Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

ISO/IEC 19757-10 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 34, Document Description and Processing Languages.

ISO/IEC 19757 consists of the following parts, under the general title *Document Schema Definition Languages (DSDL)*:

— *Part 1: Overview*

— *Part 2: Regular-grammar-based validation — RELAX NG*

— *Part 3: Rule-based validation — Schematron*

— *Part 4: Namespace-based validation dispatching language — NVDL*

— *Part 5: Datatypes*

— *Part 6: Path-based integrity constraints*

— *Part 7: Character repertoire description language — CRDL*

— *Part 8: Document schema renaming language — DSRL*

— *Part 9: Datatype- and namespace-aware DTDs*

— *Part 10: Validation management*

Introduction

This International Standard defines a set of Document Schema Definition Languages (DSDL) that can be used to specify one or more validation processes performed against Extensible Markup Language (XML) or Standard Generalized Markup Language (SGML) documents. (XML is an application profile of SGML — ISO 8879:1986.)

A document model is an expression of the constraints to be placed on the structure and content of documents to be validated against the model and the information set that needs to be transmitted to subsequent processes. Since the development of Document Type Definitions (DTDs) as part of ISO 8879, a number of technologies have been developed through various formal and informal consortia notably by the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS). A number of validation technologies are standardized in DSDL to complement those already available as standards or from industry.

Historically, when many applications act on a single document, each application inefficiently duplicates the task of confirming that validation requirements have been met. Furthermore, such tasks and expressions have been developed and utilized in isolation, without consideration of how the features and functionality available in other technologies might enhance validation objectives.

The main objective of this International Standard is to bring together different validation-related tasks and expressions to form a single extensible framework that allows technologies to work in series or in parallel to produce a single or a set of validation results. The extensibility of DSDL accommodates validation technologies not yet designed or specified.

In the past, different design and use criteria have led users to choose different validation technologies for different portions of their information. Bringing together information within a single XML document sometimes prevents existing document models from being used to validate sections of data. By providing an integrated suite of constraint description languages that can be applied to different subsets of a single XML document, this International Standard allows different validation technologies to be integrated under a well-defined validation policy.

This International Standard has the following parts:

— *Part 1: Overview*

— *Part 2: Regular-grammar-based validation — RELAX NG*

— *Part 3: Rule-based validation* — Schematron

— *Part 4: Namespace-based validation dispatching language — NVDL*

— *Part 5: Datatypes*

— *Part 6: Path-based integrity constraints*

— *Part 7: Character repertoire description language — CRDL*

— *Part 8: Document schema renaming language — DSRL*

— *Part 9: Datatype and namespace-aware DTDs*

— *Part 10: Validation management*

Document Schema Definition Languages (DSDL) — Part 10: Validation Management

1 Scope

This International Standard specifies a suite of technologies that can be used to validate the structure and contents of structured documents marked up using ISO 8879 (SGML) and its derivatives (e.g. the W3C Extensible Markup Language, XML).

This International Standard defines a set of semantics for describing and ordering validation rules, a set of syntaxes for declaring validation rules, and a syntax for defining models for the management of validation sequences. It includes:

— Specifications of relevant validation technologies that can be used in isolation or within the DSDL framework.

— References to validation technologies defined outside of this International Standard that can be used within the DSDL framework.

— Semantics for managing the sequence in which different validation technologies are to be applied during the production of validation results.

This technical report illustrates how existing pipelining languages can be used to manage the validation processes.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 19757. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 19757 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

IETF RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*, Internet Standards Track Specification, August 1998, http://www.ietf.org/rfc/rfc2396.txt

SGML, *Standard Generalized Markup Language (SGML)*, ISO 8879:1986,

UCS, *Universal Multiple-Octet Coded Character Set (UCS)*, ISO/IEC 10646:2000,

W3C XML, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, 6 October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

W3C XML-Infoset, *XML Information Set*, W3C Recommendation, 24 October 2001, http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

W3C XML-Names, *Namespaces in XML*, W3C Recommendation, 14 January 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114/

W3C XPath, *XML Path Language (XPath) Version 1.0*, W3C Recommendation, 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

W3C XML Schema, *XML Schema*, W3C Recommendation, 24 October 2001, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

3 Terms and definitions

3.1 (document) instance

A structured document that is being validated with respect to a DSDL expression of document model constraints for structure and content

4 Using Pipelining for Validation Management

For the purpose of generating this first draft for a technical report on the use of existing pipeline techniques for validation management the following validation tests were proposed:

1. Use NVDL to split out the parts of the document that are encoded using HTML, SVG and MathML from the bulk of the document, whose tags are defined using a user-defined set of markup tags.

