Active Tags : mastering XML with XML

Philippe Poulard

Extreme Markup Languages 2007®

Montréal, Québec
August 7-10, 2007

Active Tags : mastering XML with XML

Philippe Poulard
INRIA

Abstract

Many XML languages are defining tags for processing purpose rather than for describing datas : XSLT, Apache's Ant, or more recently XProc. They are all focusing on a single problematic but could rely on a common framework that would supply a set of services and interfaces.

This paper discusses Active Tags, a language-independant and general-purpose XML system for native XML programming, that aims to be a generic runtime container for several runnable markup languages. We'll depict the architecture of the system and show a few of its features : browsing non-XML datas with XPath, designing macro-tags, mixing declarative sentences with imperative constructs, and filtering SAX streams with XPath patterns.

Extreme Markup Languages®

Active Tags : mastering XML with XML

Table of Contents

Extreme Markup Languages®

Active Tags : mastering XML with XML

Philippe Poulard

§ Rationale

Lots of processing-oriented XML languages have been designed for a specific purpose. For example :

- [XSLT], that is mostly used for transforming XML to HTML.
- [Apache's Ant], that is a kind of "make file" for Java, runnable in batch mode.
- [XProc], that specifies how to perform operations on XML documents.
- [JSP]/[JSTL]/Taglibs[1], that aims to supply server-side operations within Web pages.
- [Apache's Jelly], that also defines a set of tag libraries.

Each of them suffers of its specialization ; for example :

- XSLT is poorly extensible. For example, a set of tags has been designed in Saxon[2] to perform SQL queries ; but it is impossible to deal with binary datas (BLOB). Moreover, XSLT can't be extended with any kind of library : designing a web library that would match incoming URLs against regular expressions is irrelevant in the context of XSLT.
- Apache's Ant is stronlgy related to Java, and stronlgy related to application packaging and deployment problematics.
- XProc is limited to XML documents processing and doesn't offer enough control.
- JSP/JSTL/Taglibs are designed to run within a Java EE[3] Web container ; they are tightly coupled to Web problematics and can't be run in batch mode.
- Apache's Ant and Jelly in one hand, and JSP/JSTL/Taglibs in the other hand are using shell-fashionned references such as `${myVar}` instead of the XPath syntax `{$myVar}` that could be extended for computing values `{po:discount($price) * $tax}` or handling XML nodes `{$xml/po/@total}`. Although those technologies seems to move to a Unified Expression Language [UEL], they are still very far from XML technologies.
- None of them can be used as a support for designing a declarative markup language.

The list of issues is not closed, but the most important point that legitimates the need for designing a common and reusable base for XML processing is that each implementation have to deal with redundent low-level considerations, such as how to bind a tag to its implementation, how to handle variables, or how to mix several tag libraries. Each time that a new processing XML language is designed, it is implemented separately from existing tools, and without taking the benefits of other tag libraries. This is the flaw of many strongly declarative languages such as [W3C XML Schema], [XML Catalogs], [SCXML] and many others : each of them is run within its own engine that at best ignores tags and attributes from a foreign namespace. One of the consequences is that many languages are redefining the same paradigms : for example, although the semantic of an "if-then-else" statement is almost universal in computer sciences, SCXML will define its own tags, XSLT too, and many others too. The design of these languages is neither very open nor really extensible.

We propose in this paper a global solution named [Active Tags] that was previously introduced at [XML 2006].

A global solution

The Active Tags system borrows the best to many technologies and tools like those mentionned previously, and proposes an innovative and reliable framework that can host any new processing-oriented XML language. Moreover, the Active Tags system goes further, near the **extreme** limit of the XML technologies... and beyond. In the following sections we will present a various panel of its outstanding features :

- **Cross-operable objects** : an Active Tags system operates on typed datas ; unlike many similar systems, the datas can be XML datas as well as non-XML datas ; few of the latter are said "XML

friendly", that is to say that they are sensible to XML-related operations. For example, as a file system is a hierarchical structure like XML, XPath expressions such as `//*` can be applied to directories to get all the files under the tree. Such objects are called "cross-operable objects".

- **Macro-tags** : one of the core libraries of Active Tags allows to bind tags and some other XML materials to their implementations. When the implementation of a tag is given inline thanks to other tags, it is a kind of "macro-tag". We'll see how to design an MVC architecture made full of tags.
- **Deep mixity of grammar constructs** : Active Tags also offers means for mixing declarative-oriented[4] languages with imperative sentences. We'll experiment a schema language with this facilities.
- **The architecture at a glance** : Active Tags considers XML technologies as a global system whithin which a set of subsystems that are focusing on a single problematic are cooperating to achieve a user task ; we'll see in this section that Active Tags is also a self-defined system.

In the last section, we will focus on a strategy for pipeline processing of SAX streams through an XPath-based filter **without breaking the streams** : usually, a tool such as an XSLT engine that reads an incoming SAX stream will load it entirely in an internal representation, which cause an overflow memory for too large inputs. The reference implementation of Active Tags support XPath patterns in a very acceptable way that can be combined with locale SAX to DOM conversions.

On that basis, before we go any further, let's look at an **overall presentation** of the system.

§ Overall presentation of the system

Active Tags is not a markup language, it is a set of specifications that describe the Active Tags system and some of its libraries, called "modules". It is also the name of the master specification of the whole set. It is language/platform independant, though a reference implementation is available in Java : the [RefleX] engine (free and open source). An implementation is free of the underlying architecture : interpreter, code generator, unmarshaller or any combination of them can be considered.

The learning curve for people that know XSLT and XPath is very low, because Active Tags is very close to XSLT, although it is not in competition with XSLT ; anyway, familiar XML processes such as XSLT transformations are often used in Active Tags programs. Other concepts should be familiar to people aware of publishing systems such as [Apache's Cocoon], although Active Tags is not limited to publishing.

In the same way that XSLT programs[5] are called "stylesheets", Active Tags ones are called "active sheets". Like XSLT, it is XPath centric, and active sheets will contain both instructions (active tags) and XML litterals. The main difference is that instead of having a single instruction set, an active sheet may contain severals, each bound to a namespace URI. The container that runs an active sheet and that hold the variables is called a processor instance.

One of the core module of the system is the XML Control Language [XCL], that supplies a set of tags that covers many common features :

- Usual control structure actions, such as alternative (`<xcl:if>` `<xcl:then>` `<xcl:else>`)or iterative actions (`<xcl:for-each>`), and logic procedure declaration and invokation.
- XML oriented actions, such as XML parsing (`<xcl:parse>`) and XSLT transforming (`<xcl:parse-stylesheet>` and `<xcl:transform>`) ; these actions deal with entity and URI resolving, passing parameters (`<xcl:param>`), error handling and many other options used to tune XML processes.
- XML document creation (`<xcl:document>`, `<xcl:element>`, `<xcl:attribute>` etc) and high level Active Update implementation, that allow to perform update operations on XML objects and X-operable objects (`<xcl:delete>`, `<xcl:append>` etc).
- Filtering XML streams and plain-text streams (`<xcl:filter>`) by using XPath patterns (`<xcl:rule>`) and regular expressions.

