An Introduction to XML Pipelines

Erik Bruchez
Architect
Orbeon, Inc.

Even though XML is becoming increasingly ubiquitous, there is no standard dedicated to the orchestration of XML technologies. To fill this void, we introduce a new component model with the concept of XML processors. We also define the XML Pipeline Definition Language (XPL), a declarative and implementation-agnostic language designed to describe XML pipelines of arbitrary complexity. Finally, we illustrate the numerous possibilities opened by those new technologies to the world of Web applications and more broadly to XML processing.

Document Version 1.2

Table of Contents

1 Introduction

1.1 An Actual Problem

Since the inception of XML, a wealth of tools has been created to edit, store, parse, transform, exchange and present XML content.[1] In addition, countless XML applications (also called XML vocabularies) have been defined[2], and there is no sign the trend is slowing down.

Save for the simplest applications, take transforming an XML document into another XML document, XML tools often require to be combined. For example, after an XSLT transformation, one may want to validate the resulting document.

The result is a need for a system capable of combining XML content and XML tools allowing for high flexibility and performance. However, the mechanisms available today for that purpose are almost completely lacking. This is, according to James Clark, one of the five current challenges for XML:

> *"One major omission from XML standardization is a well-defined mechanism for describing the pipeline of processing used to handle XML documents. This is especially important given the diversity of processing tools and methods available."*[3]

In the following sections, we start by illustrating the difficulty of dealing with today's tools, and then present the prerequisites for a solution.

1.2 A Simple Example

XSLT is an XML transformation language initially designed as part of the W3C's Extensible Stylesheet Language (XSL) to perform transformations on XML documents. At a high level, an XSLT transformer is a black box with two inputs and one output[4], all of which are XML documents:

- An input document (the document to transform)
- A stylesheet (specifies how to transform the input document)
- An output document (the transformed document)

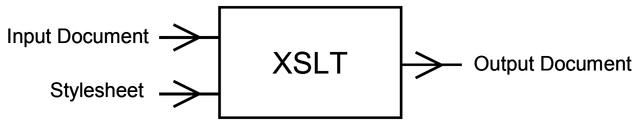

Figure 1 - XSLT Transformer

There are two primary ways of implementing such a transformation:

- Using a command-line transformer that will read and parse the input documents and output the result as files
- Developing an application that will perform the transformation using (for example) JAXP (Java^{TM} API for XML Processing)

[1] Among those are XML parsers, XSLT transformers, XML validators, SVG engines, Web Services, and scores of APIs for different programming languages to support those tools.
[2] Among those are XHTML, XSLT, XSLFO, Relax NG, XForms, XQuery and XUL.
[3] See *Keeping pace with James Clark, An interview (and analysis) with the leading authority on markup languages*, at: `http://www-106.ibm.com/developerworks/xml/library/x-jclark.html`
[4] An XSLT 1.0 transformation can actually have several input documents through the use of the `document()` function. It also supports non-XML outputs such as HTML and text outputs. For the sake of this argument, we only consider the capabilities of XSLT related to handling pure XML.

The first solution is efficient when information is available in the form of files and a single transformation is applied. However, this method does not lend itself to situations where high-performance is important. It requires parsing the files every time a transformation is performed and does not allow for optimizations such as caching stylesheets.

The second solution is more flexible, but requires an extensive knowledge of XML APIs. For example, in Java it is required to know JAXP to read and parse the files needed for a transformation. When cascading two XSLT transformations, a SAX pipeline is required, or the intermediate results must be stored in a DOM tree. The learning curve of those XML APIs is steep and out of reach for non-programmers. The main drawback of a low-level API-based solution is its inherent complexity, which increases when adding advanced features such as conditionals, validation, caching and debugging.

1.3 Towards a Solution

There is today no high-level standard API that describes how to interface with XML tools. There is not even a standard definition of the common characteristics of an XML tool. The first step towards a solution is defining the characteristics shared by most XML tools in order to create a workable definition. We group the characteristics common to XML tools under the concept of XML processor[5] that, at a high level, can be considered an XML component producing and consuming XML documents.[6]

XML processors are described in more details in section 2 below.

The second step consists in defining how XML processors interact to form XML pipelines. This can be achieved at the API level, with an XML processor API, or with the XML Pipeline Definition Language (XPL), a simple declarative, implementation-agnostic language that allows defining XML pipelines of arbitrary complexity.