2. Validate the HTML elements and attributes using the HTML 4.0 DTD (W3C XML DTD).

3. Use a set of Schematron rules stored in `check-metadata.xml` to ensure that the metadata of the HTML elements defined using Dublin Core semantics conform to the information in the document about the document's title and subtitle, author, encoding type, etc.

4. Validate the SVG components of the file using the standard W3C schema provided in the SVG 1.2 specification.

5. Use the Schematron rules defined in `SVG-subset.xml` to ensure that the SVG file only uses those features of SVG that are valid for the particular SVG viewer available to the system

6. Validate the MathML components using the latest version of the MathML schema (defined using RELAX-NG) to ensure that all maths fragments are valid. The schema will make use the datatype definitions in `check-maths.xml` to validate the contents of specific elements.

7. Use `MathML-SVG.xslt` to transform the MathML segments to displayable SVG and replace each MathML fragment with its SVG equivalent.

8. Use the DSRL definitions in `convert-mynames.xml` to convert the tags in the local name set to the form that can be used to validate the remaining part of the document using `docbook.dtd`.

9. Use the CRDL rules defined in `mycharacter-checks.xml` to validate that the correct character sets have been used for text identified as being Greek and Cyrillic.

10. Convert the Docbook tags to HTML so that they can be displayed in a web browser using the `docbook-html.xslt` transformation rules.

Each validation script should allow the four streams produced by step 1 to be run in parallel without requiring the other validations to be carried out if there is an error in another stream. This means that steps 2 and 3 should be carried out in parallel to steps 4 and 5, and/or steps 6 and 7 and/or steps 8 and 9. After completion of step 10 the HTML (both streams), and SVG (both streams) should be recombined to produce a single stream that can fed to a web browser. The flow is illustrated in Figure 1.

4.1 Validation Management using DPML

The Declarative Process Markup Language (DPML[1]) is a very simple language for composing NetKernel[2] hosted services into processes: it is not explicitly an XML pipelining language, rather it is one among many language runtimes that can be used on NetKernel to build Unix-like applications.

NetKernel treats all software components as URI addressable services and uses a RESTful abstraction to invoke them. Higher level language runtimes (DPML, XRL, Java, Beanshell, Python, Groovy, Javascript) are used to compose services into 'pipelines'. DPML uses the term 'process' rather than 'pipeline' as the latter implies a linear synchronous flow whereas 'process' encompasses asynchronous forks, conditions and loops, etc.

The NetKernel Standard Edition provides a broad set of XML technologies as simple URI addressable services. Using the higher-level language runtimes the XML technologies can be composed into heterogeneous XML systems.