In this first example, XCL is used for parsing a file and transforming it with XSLT :

```
<xcl:active-sheet xmlns:xcl="http://ns.inria.org/active-tags/xcl">
    <xcl:parse name="input" source="file:///path/to/document.xml"/>
    <xcl:parse-stylesheet name="xslt" source="file:///path/to/stylesheet.xsl"/>
    <xcl:transform output="file:///path/to/result.html" source="{ $input }" stylesheet="{ $xslt }"/>
</xcl:active-sheet>
```

In the following example, 2 modules are involved : XCL, and the SYSTEM module. The active sheet creates an XML document dynamically, and serialize it to a file.

```
<xcl:active-sheet
  xmlns:sys="http://ns.inria.org/active-tags/sys"
  xmlns:xcl="http://ns.inria.org/active-tags/xcl">
    <!--get the system property "who"-->
    <xcl:set name="who" value="{ string( $sys:env/who ) }"/>
    <!--create an XML document with some litterals-->
    <xcl:document name="xml">
        <example>
            <title>Hello { $who } !</title>
        </example>
    </xcl:document>
    <!--serialize the XML document to the standard system output ;
        as the transformation doesn't involve a stylesheet, a copy is performed-->
    <xcl:transform output="{ $sys:out }" source="{ $xml }"/>
</xcl:active-sheet>
```

In the next example, the same XML document is created dynamically, but the active sheet is hosted inside a Web server ; the root element consists on an HTTP service that defines a mapping for incoming URLs (thanks to a regular expression).

```
<web:service
  xmlns:web="http://ns.inria.org/active-tags/web"
  xmlns:xcl="http://ns.inria.org/active-tags/xcl">
    <!--
        [webapp]/index.xml
     -->
    <web:mapping match="^/index\.xml$" mime-type="application/xml">
        <!--get the parameter from the query string :
            ...index.xml?who=John+Doe
        -->
        <xcl:set name="who" value="{ string( $web:request/who ) }"/>
        <!--create an XML document with some litterals-->
        <xcl:document name="xml">
            <example>
                <title>Hello { $who } !</title>
            </example>
        </xcl:document>
        <!--serialize the XML document to the HTTP output stream ;
            as the transformation doesn't involve a stylesheet, a copy is performed-->
        <xcl:transform source="{ $xml }" output="{ value( $web:response/@web:output ) }"/>
    </web:mapping>
</web:service>
```

One can remark that every XPath expression is contained in curly braces, like "attribute value templates" in XSLT. But they can occur in active tags, in litterals, and even in text content. Moreover, the result can be any object, not necessary a string.

Like programming languages, the Active Tags system supplies some standard libraries (modules) in specific domains :

- system interactions
- Web
- I/O
- SQL
- and custom modules

§ Cross-operable objects

In the previous section, an exotic variable was used : `$web:response` appeared in an XPath expression with a location step that was selecting one of its attributes, but that didn't refer an XML node, though it behaves like if it was one.

```
<xcl:transform source="{ $xml }" output="{ value( $web:response/@web:output ) }"/>
```

The main stream that tries to make a bridge from the XML world and the OO world tend to "convert" XML abstractions to objects in order to make them usable with OO languages. As Active Tags languages rely on pure XML, the opposite way is taken : some objects are enforced to behave like XML nodes. Not all objects, because although existing techniques such as reflection or bean-ization can help to expose systematically an object as a node, the mapping to XML get automatically wouldn't be necessarily those specified by a designer[6] ; moreover, pragmatically only few objects worth to be considered as XML : in the standards modules of Active Tags, most of the data types defined are not XML-friendly, simply because the more often it is useless. Thus, the mapping of objects to XML fits in the "design-time" category rather

than in the "run-time" category ; a mapping can precisely identify which members of an object become attributes, and which ones become children in which order.

Let's have a closer look at our `$web:response` variable ; it is set automatically when entering a `<web:mapping>`, and is defined to be of the type `web:x-response` ; the "x-" prefix in the name of the data type denotes that a data of this type is a cross-operable object, or X-operable object, that is to say an object sensible to some XML operations, such as browsing it with XPath, or updating it with XML-oriented operations such as `<xcl:append>`, `<xcl:delete>` and others. The Web module specification defines precisely this data type and its capabilities : for example, that its `@web:output` attribute refer to another object, the output stream where to write the HTTP response.

At this stage, we learned that an object can expose another object as an attribute, that can also expose an object as an attribute. This lead to very unusual XPath expressions although syntaxically legal such as `$foo/@FOO/@bar`. The important point is that those objects are not necessarily representable with XML tags : an attribute can't have an attribute in markup representation. However, this is not a drawback but on the contrary a serious advantage for several reasons :

- because the (possible) high cost of round-tripping between XML and the object is avoided (from binary to base64 and base64 to binary)
- because sometimes an XML representation of an object is irrelevant : who cares about the XML representation of an "output stream" object (we are not talking about the content of that stream, but about the object itself) ?
- because delivering the XML structure implies to scan recursively the members of the object, which is sometimes inappropriate : if we hold the root of a file system, it wouldn't be reasonable to represent all the hierarchy with markup whereas the user just want to get an attribute, say the creation date.

The last point to consider is the possible incompatibility of that model with the XML data model. Schema aware applications using schema aware parsers and APIs can make use of the types of elements and attributes. This is a concept described in [W3C XML Schema] and known as PSVI [post-schema-validation infoset]. For example, if an attribute of an identified element is defined to be a `xs:date`, then instead of dealing with the string value of the date, an application can handle the date as an object. This allows sorting a set of dates correctly : if a time zone is specified, sorting regarding the string value of the dates would fail, but sorting regarding the typed datas succeeds. In the same manner that the date object is bound to the attribute after validation, an output stream object is simply bound to the `@web:output` attribute of the `$web:response` X-operable object. The main difference is that no validation neither even parsing have been performed, it is just an inherent characteristic of the host object. Notice that the attribute also has a string value, but that is totally useless in this case, thus it is implementation dependant ; in a Java implementation, it could be something that reflects the underlying object, like `[java.io.OutputStream@189c036]`.

To summerize, the concept of X-operable objects is in essence fully compatible with XML. Various objects can be handled in Active Tags : some related to XML, others not ; some X-operable, others not ; some representable with markups, others not. Of course, any combination of them can occur.

Let's have a look at a more significant example. As a file system is a hierarchic structure, it is a well candidate to be X-operable : the children of a directory would be the files under that directory, and the attributes of a file would be its characteristics, such as the creation date, if it is a file or a directory, its size, etc. The I/O module specification describes that `io:x-file` type in detail. It also defines an XPath function, `io:file()`, to create such objects :

```
    <xcl:set name="baseDir" value="{ io:file('file:///path/to/base/dir/') }"/>
```

Then, selecting a set of files thanks to XPath is straightforward :

- `$baseDir/*` : the files just under the base directory.
- `$baseDir//*` : the files in the tree under the base directory.
- `$baseDir//*[@io:extension='xml']` : the files in the tree under the base directory that ends with ".xml".
- `$baseDir//*[@io:size > 1024]` : some files with a given size.
- `$baseDir/subpath/doc.xml` : a single file.
- `$baseDir/subpath/*[name()='123.xml']` : another single file that has a name which is an illegal XML name that can't appear in a location path.

What is noticeable is the **extreme** ease to get various collections of files. Just consider the amount of code that you would use with your favorite language to achieve the same thing ; then, just consider the amount of time that you would spend if you had to reconsider what to get ; here, you just have to add a predicate or a location path. You don't have any longer to learn another syntax, or to deal with an obscure one like the Unix "find" command : it's XPath !