XML pipelines and XPL are described in more details in section 3 below.[7]

Once the concepts of XML processors and XML pipelines are available, a whole new approach to building applications becomes possible. XML processors can provide Web applications with database access, form handling and validation, as well as access to Web Services and EJBs. Each processor is configured using a specific XML vocabulary that naturally tends to be declarative. We go into further details about this approach in section 4 below.

[5] The term XML processor is used mostly to refer to XML parsers, but we find it adequate to describe any component that consumes and produces XML.
[6] It is also important to define a language-agnostic XML processor API. This is however outside the scope of this document.
[7] A formal language specification is outside the scope of this document.

2 XML Processors

2.1 Definition

An XML processor (called simply *processor* in the rest of this document) is the fundamental building block of an XML pipeline.

Definition: An XML processor is an entity composed of:

- An interface
- A behavior

An XML processor interface consists in a set of inputs and outputs identified by a *name*.[8] Inputs and outputs respectively consume and produce well-formed XML documents. Optionally, a *schema URI* property can be associated with each input or output in order to validate the XML documents consumed and produced.[9]

The behavior of a processor determines what output it produces. A processor behavior is defined by a URI identifying the implementation of the processor. In the case of a processor implemented in Java, such a URI can conveniently refer to a class name or a JNDI name.

A processor can have side effects and its outputs don't have to depend only on its inputs.

2.2 Formalism

A processor is conveniently represented graphically:

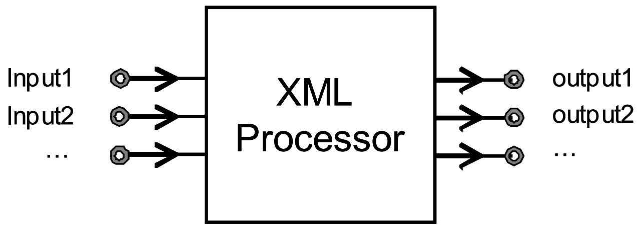

Figure 2 - XML Processor

A processor can also be represented by an XML fragment:

```
<p:processor uri="oxf/processor/xslt"
             xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:input name="input1"/>
    <p:input name="input2"/>
    ...
    <p:output name="output1"/>
    <p:output name="output2"/>
    ...
</p:processor>
```

[8] For a given processor, all inputs must have different names and all outputs must have different names, but an input may have the same name as an output.
[9] There is no restriction as to what schema language can be used. Examples include DTD, W3C XML Schema, and Relax NG.

2.3 Conventions

2.3.1 Input and Output Names

There is no requirement for how inputs and outputs are named. Very often processors deal with documents that are more appropriately considered as "data", or "contents", and documents that are more appropriately considered as "configuration". For example, in general an XSLT stylesheet configures an XSLT transformation that operates on an input document. The convention is for processors to name their inputs and outputs dealing with XML data `data`, while configuration inputs are named `config`.

NOTE: There is no specific notion of processor configuration. For this purpose a regular input is used. If a processor requires to be configured with an URL, it can define a `config` input expecting a document like this:

```
<url>file:/my-file.xml</url>
```

Such a mechanism allows for arbitrarily complex configurations.

2.3.2 Types of Processors

Processors can be arbitrarily categorized according to the number of "data" inputs and outputs they have.

- A processor with no data input but at least one data output is called a *generator*. For example, a processor generating an XML output from a file on a disk can be called a *file generator*.
- A processor with no data output but at least one data input is called a *serializer*. For example, processors generating HTML or SVG to a file from an XML input can be called respectively *HTML serializer* and *SVG serializer*.
- A processor with data inputs and outputs is called a *transformer*. For example, a processor performing an XSLT transformation can be called an *XSLT transformer*.

2.4 Non-XML Contents

The XML processor interface does not provide any mechanism to describe how a processor interacts with non-XML contents. It is up to the processor implementation to define what mechanisms and APIs are used to access non-XML content.

For example a processor can generate XML from a flat file on a disk. Such a processor will have a `data` output generating the XML document from the file, and a `config` input specifying which file the processor needs to read. Similarly, an SVG processor could have a `data` input taking an SVG document and output a PNG image to a file.