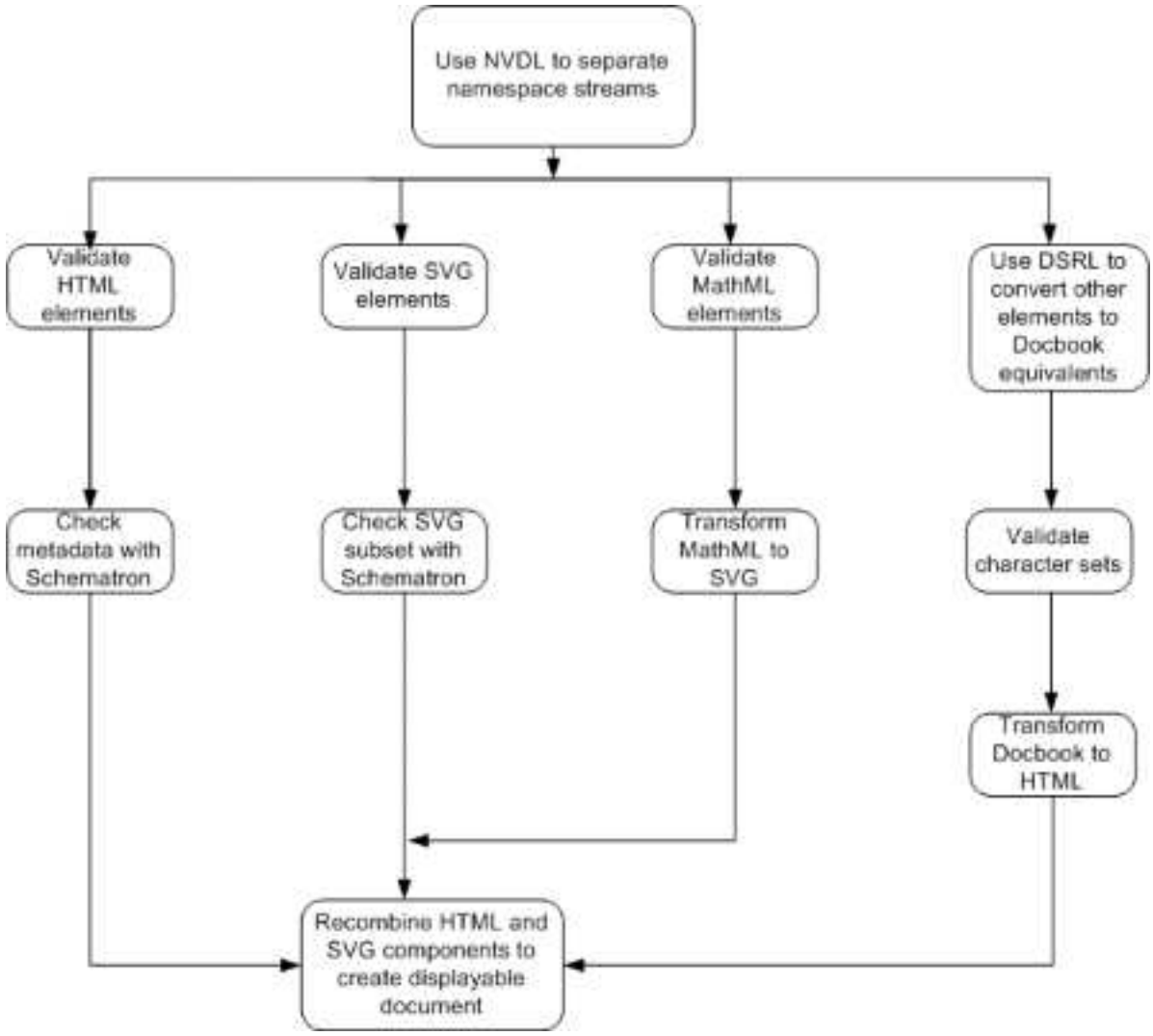

Figure 1: Pictorial Representation of Test Scenario

DPML is a syntax for constructing URI requests for services. The constructed requests are issued to the NetKernel infrastructure, which manages execution context, resource management, URI address-spaces, asynchronous execution, exceptions, etc.

4.1.1 Example of Linear DPML

To understand the DPML syntax you should know that a DPML instruction (`<instr>`) has two reserved tags:

— `<type>` is used to identify the base URI for the required service

— `<target>` is used to identify the resource which will receive the result of the service.

All other tag names are arbitrarily named arguments to be passed to the implementing service on 'active URI' request.

In DPML `var:` is used to identify transient variable resources managed by the stateful DPML runtime engine. The special URI `'this:response'` is the resource which is finally returned as the result of a process.

On NetKernel the arguments to a service are sourced with URIs of the form `this:param:xxxxx`. In our examples we assume an argument 'input' has been passed from the process that is requesting the DSDL pipeline: it is referenced with `'this:param:input'`.

If any request fails with an exception then the process flow will switch to the first <exception> block at the same level of cardinality as the instruction which threw the exception. The exception is accessed with the URI 'this:exception' and can be used as an XML resource in an error handling process or thrown to a calling process, etc. If no exception block is provided the exception will percolate up to the calling parent process.