Below is an example where a set of files is transformed to HTML :

```
<xcl:active-sheet
  xmlns:xcl="http://ns.inria.org/active-tags/xcl"
  xmlns:io="http://ns.inria.org/active-tags/io">
    <xcl:parse-stylesheet name="xslt" source="file:///path/to/stylesheet.xsl"/>
    <xcl:for-each name="file" select="{ io:file('file:///path/to/base/dir/')/*[@io:extension='xml'] }">
        <xcl:parse name="xml" source="{ $file }" style="stream"/>
        <xcl:transform output="file:///path/to/published/{ $file/@io:short-name }.html"
                       source="{ $xml }" stylesheet="{ $xslt }"/>
    </xcl:for-each>
</xcl:active-sheet>
```

And another example that shows how to merge several XML files to a single document. Each document is parsed à la DOM, and the global document is created with SAX, which is suitable for merging numbers of big files without causing a memory overflow. The engine converts transparently DOM to SAX when necessary and vice-versa.

```
<xcl:active-sheet
  xmlns:xcl="http://ns.inria.org/active-tags/xcl"
  xmlns:io="http://ns.inria.org/active-tags/io">
    <xcl:document name="merged" style="stream">
      <root>
        <xcl:for-each name="file" select="{ io:file('file:///path/to/base/dir/')//*[@io:extension='xml'] }">
            <xcl:parse name="input" source="{ $file }" style="tree"/>
            { $input }
        </xcl:for-each>
      </root>
    </xcl:document>
    <xcl:transform output="file:///path/to/merged.xml" source="{ $merged }"/>
</xcl:active-sheet>
```

To summerize, Active Tags offers through XPath a smart mean for handling objects. XPath becomes the unified accessor for browsing non-XML objects like if they were XML ones.

§ Macro tags

Binding an active tag to its implementation is the role of EXP [Extensible XML Processor], a module that can extend the Active Tags processor capabilities. In fact, there are several kind of materials that are "active" :

- active tags : `<p:element>`, that are operations or declarations
- foreign attributes : `@p:attribute` that are rather directives
- predefined properties : `$p:property`
- XPath functions : `p:function()`
- data types : `p:data-type`

Except the last one which is inherent to objects, the custom active materials are defined in an EXP document ; the part that bind the material to the concrete implementation depends on the engine ; for example, the name of a Java class is specified in RefleX :

```
<exp:module
    xmlns:exp="http://ns.inria.org/active-tags/exp"
    xmlns:eml="http://www.extrememarkup.com/2007/active-tags/foo"
    version="1.0"
    target="eml"
>
    <exp:element   name="eml:foo"     source="res:com.extrememarkup.foo.FooAction"/>
    <exp:function  name="eml:bar"     source="res:com.extrememarkup.foo.BarFunction"/>
    <exp:attribute name="eml:version" source="res:org.inria.ns.reflex.processor.core.VersionAttr"/>
</exp:module>
```

If you find in Active Tags enough services that suit your needs, and that you don't want to dive inside the implementation details, you can also omit the `@source` attribute on the active material definition, and supply it inline. This is an elegant mean to define macro-tags and macro-functions that also accept passing parameters ; for example, the I/O module implementation in RefleX defines the `io:exists()` function as a macro :

```
    <exp:function name="io:exists">
        <xcl:set value="{ value( io:file( $exp:args[1] )/@io:exists ) }"/>
    </exp:function>
```

Now that active materials can be defined with macros, it is possible to write a complete application only with tags. Let's do it in conformance with a software engineering paradigm such as MVC [Model-View-Controler], that consists essentially on the separation of well-defined components. If we look in the "overall presentation of the system" section, we have 2 examples that are almost identical : one invokable in batch, the other hosted in a Web server. The idea of a component based architecture is to design a common business-model that provides the same XML result, but that can be applied in batch as well as in a Web application, or even embedded in a non-Web application. It is simply a separation of concerns. To achieve this, we simply design our business-model in an external custom module. The benefits are that this module can be reused in any application. This architecture also enhances the independancy of the different parts of the application. Concretely :

```
<exp:module
    xmlns:exp="http://ns.inria.org/active-tags/exp"
    xmlns:xcl="http://ns.inria.org/active-tags/xcl"
    xmlns:eml="http://www.extrememarkup.com/2007/active-tags/hello"
    version="1.0"
    target="eml"
>
    <!-- Usage : <eml:say-hello
                    who=[the name of the guy to say hello]
                    variable-name=[the name of the variable to export]>
    -->
    <exp:element name="eml:say-hello">
        <!--create an XML document with some litterals-->
        <xcl:document name="xml">
            <example>
                <title>Hello { value( $exp:params/@who ) } !</title>
            </example>
        </xcl:document>
        <exp:exports>
            <exp:export name="{ string( $exp:params/@variable-name ) }" value="{ $xml }"/>
        </exp:exports>
    </exp:element>
</exp:module>
```

Using this module from the command line :

```
<xcl:active-sheet
    xmlns:sys="http://ns.inria.org/active-tags/sys"
    xmlns:xcl="http://ns.inria.org/active-tags/xcl"
    xmlns:eml="http://www.extrememarkup.com/2007/active-tags/hello"
>
    <eml:say-hello who="{ string( $sys:env/who ) }" variable-name="hello"/>
    <xcl:transform
         source="{ $hello }"
         output="{ $sys:out }"
         stylesheet="file:///path/to/hello.xsl"
    />
</xcl:active-sheet>
```

Using this module in a Web application :

```
<web:service
    xmlns:web="http://ns.inria.org/active-tags/web"
    xmlns:xcl="http://ns.inria.org/active-tags/xcl"
    xmlns:eml="http://www.extrememarkup.com/2007/active-tags/hello"
>
    <!--
        [webapp]/index.xml
     -->
    <web:mapping match="^/index\.xml$" mime-type="application/xml">
```

```
        <eml:say-hello who="{ string( $web:request/who ) }" variable-name="hello"/>
        <xcl:transform
            source="{ $hello }"
            output="{ value( $web:response/@web:output ) }"
            stylesheet="web:///WEB-INF/hello.xsl"
        />
        <!--note : the "web:" URI scheme allows to access resources
            deployed within the Web application-->
    </web:mapping>
</web:service>
```

In our tiny MVC architecture :

- The "eml" custom module is our business-model
- The stylesheet "hello.xsl" is our view
- The active sheet invokable in batch and the HTTP service hostable within a web server are our controlers

To summerize, Active Tags becomes an ideal support for user-defined runnable markup languages. Moreover, it allows to define easily tag's implementations with other tags.

§ Deep mixity of grammar constructs

So far, we have seen rather imperative active tags : `<xcl:parse>`, `<xcl:transform>`, `<xcl:set>`, `<eml:say-hello>`... and few declarative ones that were hosting "procedures" : `<xcl:active-sheet>`, `<web:mapping>`, `<exp:element>`...

Even in procedural languages, we find what we call "declarations" in this paper : the declaration of the name of the program, the declaration of a procedure, and in OOP, the declaration of a method. Languages such as W3C XML Schema, SCXML, and XML catalogs are fully declarative, in the sense that they doesn't contain at all imperative operations, or very few ones in the case of SCXML.

In this section we study how to mix deeply imperative sentences within declarations ; this feature is depicted with a schema language.

Schema processors are building an abstract tree from a schema instance. With a traditional grammar-based schema (DTD, W3C XML Schema, [RelaxNG]), as the schema instance is hard-coded, the abstract tree is static, making the expressiveness of the schema limited to what is allowed by the grammar. The flaw with grammars in XML is that they only allow to constraint content models in a declarative manner, which is in essence very concise and expressive, but when the limits of the declarative syntax are reached, there is no way out ; it is still possible to add a new tag to express the missing declarative tag, but the limit still exists a single step further, at the cost of upgrading the language.