3 XML Pipelines

3.1 Definition

An XML pipeline is a system composed of one or more XML processors connected together. For example two XSLT transformers in a chain, with the `data` output of the first transformer connected to the `data` input of the second transformer, constitute an XML pipeline:

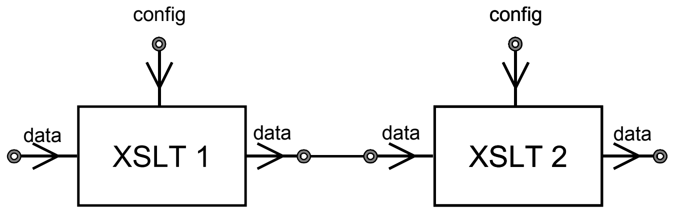

Figure 3 - Simple Pipeline

A more formal definition follows.

Definition: An XML pipeline is composed of:

- An XML processor interface
- A collection of XML processors
- A collection of connections
- A collection of conditionals

A connection joins one processor output to one processor input. Conditionals are described in more details below.

3.2 The XML Pipeline Definition Language (XPL)

The following representation in the XML Pipeline Definition Language (XPL) is equivalent to Figure 3 above:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="documents/input-document.xml"/>
        <p:input name="config" href="stylesheets/stylesheet1.xsl"/>
        <p:output name="data" id="first-transform"/>
    </p:processor>
    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="#first-transform"/>
        <p:input name="config" href="stylesheets/stylesheet2.xsl"/>
        <p:output name="data"/>
    </p:processor>
</p:config>
```

The second processor is connected to the first one by:

- Assigning an identifier with an `id` attribute on the output of the first processor.
- Referencing that identifier with an `href` attribute on the input of the second processor. When referenced in an `href` the id is prefixed with a "#" character.[10]

The input document and the stylesheets are external to the pipeline, and referenced using a URI as the value of an `href` attribute on the relevant inputs. The protocols supported by the `href` attribute are implementation-dependent. For example the `file:` protocol can be implemented in order to read files from a disk.

In addition to using the `href` attribute, it is possible to inline the contents of an input:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="documents/input-document"/>
        <p:input name="config">
            <xsl:stylesheet version="1.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                <xsl:template match="/">
                    <html>
                        <xsl:apply-templates/>
                    </html>
                </xsl:template>
            </xsl:stylesheet>
        </p:input>
        <p:output name="data" id="first-transform"/>
    </p:processor>
</p:config>
```

[10] This is a common mechanism to refer to a resource in the current document.

This is very convenient for short XML documents as it reduces the number of separate files needed.

3.3 XPL Processor and Sub-Pipelines

Instead of accessing its inputs with `href` attributes, the pipeline described above in Figure 3 above can be considered an entity with one input for the source document, two inputs for the stylesheets, and one output for the result document. This is the same as considering this pipeline itself as a processor:

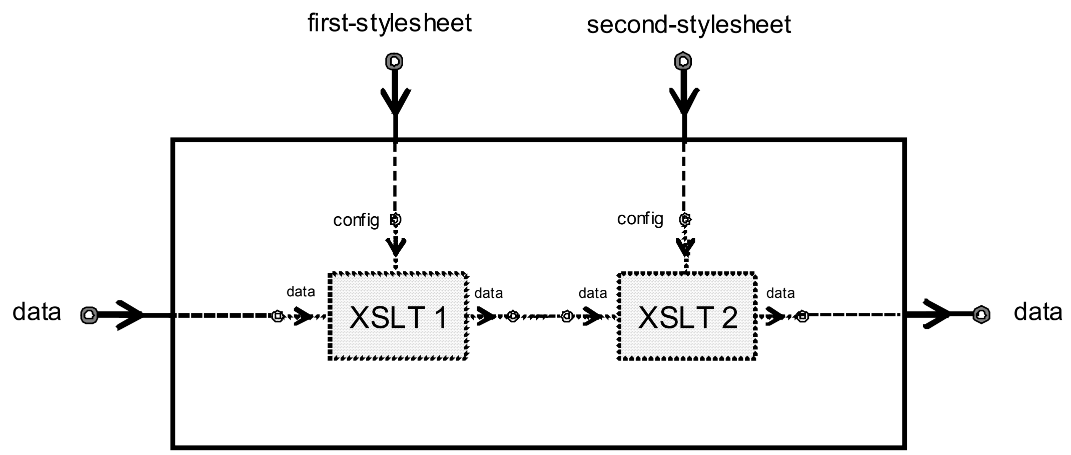

Figure 4 - Pipeline as an XML Processor

Such a processor can be used as a *sub-pipeline* in another pipeline called its *parent pipeline*. To be used this way a pipeline definition must declare an interface if it has any inputs or outputs. The interface of the pipeline is highlighted in bold in the following example:[11]