In this use-case we are assuming that the 'nvdl' service has been written to return a multipart resource. We then use multipart fragmentation to select the required named parts later in the process.

```
<idoc>
<seq>
<comment>
**************
1. Use NVDL to split out the parts of the document that
are encoded using HTML, SVG and MathML from the bulk of
the document, whose tags are defined using a user-defined
set of markup tags.

Here we assume that the NVDL service will create a
multipart response containing the 'stripped-out'
resources 'html', 'svg', 'mathml' and 'other' - these
are later accessed by using multipart URI fragmentation.
**************
</comment>
<instr>
<type>nvdl</type>
<source>this:param:input</source>
<rules>nvdl-processing-rules.xml</rules>
<target>var:nvdl-streams</target>
</instr>
<comment>
**************
2. Validate the HTML elements and attributes using the
TML 4.0 DTD (W3C XML DTD).
**************
</comment>
<instr>
<type>validate-DTD</type>
<source>var:nvdl-streams#part(html)</source>
<schema>DTD-schema.dtd</schema>
<target>var:html-validated</target>
</instr>
<comment>
**************
3. Use a set of Schematron rules stored in
check-metadata.xml to ensure that the metadata
of the HTML elements defined using Dublin Core semantics
conform to the information in the document about the
document's title and subtitle, author, encoding type,
etc.
**************
</comment>
<instr>
<type>validate-Schematron</type>
<source>var:html-validated</source>
<schema>check-metadata.xml</schema>
<target>var:html-schematronized</target>
</instr>
<comment>
**************
4. Validate the SVG components of the file using the
```

```
standard W3C schema provided in the SVG 1.2
specification.
**************
</comment>
<instr>
<type>validate-XSD</type>
<source>var:nvdl-streams#part(svg)</source>
<schema>svg-1.2.xsd</schema>
<target>var:svg-validated</target>
</instr>
<comment>
**************
5. Use the Schematron rules defined in SVG-subset.xml to
ensure that the SVG file only uses those features of SVG
that are valid for the particular SVG viewer available
to the system.
**************
</comment>
<instr>
<type>validate-Schematron</type>
<source>var:svg-validated</source>
<schema>SVG-subset.xml</schema>
<target>var:svg-schamatronized</target>
</instr>
<comment>
**************
6. Validate the MathML components using the latest
version of the MathML. schema (defined in RELAX-NG) to
ensure that all maths fragments are valid. The schema
will make use the datatype definitions in
check-maths.xml to validate the contents of specific
elements.
**************
</comment>
<instr>
<type>validate-RelaxNG</type>
<source>var:nvdl-streams#part(mathml)</source>
<schema>mathmld-1.0.rng</schema>
<target>var:mathml-validated</target>
</instr>
<comment>
**************
7. Use MathML-SVG.xslt to transform the MathML segments
to displayable SVG and replace each MathML fragment with
its SVG equivalent.
**************
</comment>
<instr>
<type>transform-XSLT</type>
<source>var:mathml-validated</source>
<transform>MathML-SVG.xslt</transform>
<target>var:mathml-as-svg</target>
</instr>
<comment>
**************
8. Use the DSRL definitions in convert-mynames.xml to
convert the tags in the local nameset to the form that
can be used to validate the remaining part of the
document using docbook.dtd.
**************
```

```
</comment>
<instr>
<type>dsrl</type>
<source>var:nvdl-streams#part(other)</source>
<rules>convert-mynames.xml</rules>
<target>var:docbook</target>
</instr>
<instr>
<type>validate-DTD</type>
<source>var:docbook</source>
<schema>docbook-validation.dtd</schema>
<target>var:docbook-validated</target>
</instr>
<comment>
**************
9. Use the CRDL rules defined in mycharacter-checks.xml
to validate that the correct character sets have been
used for text identified as being Greek and Cyrillic.

Note here we target to the same variable like  x=f(x)
**************
</comment>
<instr>
<type>crdl</type>
<source>var:docbook-validated</source>
<crdl-rules>mycharacter-checks.xml</crdl-rules>
<target>var:docbook-validated</target>
</instr>
<comment>
**************
10. Convert the Docbook tags to HTML so that they can be
displayed in a web browser using the docbook-html.xslt
transformation rules.
**************
</comment>
<instr>
<type>transform-XSLT</type>
<source>var:docbook-validated</source>
<transform>docbook-html.xslt</transform>
<target>var:docbook-as-html</target>
</instr>
<comment>
**************
After completion of step 10 the HTML (both streams), and
SVG (both streams) should be recombined to produce a
single stream that can fed to a web browser.
**************
</comment>
<instr>
<type>transform-XSLT</type>
<source>var:docbook-as-html</source>
<transform>stylesheet-to-aggregate-everything.xslt</transform>
<html-param>var:html-schematronized</html-param>
<svg-param>var:svg-schematronized</svg-param>
<mathml-param>var:mathml-as-svg</mathml-param>
<target>this:response</target>
</instr>
<exception>
<comment>
**************
```

```
Here we are catching any exception in the
process - we are simply logging it and then
rethrowing it to the process which called us.
We could start the execution of an error
handling process here.
**************
</comment>
<instr>
<type>log</type>
<operand>this:exception</operand>
</instr>
<instr>
<type>throw</type>
<operand>this:exception</operand>
</instr>
</exception>
</seq>
</idoc>
```

4.1.2 Example of Asynchronous DPML

This example shows the same 'pipeline' as we showed in the previous sub-clause. In this sub-clause it is re-written using an asynchronous process pattern. Each sub-part of the use-case is executed asynchronously in forked sub-processes.

To illustrate how XML pipelining is generally not language specific we have written the sub-processes in different languages: it is frequently very valuable to be able to move back and forth between declarative and procedural approaches within the same XML process.