In order to be much more expressive without adding tags again and again, a fully declarative schema language that is part of the Active Tags suite has been designed : ASL [Active Schema Language]. The immediate benefit is to avoid to compromise a user's XML structure just because some constraints can't be expressed by grammar-based schemata. ASL contains similar constructs than others schema languages : an element declaration is still made of sequences or choices of element references, texts or attributes, but they are mixed with imperative constructs. As the content models are computed at runtime while validating, the result abstract tree becomes dynamic, increasing dramatically the expressiveness of the schema : the content models can adapt themselves to the incoming data to validate in an **extreme** flexible way. Additionally, ASL allows to compute dynamically occurrence constraints, that are at best hard-coded in existing schema languages.

Before showing our first example, other ASL features that won't be detailed in this paper worth to be mentionned :

- The capability to design semantic data types, which is -curiously- a feature available in almost all programming languages, but missing in XML schema languages ; an initiative that offers a good base for this feature is [DTLL], since a data structure can be built from a raw text.
- The capability to mix together several schema languages : for example, ASL can add type-based constraints on the attributes defined in a DTD. ASL act as a kind of "schema patcher".
- The capability to define internationalized custom error messages.
- And others...

In our example, we design an XML structure like this :

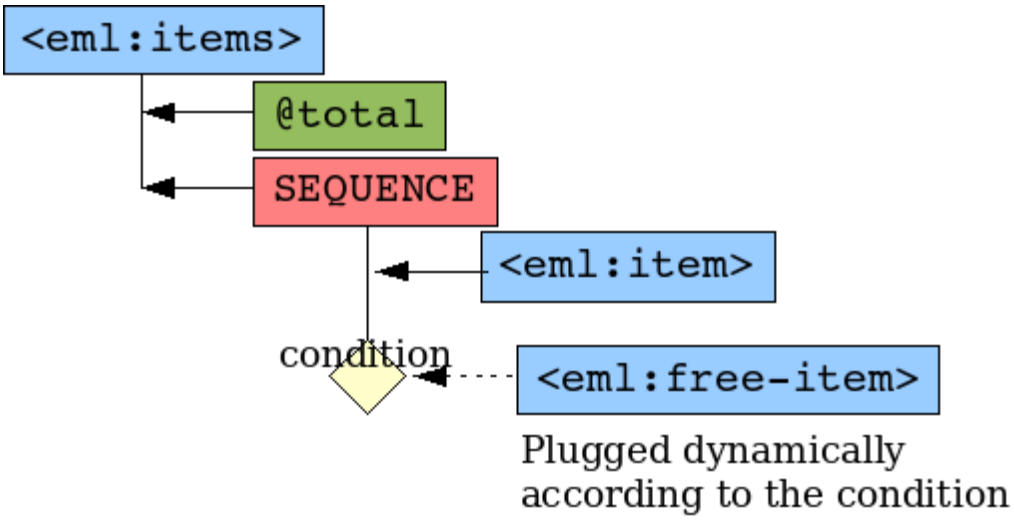
```
<purchase-order xmlns="http://www.extrememarkup.com/2007/active-tags/po">
    <items total="188.93">
      <item partNum="872-AA">
         <productName>Lawnmower</productName>
         <quantity>1</quantity>
         <USPrice>148.95</USPrice>
         <comment>Confirm this is electric</comment>
      </item>
      <item partNum="926-AA">
         <productName>Baby Monitor</productName>
         <quantity>1</quantity>
         <USPrice>39.98</USPrice>
         <shipDate>1999-05-21</shipDate>
      </item>
      <free-item partNum="261-ZZ">
         <productName>Kamasutra for dummies</productName>
         <quantity>1</quantity>
      </free-item>
   </items>
</purchase-order>
```

Nothing special in this structure, except that `<free-item>` elements are allowed only if the total amount exceeds 500$ (which makes the above document invalid). Additionally, the number of free items offered is total/500. With ASL, this can be written exactly as it has been expressed :

```
<asl:active-schema target="eml"
    xmlns:xcl="http://ns.inria.org/active-tags/xcl"
    xmlns:asl="http://ns.inria.org/active-schema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:eml="http://www.extrememarkup.com/2007/active-tags/po"
>
    <asl:element name="eml:purchase-order" root="always">
        <asl:sequence>
            <asl:element ref-elem="eml:items"/>
        </asl:sequence>
    </asl:element>

    <asl:element name="eml:items" root="never">
        <asl:attribute name="total" type="xs:decimal"/>
        <asl:sequence>
            <asl:element ref-elem="eml:item" min-occurs="1" max-occurs="unbounded"/>
            <xcl:if test="{ asl:element()/@total &gt; 500 }">
                <xcl:then>
                    <asl:element ref-elem="eml:free-item"
                        min-occurs="1"
                        max-occurs="{ floor( asl:element()/@total div 500 ) }"/>
                </xcl:then>
            </xcl:if>
        </asl:sequence>
    </asl:element>

    <asl:element name="eml:item" root="never">
        <!--content model here-->
    </asl:element>

    <asl:element name="eml:free-item" root="never">
        <!--content model here-->
    </asl:element>

    <!--other element definitions here-->

</asl:active-schema>
```

Of course, this schema could be simplified, but it demonstrates that an imperative operation is used to build the content model during the validation. The content model of the `<eml:items>` element will vary according to the total amount found in the incoming document.

Figure 1: Component view of a dynamic content model



The abstract tree of the definition of `<eml:items>` may vary at runtime according to the result of the condition expressed in the "if" statement.

Here are several possible "instanciations" of this element definition :

```
<asl:element name="eml:items" root="never">
    <asl:attribute name="total" type="xs:decimal"/>
    <asl:sequence>
        <asl:element ref-elem="eml:item" min-occurs="1" max-occurs="unbounded"/>
    </asl:sequence>
</asl:element>
```

```
<asl:element name="eml:items" root="never">
    <asl:attribute name="total" type="xs:decimal"/>
    <asl:sequence>
        <asl:element ref-elem="eml:item" min-occurs="1" max-occurs="unbounded"/>
        <asl:element ref-elem="eml:free-item" min-occurs="1" max-occurs="1"/>
    </asl:sequence>
</asl:element>
```

```
<asl:element name="eml:items" root="never">
    <asl:attribute name="total" type="xs:decimal"/>
    <asl:sequence>
        <asl:element ref-elem="eml:item" min-occurs="1" max-occurs="unbounded"/>
        <asl:element ref-elem="eml:free-item" min-occurs="1" max-occurs="3"/>
    </asl:sequence>
</asl:element>
```

Each of these "realizations" of the element definition leads to a different abstract tree of the grammar. But all are expressed in a single self-adaptative schema.

[Schematron] is a technology that offers similar services ; however, there is a fundamental difference : Schematron act outside content models whereas ASL define them. Schematron will report constraints violations after grammar-based validation. A tool such as an editor will propose to insert a `<eml:free-item>` whereas it is forbidden. ASL will introduce it in the content model only when the conditions are met, and it will allow the right number of insertions of that element according to the total of items purchased ; of course, to keep this example simple the total is an attribute but a real schema could compute it dynamically.