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:param type="input" name="data"/>
    <p:param type="input" name="first-stylesheet"/>
    <p:param type="input" name="second-stylesheet"/>
    <p:param type="output" name="data"/>

    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="#data"/>
        <p:input name="config" href="#first-stylesheet"/>
        <p:output name="data" id="first-transform"/>
    </p:processor>
    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="#first-transform"/>
        <p:input name="config" href="#second-stylesheet"/>
        <p:output name="data" ref="data"/>
    </p:processor>
</p:config>
```

The inputs and outputs of the contained processors may refer to the inputs and outputs of the pipeline using the `href` attribute.

As XPL is an XML vocabulary, one can create an XPL processor able to interpret XPL itself. An XPL processor can be defined with one `config` input:

[11] As usual, input and output names are arbitrary.

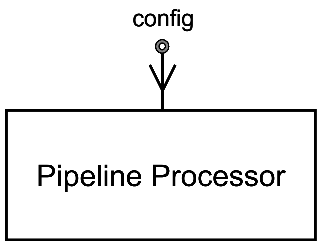

Figure 5 - XPL Processor

An XPL processor may have no other input or output in addition to its `config` input.

It is possible to generate the configuration of an XPL processor dynamically. For example, use an XSLT transformation to generate a pipeline depending on a configuration file:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:processor uri="oxf/processor/xslt">
        <p:input name="data" href="documents/configuration.xml"/>
        <p:input name="config">
            <xsl:stylesheet version="1.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                <xsl:template match="/">
                    <xsl:choose>
                        <xsl:when test="count(elements) > 1">
                            <p:processor uri="oxf/processor/xslt">
                                <p:input name="data" href="documents/input-1.xml"/>
                                <p:input name="config" href="stylesheet-1.xsl"/>
                                <p:output name="data" id="transform"/>
                            </p:processor>
                        </xsl:when>
                        <xsl:otherwise>
                            <p:processor uri="oxf/processor/xslt">
                                <p:input name="data" href="documents/input-2.xml"/>
                                <p:input name="config" href="stylesheet-2.xsl"/>
                                <p:output name="data" id="transform"/>
                            </p:processor>
                        </xsl:otherwise>
                    </xsl:choose>
                    <p:processor uri="oxf/processor/text-serialzer">
                        <p:input name="data" href="#transform"/>
                    </p:processor>
               </xsl:template>
            </xsl:stylesheet>
        </p:input>
        <p:output name="data" id="sub-pipeline-config"/>
    </p:processor>
    <p:processor uri="oxf/processor/pipeline">
        <p:input name="config" href="#sub-pipeline-config"/>
    </p:processor>
</p:config>
```

From an implementation perspective, an XPL processor can be the heart of an XML pipeline system, because it contains the logic required to create and connect other XML processors.

3.4 Conditionals

Generating a pipeline configuration dynamically using a scripting language rapidly becomes difficult to understand and may have an impact on runtime performance. Simple conditionals are, therefore, directly supported in the PDL. The general syntax, very close from XSLT, is as follows:

```
<p:choose href="#condition-document">
    <p:when test="first-condition">
        ...
    </p:when>
    <p:when test="second-condition">
        ...
    </p:when>
    ...
    <p:otherwise>
        ...
    </p:otherwise>
</p:choose>
```

The conditions are expressed in XPath and operate on the XML document specified by the `href` attribute on `p:choose`. Each branch can contain regular processor declarations as well as nested conditions. The example above can be rewritten using PDL conditionals:

```
    <p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
        <p:processor uri="oxf/processor/pipeline">
            <p:input name="config">
                <p:choose href="documents/configuration">
                    <p:when test="count(elements) > 1">
                        <p:processor uri="oxf/processor/xslt">
                            <p:input name="data" href="documents/input-1.xml"/>
                            <p:input name="config" href="stylesheet-1.xsl"/>
                            <p:output name="data" id="transform"/>
                        </p:processor>
                    </p:when>
                    <p:otherwise>
                        <p:processor uri="oxf/processor/xslt">
                            <p:input name="data" href="documents/input-2.xml"/>
                            <p:input name="config" href="stylesheet-2.xsl"/>
                            <p:output name="data" id="transform"/>
                        </p:processor>
                    </p:otherwise>
                </p:choose>
                <p:processor uri="oxf/processor/text-serialzer">
                    <p:input name="data" href="#transform"/>
                </p:processor>
            </p:input>
        </p:processor>
    </p:config>
```