In the parent process we are asynchronously invoking sub-processes and passing an argument `'parameter'` to the child process. Each child process accesses the argument within their execution context with the URI `'this:param:parameter'`; just a named argument on the active URI.

```
<idoc>
<seq>
<comment>
**************
A. Fork an asynchronous DPML process to process the HTML
**************
</comment>
<instr>
<type>async</type>
<uri>active:dpml</uri>
<operand>html-process.idoc</operand>
<parameter>this:param:input</parameter>
<target>var:html-proc</target>
</instr>
<comment>
**************
B. Fork an asynchronous Beanshell (scripted Java)
process to process the SVG
**************
</comment>
<instr>
<type>async</type>
<uri>active:beanshell</uri>
<operand>svg-process.bsh</operand>
<parameter>this:param:input</parameter>
<target>var:svg-proc</target>
```

```
</instr>
<comment>
**************
C. Fork an asynchronous Python process to process the
MathML
**************
</comment>
<instr>
<type>async</type>
<uri>active:python</uri>
<operand>mathml-process.py</operand>
<parameter>this:param:input</parameter>
<target>var:mathml-proc</target>
</instr>
<comment>
**************
D. Continue with the Docbook processing in this
parent...
**************
</comment>
<comment>
**************
8. Use the DSRL definitions in convert-mynames.xml to
convert the tags in the local nameset to the form that
can be used to validate the remaining part of the
document using docbook.dtd.
**************
</comment>
<instr>
<type>nvdl</type>
<source>this:param:input</source>
<rules>nvdl-docbook-processing-rules.xml</rules>
<target>var:docbook</target>
</instr>
<instr>
<type>dsrl</type>
<source>var:docbook</source>
<rules>convert-mynames.xml</rules>
<target>var:docbook</target>
</instr>
<instr>
<type>validate-DTD</type>
<source>var:docbook</source>
<schema>docbook-validation.dtd</schema>
<target>var:docbook-validated</target>
</instr>
<comment>
**************
9. Use the CRDL rules defined in mycharacter-checks.xml
to validate that the correct character sets have been
used for text identified as being Greek and Cyrillic.
Note here we target to the same variable like  x=f(x)
**************
</comment>
<instr>
<type>crdl</type>
<source>var:docbook-validated</source>
<crdl-rules>mycharacter-checks.xml</crdl-rules>
<target>var:docbook-validated</target>
</instr>
```

```
<comment>
**************
10. Convert the Docbook tags to HTML so that they can be displayed
in a web browser using the docbook-html.xslt
transformation rules.
**************
</comment>
<instr>
<type>transform-XSLT</type>
<source>var:docbook-validated</source>
<transform>docbook-html.xslt</transform>
<target>var:docbook-as-html</target>
</instr>
<comment>
**************
E. Rejoin all asynchronous processes...
**************
</comment>
<instr>
<type>join</type>
<operand>var:html-proc</operand>
<target>var:html</target>
</instr>
<instr>
<type>join</type>
<operand>var:svg-proc</operand>
<target>var:svg</target>
</instr>
<instr>
<type>join</type>
<operand>var:mathml-proc</operand>
<target>var:mathml</target>
</instr>
<comment>
**************
After completion of step 10 the HTML (both streams), and
SVG (both streams) should be recombined to produce a
single stream that can fed to a web browser.
**************
</comment>
<instr>
<type>transform-XSLT</type>
<source>var:docbook</source>
<transform>
stylesheet-to-aggregate-everything.xslt
</transform>
<html-param>var:html</html-param>
<svg-param>var:svg</svg-param>
<mathml-param>var:mathml</mathml-param>
<target>this:response</target>
</instr>
<exception>
<comment>
**************
Here we are catching any exception in this
pipeline or any of the asynchronous child
pipelines - we are simply logging it and then
rethrowing it to the process which called this
pipeline. We could start the exectuion of an
error handling process here.
```

```
**************
</comment>
<instr>
<type>log</type>
<operand>this:exception</operand>
</instr>
<instr>
<type>throw</type>
<operand>this:exception</operand>
</instr>
</exception>
</seq>
</idoc>
```

svg-process.bsh : An SVG Pipeline Process in Beanshell (Scripted Java)

```
main()
{ //Execute NVDL to extract SVG
req=context.createSubRequest();
req.setURI("nvdl");
req.addArgument("source", "this:param:parameter");
req.addArgument("rules", "nvdl-svg-processing-rules.xml");
result=context.issueSubRequest(req);

//Validate with XML Schema
req=context.createSubRequest();
req.setURI("validate-XSD");
req.addArgument("source", result);
req.addArgument("schema", "svg-1.2.xsd");
result=context.issueSubRequest(req);

//Validate with Schematron
req=context.createSubRequest();
req.setURI("validate-Schematron");
req.addArgument("source", result);
req.addArgument("schema", "SVG-subset.xml");
result=context.issueSubRequest(req);

//Issue response
response=context.createResponseFrom(result);
context.setResponse(response);
}
```

mathml-process.py : A MathML Pipeline Process - in Python

```
#Execute NVDL to extract MathML
req=context.createSubRequest()
req.setURI("nvdl")
req.addArgument("source", "this:param:parameter")
req.addArgument("rules", "nvdl-mathml-processing-rules.xml")
result=context.issueSubRequest(req)

#Validate with Relax NG
req=context.createSubRequest()
req.setURI("validate-RelaxNG")
req.addArgument("source", result)
req.addArgument("schema", "mathml-1.0.rng")
```

```
result=context.issueSubRequest(req)

#Transform MathML to SVG
req=context.createSubRequest()
req.setURI("transform-XSLT")
req.addArgument("source", result)
req.addArgument("transform", "MathML-SVG.xslt")
result=context.issueSubRequest(req)

#Issue Response
response=context.createResponseFrom(result)
context.setResponse(response)
```