To summerize, the flexibility affoarded by Active Tags to declarative oriented languages leads to simplicity and efficiency. Applied to ASL, we get a new schema language much more powerful than legacy ones (DTD, W3C XML Schema, Relax NG, Schematron). Moreover, by considering Active Tags while creating new markup languages, the designer will be able to make them more simple and more powerful.

§ The architecture at a glance

We have encountered some tags that are considered as operations, and other tags that are considered as litterals. How the engine makes the difference ? Each module is registered to the engine thanks to a catalog. In Active Tags, a catalog is a mean to bind a name with a resource. Each time the engine encounters a tag, a catalog lookup is performed to find if a module exists for the namespace URI of the tag ; if one exists, it must contains its definition, which allows to load the relevant implementation of the tag.

The concept of XML catalogs has been extended in Active Tags. Active Catalogs are compatible with XML catalogs (and even SGML catalogs), which role is limited to map a name to another name. In the actual XML world, a client-application of a catalog will process the resolved name to get the relevant resource : an XML document, an XSLT stylesheet, a schema, etc. Possible issues with XML catalogs include :

- the resource is supplied in the form of a URI reference.
- the same reference can't be used for different kind of resources.
- a reference can't be mapped to several resources of the same kind.

Active Catalogs offers many useful services for Active Tags :

- the resource is supplied in the form directly suitable to the client-application.
- the same reference can be used for different kinds of resources.
- a reference can be mapped to several instances of a kind of resource.
- different lookup strategies can be considered according to the kind of the resource.
- caching facilities are available for resources requested multiple-time with a retention policy.

Thanks to these **extremely** convenient features, Active Tags can supply various different resources from the same namespace URI :

- a module,
- several schema instances,
- other catalogs,
- a stylesheet,
- a simple URL as a string,
- any user-defined resource.

As shown below, the kind of resource is given thanks to a selector :

```
<cat:catalog
    xmlns:cat="http://ns.inria.org/active-tags/cat"
    xmlns:exp="http://ns.inria.org/active-tags/exp"
    xmlns:asl="http://ns.inria.org/active-schema"
>
    <cat:resource name="http://www.extrememarkup.com/2007/active-tags/po" uri="po-module.exp"
    selector="exp:module"/>
    <cat:resource name="http://www.extrememarkup.com/2007/active-tags/po" uri="po-schema.asl"
    selector="asl:schema"/>
</cat:catalog>
```

In fact, a selector is itself a kind of resource that embeds a lookup strategy, a caching policy, and a receipe for building the resource expected. Active Tags specifies a set of predefined selectors, including those that make Active Catalogs compatible with XML catalogs : for example, a parser that needs to resolve a public and/or system identifier will lookup in the catalogs with the key `xml:external-identifier` ; from the point of view of XML catalogs and SGML catalogs, the implementations of the declarations will be necessarily sensitive to such keys. Moreover, there is no requirement that a resource will be built from a unique type of source ; for example, a schema can be built from any schema technology supported by the underlying engine, and that can be a DTD, a Relax NG schema, a W3C XML Schema, or an Active Schema ; similarly, when a catalog is expected, an instance can be supplied by an XML catalog, an SGML catalog, an Active Catalog, or even by a built-in catalog.

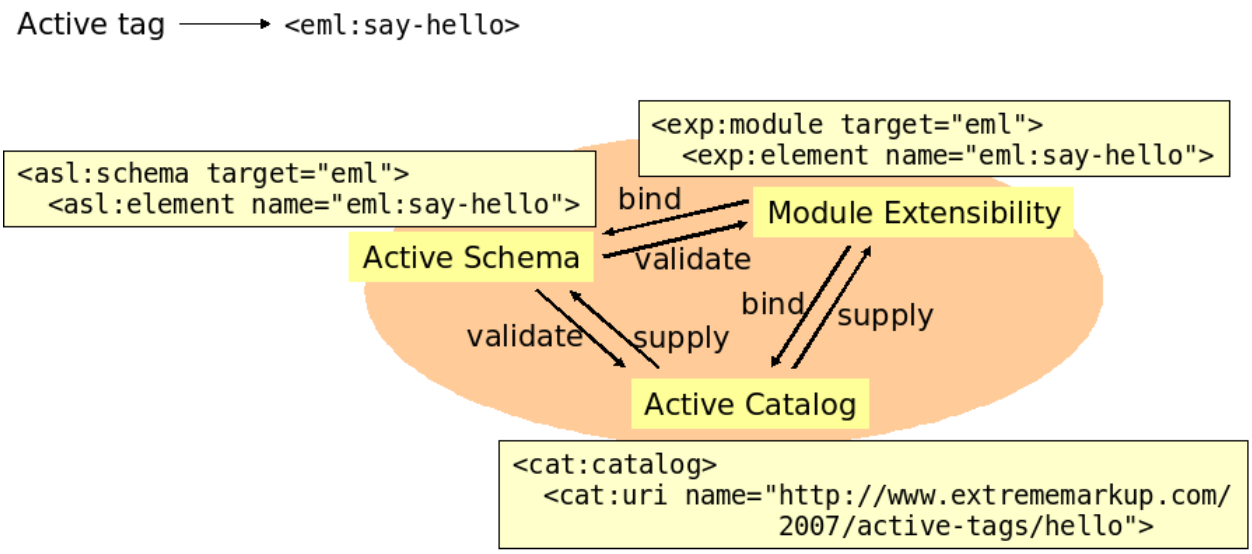
```
<cat:catalog
     xmlns:cat="http://ns.inria.org/active-tags/cat"
>
    <!--an active catalog-->
```

```
    <cat:resource name="http://www.extrememarkup.com/2007/active-tags/catalog1.cat"
        uri="catalog1.cat" selector="cat:catalog"/>
    <!--an SGML catalog-->
    <cat:resource name="http://www.extrememarkup.com/2007/active-tags/catalog2.cat"
        uri="catalog2.sgml" selector="cat:catalog"/>
    <!--a built-in catalog-->
    <cat:resource name="http://www.extrememarkup.com/2007/active-tags/catalog3.cat"
        uri="res:com.extrememarkup.foo.FooCatalog" selector="cat:catalog"/>
</cat:catalog>
```

Finally, Active Catalog is a centric module in the Active Tags engine for retrieving and building resources with a great flexibility. In action, different modules will cooperate to achieve a given task, which is a valuable facet of the system for designing Active Tags applications. The modules involved for validating and binding an active tag to its implementation are :

- the EXP module that binds the class to the active tag,
- the Active Schema module that validates the attributes and the content of the active tag,
- the Active Catalog module that supplies the 2 resources above.

Figure 2: The resolution of an active tag



The engine will lookup in the catalog for the schema that validates the tag and for its implementation. This is made for each active tag, including those that are describing catalogs, modules, and schemata.

Moreover, the EXP module has also itself an EXP module that binds an implementation to its tags, and a schema that expresses constraints on this tags ; both resources are supplied with a catalog. Similarly, an Active Schema and an Active Catalog are also validated by a schema, bound to classes described by the EXP module, and supplied by a catalog.

To summerize, Active Tags has repurposed and extended existing concepts such as catalogs for its own needs. A unified mechanism is used to retrieve both resources and the different components of the engine, such as an implementation of a module or the schema that will validate the active tags encountered in a user active sheet. The primal components of the engine are themselves following the same way, which makes Active Tags a self-defined system.