Outputs declared in a branch are subject to the following conditions:

- An output id cannot override an output id in scope before the corresponding `choose` element
- The scope of an output id is local to the branch if it is connected inside that branch[12]
- The set of output ids not connected inside a branch becomes visible to processors declared after the corresponding `choose` element
- The set of output ids not connected inside the branch must be consistent among all branches

The last conditions means that if a branch has two non-connected outputs, e.g. `output1` and `output2`, then all other branches must declare the same outputs. On the other hand, inputs in branches do not have to refer to the same outputs.

It is convenient to consider that a `choose` element has an interface like any XML processor. This is easier to understand graphically:

[12] This may change in future revisions of the language.

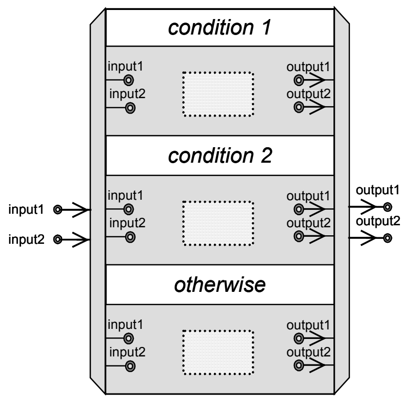

Figure 6 - Conditionals

3.5 The href Attribute

We have already seen in previous sections that the `href` attribute on `input` and `choose` elements can be used to:

- Reference external documents
- Reference outputs of other processors

In addition, it is also possible to use the `href` attribute to:

- Aggregate documents using the `aggregate()` function
- Select part of a document using XPointer[13]

The complete syntax of the `href` attribute is described below in a BNF-like syntax:

```
href              ::= ( local_reference | uri | aggregation ) [ xpointer ]
local_reference   ::= "#" id
aggregation       ::= "aggregate(" root_element_name "," agg_parameter ")"
root_element_name ::= "'"  name "'"
agg_parameter     ::= href [ "," agg_parameter ]
xpointer          ::= "#xpointer(" xpath_expression ")"
```

3.5.1 URI

The URI syntax is formally defined in RFC 2396. A URI is used to reference an external document. A URI can be either:

- Absolute, if a protocol is specified. For instance "`file:/dir/file.xml`".
- Relative, if no protocol is specified. For instance "`../file.xml`". The document is loaded relatively to the URL of the PDL document where the `href` is declared, as specified in RFC 1808.

3.5.2 Aggregation

Multiple documents can be aggregated with the `aggregate()` function.

- The name of the root element that will contain the aggregated document is specified as the first argument.
- The documents to aggregate are specified as the following arguments. There is no restriction on the number of documents that can be aggregated.

[13] At the time of writing, XPointer is a W3C working draft.

For instance, if we have a first document (with output id `first`):

```
<employee>John</employee>
```

And a second document (with output id `second`):

```
<employee>Marc</employee>
```

Those two documents can be aggregated with `aggregate('employees', #first, #second)`. This produces the document:

```
<employees>
    <employee>John</employee>
    <employee>Marc</employee>
</employees>
```

3.5.3 XPointer

The XPointer syntax can be used select parts of a document. For instance, if we have a document (in a file called `company.xml`):

```
<company>
    <name>Orbeon</name>
    <site>
        <web>http://www.orbeon.com/</web>
        <ftp>ftp://ftp.orbeon.com/</ftp>
    </site>
</company>
```

The expression `company.xml#xpointer(/company/site)` produces the document:

```
<site>
    <web>http://www.orbeon.com/</web>
    <ftp>ftp://ftp.orbeon.com/</ftp>
</site>
```