4.2 Validation Management using Cocoon

The Apache Cocoon Project provides mechanisms that can be used to control the flow of files through data pipelines. The basic idea behind Cocoon is to identify a set of actions that need to be taken with a file whose location and name match a specific pattern are requested from an Apache server.

A Cocoon workflow is defined in terms of:

— components

— views

— resources

— action sets

— pipelines

— flows containing scripts.

Cocoon allows users to:

— Generate trees from existing files, databases, programs, etc.

— Match specific file naming patterns, or specific fragments of a file.

— Transform trees by applying XSLT transformations.

— Aggregate data from multiple trees.

— Serialize trees as XML, FOP, HTML, XHTML files, or any other form of output for which a serializer has been defined.

— Identify exceptions and define error-handling processes.

Documents to be processed using Cocoon can be wrapped in a `xsp:page` element to create an Extensible Server Page (XSP). Configuration of Cocoon processes takes place within a "sitemap" (`xsp:sitemap`) that acts as a wrapper for a set of component, view and pipelines elements. Multiple pipeline elements and match rules can be defined within the pipelines element. Matching, which are done against URIs, by can be defined using regular expressions and/or wildcards.

A typical pipeline might include the following:

```
 <map:pipeline>
  <map:match pattern="docs/*.html">
```

```
    <map:generate src="docs/*.xml">
    <map:transform src="stylesheet/xml2html.xsl">
    <map:serialize type="html">
  </map:match>
 </map:pipeline>
```

The fact that Cocoon pipelines can include calls to functions or scripts makes it unsuitable for declarative control of validation management, which is a goal of DSDL. It is possible, however, to envisage adopting the declarative markup used to define pipelines within Cocoon as the basis for a subset of Cocoon functionality that could be used for validation management within DSDL. For example, trees of validated elements could be created using an extension to the `map:generate` option, e.g.:

```
<map:generate type="nvdl" nvdl-rules="rules.nvdl" src="sample.doc"/>
```

The following example suggests how the DSDL multitrack scenario might be coded using an extended subset of Cocoon commands

```
<map:piplines name="multitrack-validation-example">
 <map:pipeline name="recombine-multiple-tracks">
  <map:pipeline name="check-html">
    <map:generate type="nvdl" rules="test.nvdl" mode="html"
     src="{1)"/>
    <map:transform type="schematron" src="test-html.sch"/>
  </map:pipeline>
  <map:pipeline name="check-svg">
    <map:generate type="nvdl" rules="test.nvdl" mode="svg"
     src="{1)"/>
    <map:transform type="schematron" src="test-svg.sch"/>
  </map:pipeline>
  <map:pipeline name="check-html">
    <map:generate type="mathml" rules="test.nvdl" mode="mathml"
     src="{1)"/>
    <map:transform type="xslt" src="mathml2svg.xsl"/>
  </map:pipeline>
  <map:pipeline name="check-rest">
    <map:generate type="nvdl" rules="test.nvdl" mode="xml"
     src="{1)"/>
    <map:transform type="dsrl" src="rename.dsrl"/>
    <map:generate type="crdl" rules="test-chars.crdl"/>
    <map:transform type="xslt" src="docbook2HTML.xslt"/>
  </map:pipeline>
  <map:serialize type="xhtml"/>
 <map:pipeline>
</map:pipelines>
```

It should be noted that the use of the outermost pipeline element to recombine the multiple fragments is only conjecture at this point. It is unclear how the serialized would be able to determine where in the original NVDL input stream each transformed set of data is to be positioned as NVDL does not uniquely identify fragments.