§ Don't break your streams

Streaming means processing data as it is being delivered. In this paper, filtering consist on traversing the input tree, and eventually altering the output. This section explains the strategy adopted for saving the memory with XPath-based filters when streaming, without imposing restrictions on the XPath syntax.

Evaluating XPath for streaming XML

XCL defines a subset of tags designed for filtering DOM or SAX inputs with XPath patterns. Some interesting attempts for using XPath while streaming have already been lead, such as [XSQ] and [STX] but usually a small subset of XPath is supported ; moreover, XSLT processors are still loading the entire stream in memory ("breaking the stream"), which cause a memory overflow for too big streams. To keep the benefit of streaming with SAX, the reference implementation of Active Tags -the RefleX engine- takes care of the memory usage. As XPath patterns à la XSLT are focusing essentially on the path to cross to reach a node of the tree, it is obvious that the hierarchy of the current branch must be maintained. Other nodes should be pruned.

Thus, if nothing more were needed, the data model would be reduced to a single branch of the tree, from the document root to the current node. In order to save memory usage, when the next node is read from the input source, the current node is pruned from the branch on behalf to the new one. Nodes are kept only for elements that have content, but removed when exiting the content. With this very simple basic approach, basic patterns like those below can be considered :

- `/` : the document root
- `a` : a node
- `a/b/c` : a relative strict hierarchy of nodes
- `a//b` : a relative hierarchy of nodes
- `/a/b/c` : an absolute strict hierarchy of nodes
- `/a//b` : an absolute hierarchy of nodes

Of course, other types/node tests can be considered : `text()`, `comment()`, any element `*`, any `node()`, any element within a namespace `prefix:*`, any `processing-instruction()`, and `processing-instruction('target')`. Attributes and namespace nodes are not involved on node tests because they are not encountered while traversing the input tree, but they might be present on predicates :

- `a[@b]`
- `a[not(@b)]`
- `a[@b='c']`
- `a[@b='c']/d[@e]`

...which doesn't require much more efforts because attributes are supplied by SAX when an element is encountered. Other predicates could be considered, for example, it would be very convenient to match an entry that satisfies :

- `/a/b/c[1]`
- `a/*[2]`
- `a/comment()[3]`
- `a/node()[position() < 4]`

...which denotes that the position of the node must be retrieved. However, the position of the node depends on the node test involved in the pattern. Some additional index counters must be stored on each node of the current branch, which is very acceptable regarding the memory usage. The following table shows which index has to be used according to the XML construct encountered and the node test involved :

Table 1: Kind of indexes stored by a parent node

XML construct	"natural" index	"type" index	"family" index	"name" index
`<p:element>`	`node()`	`*`	`p:*`	`element`
`some text`	`node()`	`text()`	N/A	N/A
`<!--comment-->`	`node()`	`comment()`	N/A	N/A

XML construct	"natural" index	"type" index	"family" index	"name" index
`<?target ?>`	`node()`	`processing-instruction()`	N/A	`processing-instruction('target')`

For example, if a comment `<!--comment-->` is read in the input source, the "type index" will be used if the node test involved in the pattern is `comment()`, but the "natural index" will be used if the node test is `node()`. The index of the document root is 1. Thus, each time a node is encountered, its parent node increments up to 4 counters. The engine stores counters with keys that reflect both the name and the type of the node considered. The counter bound to the "`node()`" key is incremented for each XML construct read. Other counters are incremented according to the type of the node.

Now, let's consider the following expressions:

- `/a/b/c[last()]`
- `a/*[count() > 3]`
- `a/node()[last()]`

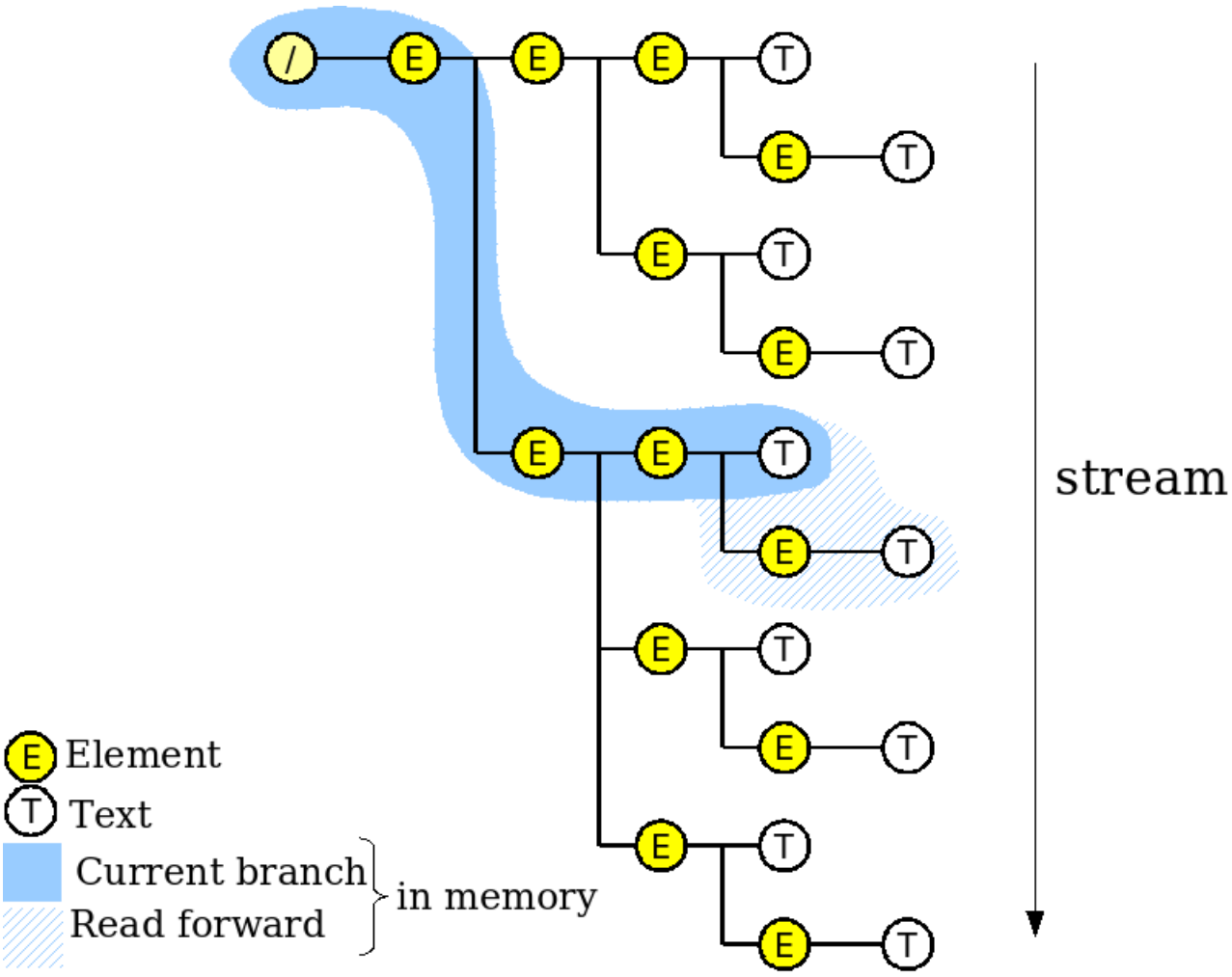
Unlike with the `position()` function, the `last()` and `count()` functions requires to know in advance the total number of nodes, which is not yet known. For the same reasons than previously, 4 values of size have to be considered : the "natural" size, the "type" size, the "family" size (namespace), and the "name" size. However, unlike previously, the values are kept unknown unless one of these sizes is expected explicitely or implicitely in a predicate. To compute the value, the engine will automatically read further input until the end of the parent element, without loosing the current node and its following siblings. A cache is necessary to achieve this. The engine just connects the nodes read in advance to the current one, so that a subtree is kept in memory instead of a single branch. Once the predicate becomes evaluable, the events are processed from the cached tree instead of the input, and the tree will be pruned as one goes along.