3.5.4 Multiple References to an Identifier

The same id may be referenced multiple times in the same PDL document. For instance the id `doc` is referenced by two processors in the following example:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
    <p:processor uri="A">
        <p:output name="data" id="doc"/>
    </p:processor>
    <p:processor uri="B">
        <p:input name="data" href="#doc"/>
    </p:processor>
    <p:processor uri="C">
        <p:input name="data" href="#doc"/>
    </p:processor>
</p:config>
```

It is guaranteed that the document seen by B and C are identical.

This situation can be graphically represented as:

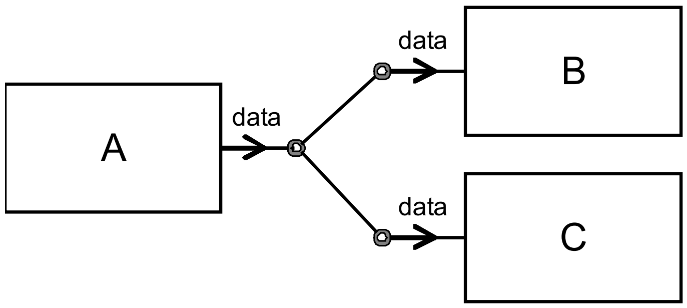

Figure 7 - Multiple References to an Identifier

3.6 Processing Model

Executing a pipeline means producing one document for each of its outputs. Starting with the pipeline inputs, it is possible to determine an execution order assuming that each processor requires reading all its inputs before generating its outputs. A pipeline consisting of only one XSLT transformer, for example, first requires feeding the XSLT transformer with its stylesheet, then with the document to transform. However, this requires prior knowledge of the processor's behavior.

Another solution consists in starting with the pipeline outputs and, for each of those, asking the associated processor to generate that output. It is then up to the processor to determine which inputs it needs and read those, cascading until all outputs can be generated. This lazy evaluation approach is simple to implement and has further benefits such as facilitating the implementation of caches.

Consider an XSLT transformer in a pipeline, where the XSLT transformer's output is connected to a file serializer, and both its inputs are connected to file generators. Assuming that the generators and serializers do not need configuration and know which file to read and to which file to write:

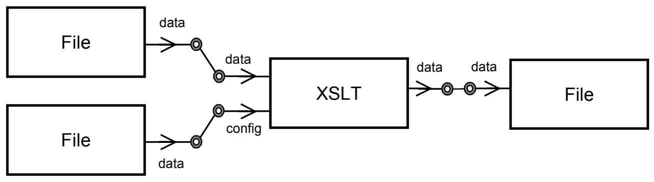

Figure 8 - Simple Pipeline with XSLT Transformer

Processing will be as follows:

1. The file serializer, only processor without output in this example, is told to start processing.
2. In order to write to the file the serializer requires its `data` input. It asks the XSLT transformer to write its `data` output.
3. The XSLT transformer requires first its `config` input in order to create a native transformer object (using for example JAXP in Java) representing the stylesheet. It asks the file generator connected to its `config` input to write its `data` output.
4. The file generator reads the file on disk, parses it, and outputs an XML document on its `data` output.
5. The XSLT transformer receives the XML document on its `config` input. It creates a native transformer object.
6. The XSLT transformer asks the file generator connected to its `data` input to write its `data` output.
7. The file generator reads the file on disk, parses it, and outputs an XML document on its `data` output.

8. The XSLT transformer receives the XML document on its `data` input. It performs the transformation and generates its `data` output.
9. The file serializer receives its `data` input and finally serializes the XML document into the file.

Processors having multiple outputs can be slightly more difficult to deal with. A processor might have to generate different outputs at different times from the same input. It is not possible to read the input every time an output must be generated, as the input may change. Therefore the processor must store the XML document generated on its first input, and then subsequently use the stored document when generating serving other outputs.

This is not the only way to execute a pipeline, but it has proven very effective. Improvements such as parallelism could be introduced to reduce latency and improve performance.

3.7 Validation

It is often crucial to ensure documents produced or consumed by a processor are valid. In the context of XML pipelines, there are two distinct types of data validation:

- Processor-defined validation
- User-defined validation

The first type of validation consists in mandating that, for a given processor, an input or output be validated against a schema. For example, an XPL processor's `config` inputs must respect certain syntax in order to be correctly interpreted. A processor can perform additional validations beyond the validation level supported by the schema language, but automatic schema validation is an extremely convenient first step. Users of processors do not need to take any action to enable processor-defined validation.

The second type of validation is supported by the `schema-href` and `schema-uri` attributes on `input` and `output` elements in XPL. They allow users to refer to an XML schema[14] on any processor input and output. `schema-href` refers to a document using the semantics of the `href` attribute (see section 3.5 above). `schema-uri` adds an indirection level, where an external mapping defines how schema URIs map to the actual documents containing the schema.[15]

User-defined validation is performed in addition to processor-defined validation. For example, a form processor can make sure the general structure of the input document representing the form is correct, while the user in the application can specify the format of the specific form being handled at that point in the pipeline. These are examples of how schemas are specified:

```
<p:input name="data" href="file.xml"
         schema-href="schemas/address-book.rng"/>
<p:input name="data" href ="file.xml"
         schema-uri="http://www.orbeon.com/oxf/pipeline"/>
```

Both the output and input of a connection can be validated using different schemas. A given processor can make sure its output respects a certain format, while the corresponding input needs to respect a stricter schema. This means that in some cases, up to three different validation steps can take place for a single document.

In the following example, the data output of the first processor is first validated against `schema1.rng`, then `schema2.rng`:

```
<p:processor uri="oxf/processor/xslt">
    <p:input name="data" href="input-document.xml"/>
```

[14] Different types of XML schema languages are available, in particular W3C Schemas and Relax NG. In addition, legacy DTDs are still widely used. An XPL processor for example can determine the schema type based on extension and/or contents.
[15] This is similar to the concept of SYSTEM and PUBLIC ids used by DTDs.

```
    <p:input name="config" href="my-stylesheet.xsl"/>
    <p:output name="data" id="transformation" schema-uri="schemas/schema1.rng"/>
</p:processor>
<p:processor uri="oxf/processor/xml-serializer">
    <p:input name="data" href="#transformation" schema-uri="schemas/schema2.rng"/>
</p:processor>
```

3.8 Caching

The domain of caching is quite vast and it is beyond the scope of this section to present more than a few hints of what can be accomplished with caching mechanisms.

Consider a variation of the pipeline described in Figure 8 above. The pipeline is created by a Java Servlet or similar server-side framework. The pipeline reads files on a disk depending on a configuration provided by the HTTP request object, transforms them using an XSLT transformer, and serializes them to HTML into an output stream obtained from the HTTP response object:

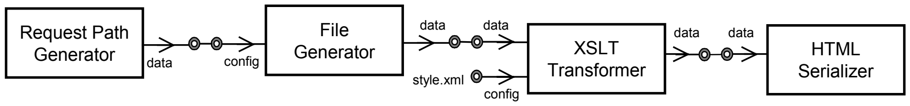

Figure 9 - Simple Web-Publishing Pipeline

What this pipeline accomplishes is serving HTML documents produced by transforming XML documents on a disk through an XSLT stylesheet. A different document is served to the user for every different URL entered. For example, `http://www.example.org/file1.xml` may cause the file generator to read and parse `/home/example/documents/file1.xml`, while `http://www.example.org/another/file.xml` may cause it to read and parse `/home/example/documents/another/file.xml`. We intentionally leave implementation details out.

The performance of such a pipeline may suffer under high load. For example, creating an XSLT native transformer object is relatively time-consuming with most of today's XSLT transformers, in particular if the stylesheet is compiled instead of being interpreted. If the XML document representing the stylesheet does not change, it makes sense to not recompile the stylesheet and to cache the native transformer object. This type of caching, consisting in caching objects associated with XML documents, is called object-level caching.

Another type of caching consists of caching XML documents themselves. For example, if neither the data source file nor the stylesheet have changed, and the stylesheet does not have side effects[16], the XSLT transformer's output can be cached and reused, saving the whole step of the XSLT transformation at the cost of some memory or disk space. This type of caching is called document-level caching. The features of a document-level cache can include advanced features such as persistent storage and cache pre-population.

In Figure 9 above, it does not make much sense to cache the XML content resulting from the XSLT transformation, since it is immediately serialized into HTML. It is possible to cache the serialized HTML output. If the input documents rarely change, as is the case in many web-publishing systems, it is efficient to execute the transformation once and to serve the resulting document from cache for all subsequent queries.

[16] XSLT 1.0 can be considered a functional language and does not have side effects. However, XSLT transformers often include non-standard extensions that introduce side effects, such as functions providing the current date, or interfaces with native languages. If stylesheets make use of such extensions, they are no longer guaranteed to be free of side effects and two executions of the same transformation on the same input may yield different outputs. It is important to consider this fact when designing pipeline caches.

4 A New Approach to Building Applications

4.1 Fundamental Processors

The pipeline in Figure 9 above illustrates how XML pipelines can be used to implement a simple web-publishing application.

Creating a new pipeline using XPL is simple, but implementing a new processor is typically more involved as it requires using a regular programming or scripting language. This shows the importance of creating reusable XML processors that can be recombined in multiple ways. Some fundamental reusable processors have already been mentioned in this document. They are listed in the following table, along with similar processors:

Name	Function
XPL processor	Interprets XPL and builds pipelines
XSLT transformer	Performs an XSLT transformation
URL generator	Generates an XML document from a URL
HTML serializer	Serializes an XML document into a byte stream as per XSLT's HTML output method
XML serializer	Serializes an XML document into a byte stream as per XSLT's XML output method
Text serializer	Serializes an XML document into a byte stream as per XSLT's Text output method
SVG serializer	Serializes an XML document as an image from an SVG description

Table 1 - Common XML Processors

4.2 Towards Reusable XML Components

In order to build more complex applications, additional general-purpose processors are needed, in particular to address the need for:

- Data access (SQL, LDAP, XML databases)
- Interaction with existing components (EJB, Web Services)
- Access to a Web application context (HTTP request)
- Web application dispatching and form handling

The purpose of this section is not to describe in details processors performing such functionality, but to highlight recurrent features they are likely to exhibit.

As an example, consider a processor designed specifically to perform SQL access, in particular selects, updates and inserts. Since SQL is not an XML vocabulary, it is clear that a special XML vocabulary should be used. Such a language, inspired by the Cocoon ESQL tag library, can look like this:

```
<sql:config xmlns:sql="http://orbeon.org/oxf/xml/sql">
    <query-results>
        <sql:connection>
            <sql:pool>test</sql:pool>
            <users>
                <sql:execute>
                    <sql:query>
                        select userid, name
                          from employee
                         order by name
                    </sql:query>
                    <sql:results>
                        <sql:row-results>
                            <user>
                                <user-id><sql:get-string column="userid"/></user-id>
                                <name><sql:get-string column="name"/></name>
                            </user>
                        </sql:row-results>
                    </sql:results>
                </sql:execute>
            </users>
        </sql:connection>
    </query-results>
</sql:config>
```

The fragment above performs in a declarative fashion the following actions:

- Obtaining a connection pool
- Executing a SQL query
- Retrieving all the resulting rows and formatting them in XML

The same work done in Java using an imperative syntax would require:

- Knowledge of the JDBC, JNDI, SAX or DOM APIs
- Creating a new class or method within an existing class
- Compilation, packaging and deployment
- Exception handling
- More lines of code

The example above shows by defining a simple declarative XML vocabulary interpreted by a reusable XML processor, the result is increased productivity. Similarly, specific XML processors can interpret other existing or new languages. The following table lists some examples:

Name	Purpose
XQuery, XQL, Oracle SQL/XML	XML database query language
XForms	Web form representation and validation
XPL	Description of XML pipelines
OXF-SQL	SQL database query language
OXF Dispatching Language	Web application dispatching
OXF Component Call Language	Web Services and EJB calling language

Table 2 - XML Processors and XML Vocabularies

5 Conclusion

The perspectives opened by XML processors, XML pipelines and XPL, are numerous. Applications range from standalone applications to Web applications, including publishing frameworks and dynamic web applications. XML pipelines can be used in conjunction with other existing technologies, or to create applications from the ground up.

To fulfill those promises, it is crucial to work towards standardizing the XML processor API and XPL so that libraries of reusable XML processors become widespread. We hope this paper is a contribution to such efforts by demonstrating the importance and laying out the fundamental aspects of XML pipelines.

6 References

More information about OXF, as well as the latest version of this document, can be found at:

`http://www.orbeon.com/oxf/`

An IBM developerWorks article: *Keeping pace with James Clark, An interview (and analysis) with the leading authority on markup languages*:

`http://www-106.ibm.com/developerworks/xml/library/x-jclark.html`

A related JavaWorld article, *Boost Struts with XSLT and XML, An introduction to Model 2X*:

```
http://www.javaworld.com/javaworld/jw-02-2002/jw-0201-strutsxslt_p.html
```

A simple Pipeline Definition Language defined independently by Sun:

```
http://www.w3.org/TR/xml-pipeline/
```

Cocoon, an open source XML publishing framework:

```
http://xml.apache.org/cocoon/
```

Specification of XSL Transformations (XSLT) Version 1.0:

```
http://www.w3.org/TR/xslt
```