4.3 Validation Management using XPL

The following example of the use of XPL for validation management has been submitted by the developers of the language:

```
 <p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
           xmlns:oxf="http://www.orbeon.com/oxf/processors">

    <p:param name="source-document" type="input"/>
    <p:param name="result-document" type="output"/>
```

```
  <!--
  First pass at writing a pipeline to implement the DSDL Test Scenario.

  Questions / issues:

  o = questions by Erik Bruchez
  v = answers by Eric van der Vlist

 o What is expected of the output of validators? Is the flow supposed to be
   interrupted when a validation error occurs?

(v) Both questions are controversial :-) ...
An overall principle for DSDL is that DSDL is only about validation
and do not carry any kind of PSVI information. Following this principle,
the result of a DSDL validation should be "valid" or "invalid".

Now, this scenario seems to prove that this might not be the case for
Part 10 (Validation Management) and this is also why I am now thinking
that XPL may be interesting while if that was only about "valid"/"invalid"
XPL wouldn't have been such a good fit IMO.

My answer to your first question seems thus to be "a validation report
containing at least a "yes/no" answer plus adhoc content.

My personal answer to the second question would be "that depends".
On the XMLfr publication process for instance, I have two kind of validations:
a RNG schema that returns errors and a Schematron that validate good practises
and returns warnings. If the first one could interrupt the flow,
the second one shouldn't do it.

If we wanted to bring that notion in Validation Management, that could
mean that instead of "yes/no"; we could have an error level and that when
invoking a validator we could define the level of the error to raise
in case the validation fails.

The default behaviour could be to stop at the first error (as you've
implied in the pipeline), but an optional "config" input could be added
that would allow to specify an error level. If the
error level is positive, no exception would be raised.

Now, a validator might have several outputs. What about defining
several outputs for a validator:

    - the data output (useful only for schema languages that, like DTDs
      or WXS augment the infoset with stuff such as default values).
    - the report output with a yes/no (or level) information, error
      messages or (for Schematron) the validation report.
    - to that, we could add a PSVI output in the case of W3C XML Schema
     (assuming we had an XML format for the PSVI).

When a validator would be configured with a positive error level,
error detection could be done by checking the report output.

All these answers are personal and should be checked with DSDL Working group.

o For simplicity, I assumed that the NVDL processor here would produce outputs
  with those particular names. This would be possible only if the NVDL processor
  could be configured to map those output names to namespaces.
  Practically, this processor could either:
```

```
   o Have predefined output names, like document-1, document-2, etc.
   o Produce a single XML document with all the streams aggregated

   I do not know NVDL well enough to see what would be natural here.

  (v) None of them are natural :-) ...

      Right now, NVDL is currently for validation only and takes care of
      invoking the different validators to return a single "yes/no" answer.

      Using it to split a document like mentioned in that scenario is thus
      an extrapolation of what does the current NVDL implementation.

      However, given the fact that NVDL splits documents according to their
      namespaces, I wonder if the aggregating streams would be different enough
      from the original document ;-) ...

      Thus, I am wondering if predefined names wouldn't be the best solution.
      Maybe, instead of using document-i, we could map namespaces URIs on names
      (like they are mapped to namespaces prefixes).

o I have used a single validation processor that supports W3C XML Schema, Relax NG,
  Schematron, and DTDs (here DTDs would either have to be encaspulated into a root
  element, or referred externally). You could of course propose one processor per
  schema type. The PresentationServer validation processor currently supports
  transparently W3C Schema and  Relax NG.

o I proposed using XSLT to recombine the final document in the end.

o Otherwise, the pipeline is very simple. Nothing is against parallel execution on XPL.
  Without exception support, the processing would just stop if there is a validation
  error. With exception support, it could resume, locally (per branch) if needed, or
  just propose a global fallback. Everything that is possible with exceptions.

   -->

   <!--
1. Use NVDL to split out the parts of the document that are encoded using HTML, SVG and
   MathML from the bulk of the document, whose tags are defined using a user-defined
   set of markup tags.
    -->
   <p:processor name="oxf:nvdl">
       <p:input name="document" href="#source-document"/>
       <p:input name="rules">
           <rules>
               NVDL rules
           </rules>
       </p:input>
       <p:output name="html-stream" id="html-stream"/>
       <p:output name="svg-stream" id="html-stream"/>
       <p:output name="mathml-stream" id="html-stream"/>
    <!--

       (v) typo: the ids should be "svg-stream" &amp;amp; "mathml-stream"...