This strategy also works when forward and descendant axis are involved in the pattern, but it costs lots of memory. Users should be aware that using such patterns is low-efficient if almost all the tree have to be read in advance, and a strategy based on DOM instead of SAX would become more suitable. On the opposite, near from the bottom of a branch, the ability to use such patterns would be a great advantage

without loosing the benefit of streaming. Reading forward is performed automatically until a predicate becomes evaluable. Here are some other patterns that require reading forward:

- `a[following-sibling::b]`
- `a[b]`
- `a[*[not(self::b)]]`

Figure 3: Memory usage for XPath-based filtering with SAX



A single branch of the tree is stored in memory. Nodes are pruned as one goes along, but some node may be read in advance if an XPath expression is not yet evaluable.

Filtering in Active Tags

In Active Tags and RefleX, rule-based filters can be defined with the `<xcl:filter>` element ; within a filter definition, rules are described with the `<xcl:rule>` element. A filter can be defined alone in an external active sheet, and connected to a pipeline by another one :

```
<xcl:filter xmlns:xcl="http://ns.inria.org/active-tags/xcl">
    <xcl:rule pattern="/">
        <xcl:forward>
            <wrapper>
                <xcl:apply-rules/>
            </wrapper>
```

```
        </xcl:forward>
    </xcl:rule>
    <xcl:rule pattern="bar[@delete]"/>
    <xcl:rule pattern="foo[1]/bar[last()]">
        <xcl:forward>
            <newBar>
                <xcl:apply-rules/>
            </newBar>
        </xcl:forward>
    </xcl:rule>
</xcl:filter>
```

By default, any node that hasn't been matched by a pattern is forwarded to the next step of the pipeline. It is then easy to alter the input stream by only designing rules with patterns that will match specific nodes. A notifiable aspect is that a filter can be designed indifferently for a DOM input or a SAX input ; the output should differ only when side effects due to relaxing the tree while reading with SAX are occurring, of if deferred actions are occurring, which is not detailed in this paper. The following snippet code connects the previous filter definition to a pipeline where an [XInclude] filter is also used :

```
  <!--use the filter defined above-->
  <xcl:parse-filter name="myFilter" source="file:///path/to/filter.xcl"/>
  <!--use an XInclude filter (built-in)-->
  <xcl:parse-filter name="xinclude" source="http://www.w3.org/2001/XInclude"/>
  <!--connect a source to the XInclude filter, then to the custom filter and then to an output-->
  <xcl:parse name="input" source="file:///path/to/input.xml" style="stream"/>
  <xcl:filter filter="{ $xinclude }" name="included" source="{ $input }"/>
  <xcl:filter filter="{ $myFilter }" name="filtered" source="{ $included }"/>
  <xcl:transform output="file:///path/to/output.xml" source="{ $filtered }"/>
  <!--as this transformation doesn't involve an XSLT stylesheet, a copy is performed-->
```

XInclude is considered by the engine as a built-in filter. Other convenient filters are also defined for text processing : one that reads an input line by line, and another that filters it against a regular expression.

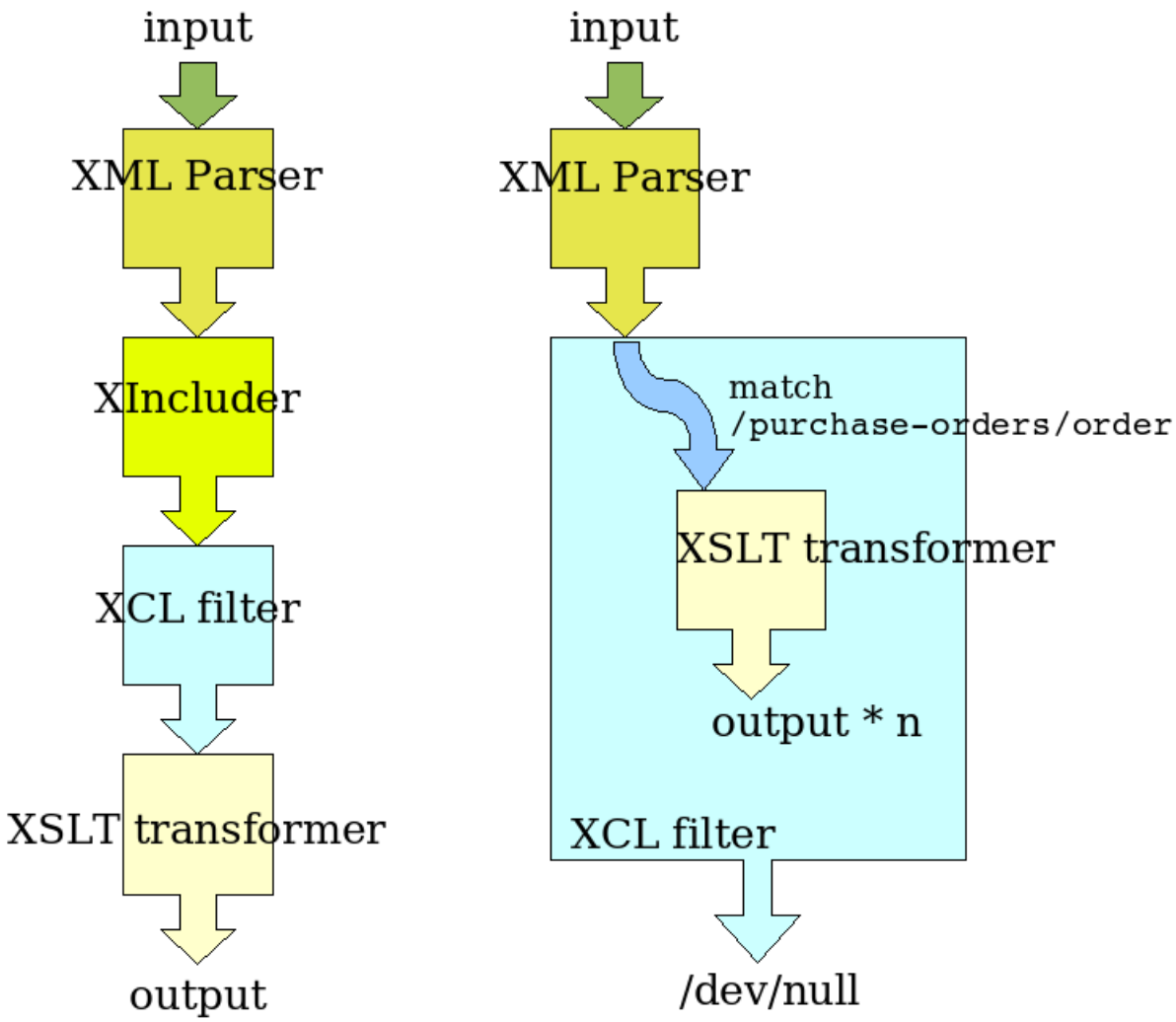
When XPath expressions may read backward, like those shown below, it is better to consider them locally, by "casting" a SAX fragment to a DOM subtree, which is straightforward in Active Tags :

```
    <xcl:rule pattern="someNode">
      <!--create a DOM document that contains all the subtree-->
      <xcl:document name="container" style="tree">
          <xcl:forward channel="container">
              <xcl:apply-rules/>
          </xcl:forward>
      </xcl:document>
      <!--forward some nodes from the DOM subtree-->
      <xcl:forward>
          <!--this expression could select a node backward-->{
          $container/there/following-sibling::foo[1]/previous-sibling::bar[1]
     }</xcl:forward>
  </xcl:rule>
```

