmlns:html="http://www.w3.org/1999/xhtml"
           xmlns:w="http://schemas.microsoft.com/office/word/2003/2/wordml">


  <!-- sensor rules -->
  <xpr:sensor format="xh10"   test="/html:html and not(//html:ruby)"/>
  <xpr:sensor format="xh11"   test="/html:html and //html:ruby"/>
  <xpr:sensor format="wordml" test="/w:wordDocument" />

  <xpr:sensor format="odt"    schema-href="odf.rng"      schema-type="relaxng"/>
  <xpr:sensor format="fo"     schema-href="fo.rng"       schema-type="relaxng"/>
  <xpr:sensor format="db4"    schema-href="docbook4.rng" schema-type="relaxng"/>
  <xpr:sensor format="db5"    schema-href="docbook5.rng" schema-type="relaxng"/>

  <!-- template rules -->
  <xpr:template in="odt" out="db4">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="odt-to-db4.xsl"/>
```

```
  </p:step>
</xpr:template>

<xpr:template in="odt" out="xh10">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="odt-to-xh10.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="wordml" out="db4">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="wordml-to-db4.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="wordml" out="xh10">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="wordml-to-xh10.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="db4" out="db5">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="db4-to-db5.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="db5" out="fo">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="db5-to-fo.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="db5" out="xh10">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="db5-to-xh10.xsl"/>
  </p:step>
</xpr:template>

<xpr:template in="xh10" out="db5">
  <p:step name="{$name}" type="p:xslt">
    <p:input port="document" step="{$from}" source="{$source}"/>
    <p:input port="stylesheet" href="xh10-to-db5.xsl"/>
  </p:step>
```

```
  </xpr:template>

  <xpr:template in="xh10" out="fo">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="xh10-to-fo.xsl"/>
    </p:step>
  </xpr:template>

  <xpr:template in="xh10" out="xh11">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="xh10-to-xh11.xsl"/>
    </p:step>
  </xpr:template>

  <xpr:template in="xh11" out="xh10">
    <p:step name="{$name}" type="p:xslt">
      <p:input port="document" step="{$from}" source="{$source}"/>
      <p:input port="stylesheet" href="xh11-to-xh10.xsl"/>
    </p:step>
  </xpr:template>

</xpr:rules>
```