       -->
       <p:output name="other-stream" id="other-stream"/>
   </p:processor>

   <!--
```

```
2. Validate the HTML elements and attributes using the HTML 4.0 DTD (W3C XML DTD).
   -->
   <p:processor name="oxf:validation">
       <p:input name="data" href="#html-stream"/>
       <p:input name="schema">
           <!-- Reference to DTD for HTML -->
           <dtd href="..."/>
       </p:input>
       <p:output name="data" id="html-stream-validated"/>
   </p:processor>

   <!--
3. Use a set of Schematron rules stored in check-metadata.xml to ensure that the metadata
   of the HTML elements defined using Dublin Core semantics conform to the information in
   document about the document's title and subtitle, author, encoding type, etc.
   -->
   <p:processor name="oxf:validation">
       <p:input name="data" href="#html-stream-validated"/>
       <!-- Reference to Schematron schema for HTML metadata -->
       <p:input name="schema" href="check-metadata.xml"/>
       <p:output name="data" id="html-stream-schematronized"/>
      <!--

(v) Note that in the case of Schematron, the data output is identical to the data input.

       -->
   </p:processor>

   <!--
4. Validate the SVG components of the file using the standard W3C schema provided in the
   SVG 1.2 specification.
   -->
   <p:processor name="oxf:validation">
       <p:input name="data" href="#svg-stream"/>
       <!-- Reference to W3C Schema for SVG -->
       <p:input name="schema" href="svg-1.2.xsd"/>
       <p:output name="data" id="svg-stream-validated"/>
   </p:processor>

   <!--
5. Use the Schematron rules defined in SVG-subset.xml to ensure that the SVG file only uses
   those features of SVG that are valid for the particular SVG viewer available to the sys
   -->
   <p:processor name="oxf:validation">
       <p:input name="data" href="#svg-stream-validated"/>
       <!-- Reference to Schematron schema for SVG subset -->
       <p:input name="schema" href="SVG-subset.xml"/>
       <p:output name="data" id="svg-stream-schmatronized"/>
   </p:processor>

   <!--
6. Validate the MathML components using the latest version of the MathML. schema (defined
   in RELAX-NG) to ensure that all maths fragments are valid. The schema will make use the
   datatype definitions in check-maths.xml to validate the contents of specific elements.
   -->
   <p:processor name="oxf:validation">
       <p:input name="data" href="#mathml-stream"/>
       <!-- Reference to Relax NG shema for MathML -->
       <p:input name="schema" href="mathml-1.0.rng"/>
       <p:output name="data" id="mathml-stream-validated"/>
```

```
    </p:processor>

    <!--
 7. Use MathML-SVG.xslt to transform the MathML segments to displayable SVG and replace each
    MathML fragment with its SVG equivalent.
    -->
    <p:processor name="oxf:xslt">
        <p:input name="data" href="#mathml-stream-validated"/>
        <p:input name="config" href="MathML-SVG.xslt"/>
        <p:output name="data" id="mathml-as-svg"/>
    </p:processor>

    <!--
 8. Use the DSRL definitions in convert-mynames.xml to convert the tags in the local nameset
    to the form that can be used to validate the remaining part of the document using
    docbook.dtd.
    -->
    <p:processor name="oxf:dsrl">
        <p:input name="data" href="#other-stream"/>
        <p:input name="config" href="convert-mynames.xml "/>
        <p:output name="data" id="docbook-stream"/>
    </p:processor>

    <p:processor name="oxf:validation">
        <p:input name="data" href="#docbook-stream"/>
        <!-- Reference to DTD Docbook -->
        <p:input name="schema">
            <dtd href="..."/><!-- Reference to W3C DTD -->
        </p:input>
        <p:output name="data" id="docbook-stream-validated"/>
    </p:processor>

    <!--
 9. Use the CRDL rules defined in mycharacter-checks.xml to validate that the correct
    character sets have been used for text identified as being Greek and Cyrillic.
    -->
    <p:processor name="oxf:crdl">
        <p:input name="data" href="#docbook-stream-validated"/>
        <p:input name="config" href="mycharacter-checks.xml "/>
        <p:output name="data" id="docbook-stream-validated-2"/>
    </p:processor>

    <!--
10. Convert the Docbook tags to HTML so that they can be displayed in a web browser using
    the docbook-html.xslt transformation rules.
    -->
    <p:processor name="oxf:xslt">
        <p:input name="data" href="#docbook-stream-validated-2"/>
        <p:input name="config" href="docbook-html.xslt"/>
        <p:output name="data" id="docbook-as-html"/>
    </p:processor>

    <!--
    After completion of step 10 the HTML (both streams), and SVG (both streams) should be
    recombined to produce a single stream that can fed to a web browser.
    -->
    <p:processor name="oxf:xslt">
        <p:input name="data" href="#html-stream-schematronized"/>
        <p:input name="html-2" href="#docbook-as-html"/>
        <p:input name="svg-1" href="#svg-stream-schmatronized"/>
```

```
        <p:input name="svg-2" href="#mathml-as-svg"/>
        <p:input name="config" href="stylesheet-to-aggregate-everything.xsl"/>
        <p:output name="data" ref="result-document"/>
    </p:processor>

</p:config>
```

Bibliography

[1] *DPML Quick Reference Guide*,
http://www.1060research-server-1.co.uk/docs/2.0.2/book/declarative/doc_guide_dpml_quick_reference.html

[2] *NetKernel Service Oriented Microkernel XML Application Server*,
http://www.1060research.com/netkernel/index.html