Other scenarii where a stream have to be splitted in several output files can be considered : to achieve this, a channel is created and connected to the pipeline at runtime :

```
    <xcl:rule pattern="/purchase-orders/order">
        <!--create a SAX document for each order-->
        <xcl:document name="order" style="stream">
            <xcl:forward channel="order">
                <xcl:apply-rules/>
            </xcl:forward>
        </xcl:document>
        <!--save each "order" in a file-->
        <xcl:transform output="file:///path/to/purchase-orders/order-{ @id }.xml" source="{ $order }"/>
    </xcl:rule>
```

Figure 4: Pipelines



On the left, a simple pipeline ; on the right, the inputs that match the XPath pattern are redirected to a channel for serialization.

To summerize, the strategy adopted for streaming is to propose to the developper almost all the capabilities of XPath without syntaxic reductions. As the cooperation of SAX and DOM is also a good alternative in certain cases, Active Tags supplies convenient means to switch easily from one model to the other.

§ Conclusion

Active Tags offers filtering facilities with a higher API than those used by SAX, and without breaking the streams. This is a very valuable feature when building pipelines. Combining DOM and SAX is also an interesting strategy for the programmer that needs to go beyond the limits of XPath while streaming.

We also have seen that Active Tags goes really further than similar technologies and tools, by considering XML technologies in a global system. Each basic component of the system can rely on the other parts of the system. They are reusable, and are not overlapping each other since each is focusing on a single problematic : resource lookup, validation, implementation binding. Active Tags offers to XML language designers a new approach that facilitates both expressiveness and extensibility with the help of other tag libraries, even for designing declarative languages.

Thanks to the expressive power of XPath, which is also used to browse non-XML datas, the programmer can design complete applications with a very concise and efficient code. Thanks to its capabilities to mix imperative and declarative structures, Active Tags introduces an innovative XML pattern for which the Active Schema Language becomes a proof-of-concept.

Through various examples, we shown that the Active Tags system is a serious candidate for XML-oriented programming, thanks to its very sane and efficient approach unique in the XML world. It has been chosen by INRIA with the RefleX implementation on a project where the programs were made with active tags without any Java code. It is today deployed in a production environment.

Thanks to its innovative and efficient architecture and despite the intrinsic verbosity of XML, coding with Active Tags will appear much more shorter than with your favorite programming language !

Notes

1. Custom tag libraries, that is to say user-defined libraries exposed as tags in JSP
2. *XSLT and XQuery Processing*, **http://www.saxonica.com/**
3. *Java Platform, Enterprise Edition*, **http://java.sun.com/javaee/index.jsp**
4. By opposition to procedural : XML declarative languages won't be necessary operated sequentially since they specify "what" rather than "how to" ; a specialized processor of such a declarative language will process it according to the semantic of each tag. [W3C XML Schema], [XML Catalogs] and [SCXML] are such languages.
5. Some people refute to label stylesheets as "programs". This controversy is out of the scope of this document.
6. This will also prevent issues : automatic mappings on objects can cause loops when using descendant axes, as objects may be arranged in a graph, not as a hierarchy.

Bibliography

[Active Tags] *Active Tags technologies*, **http://ns.inria.org/active-tags/**

[Apache's Ant] *A Java-based build tool*, **http://ant.apache.org/**.

[Apache's Cocoon] *A web development framework*, **http://cocoon.apache.org/**.

[Apache's Jelly] *Executable XML*, **http://jakarta.apache.org/commons/jelly/**.

[DTLL] Jeni Tennison, *Datatype Library Language*, **http://www.jenitennison.com/datatypes**

[JSP] *JavaServer Pages Technology*, **http://java.sun.com/products/jsp/**

[JSTL] *JavaServer Pages Standard Tag Library*, **http://java.sun.com/products/jsp/jstl/**

[RefleX] *An Active Tags engine in Java*, **http://reflex.gforge.inria.fr/**

[RelaxNG] *Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG*, ISO/IEC FDIS 19757-2 **http://www.y12.doe.gov/sgml/sc34/document/0362_files/relaxng-is.pdf**

[Schematron] Rick Jelliffe, *A language for making assertions about patterns found in XML documents*, **http://www.schematron.com/spec.html**

[SCXML] Jim Barnett, *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, W3C Working Draft 21 February 2007, **http://www.w3.org/TR/scxml/**

[STX] *Streaming Transformations for XML (STX)* 1.0, Working Draft 1 July 2004, **http://stx.sourceforge.net/documents/spec-stx-20040701.html**.

[UEL] *Unified Expression Language*, **http://java.sun.com/products/jsp/reference/techart/unifiedEL.html**

[W3C XML Schema] Henry S. Thompson, *XML Schema Part 1: Structures Second Edition*, W3C Recommendation 28 October 2004 **http://www.w3.org/TR/xmlschema-1/**

[XCL] *XML Control Language*, specification **http://ns.inria.org/active-tags/xcl/xcl.html**

[XInclude] Jonathan Marsh, *XML Inclusions (XInclude) Version 1.0 (Second Edition)*, W3C Recommendation 15 November 2006 **http://www.w3.org/TR/xinclude/**

[XML 2006] *Active Tags : an XML system for native XML programming*, in panel "Next-Generation XML APIs", **http://2006.xmlconference.org/programme/presentations/156.html**

[XML Catalogs] *OASIS Standard V1.1, 7 October 2005*, **http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html**

[XProc] Norman Walsh, *An XML Pipeline Language*, W3C Working Draft 17 November 2006 **http://www.w3.org/TR/2006/WD-xproc-20061117/**

[XSLT] James Clark, *XSL Transformations (XSLT)* 1.0, W3C Recommendation 16 November 1999 **http://www.w3.org/TR/xslt**

[XSQ] Feng Peng, *XSQ: A streaming XPath engine*, ACM Transactions on Database Systems (TODS), **30**:2, 2005, **http://doi.acm.org/10.1145/1071610.1071617**.

The Author

Philippe Poulard
INRIA
philippe.poulard@sophia.inria.fr
http://reflex.gforge.inria.fr/
2004 route des lucioles - BP 93
FR-06902
Sophia Antipolis
France

Philippe Poulard is a software engineer at INRIA (french national institute for research in computer science and control) where he is involved in Web-oriented problematics. He has been specialized in XML technologies and e-documentation for 9 years. During this period, he has developed XML and SGML-based solutions and prototypes on behalf of the French Army and INRIA. More recently he has designed and implemented a set of XML technologies named "Active Tags" (http://ns.inria.org/active-tags/). He also teaches XML and Java at Nice/Sophia-Antipolis university and Aix/Marseille university. He has an engineer degree (M.Sc) from the Conservatoire National des Arts et Métiers.

Extreme Markup Languages 2007®
Montréal, Québec, August 7-10, 2007
This paper was formatted from XML source via XSL by Mulberry Technologies, Inc.