

# Contents

Page

Foreword.....	iv
Introduction.....	v
1 Scope.....	1
2 Normative references.....	1
3 Terms and definitions.....	2
4 Using Pipelining for Validation Management.....	3
4.1 Validation Management using DPML.....	4
4.1.1 Example of Linear DPML.....	4
4.1.2 Example of Asynchronous DPML.....	8
4.2 Validation Management using Cocoon.....	12
4.3 Validation Management using XPL.....	14
5 Managing Multiple Input Sources.....	17
6 Validating Included Fragments.....	17
7 Assigning Default Values.....	17
8 Validating Document Fragments.....	18
9 Managing Encrypted Fragments.....	18
Annex A (informative) Example.....	19
Bibliography.....	20

## 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 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national standards bodies for voting. Publication as an International Standard requires approval of at least 75% of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements in this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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

## 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 methodology 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 validation processes, including:

- processing of multiple input sources
- creation of new document instances that are validated against external document type definitions (DTDs) and schemas
- adding default values from DTDs and schemas to validated document instances
- using an `xml:base` attribute to resolve local URIs
- validation of document fragments
- dereferencing XInclude elements
- decrypting encrypted XML elements.

## 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 HTML, *HTML 4.01 Specification*, W3C Recommendation, 24 December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>

W3C MathML, *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*, W3C Recommendation, 21 October 2003, <http://www.w3.org/TR/2003/REC-MathML2-20031021/>

W3C SVG, *Scalable Vector Graphics (SVG) 1.1 Specification*, W3C Recommendation, 14 January 2003, <http://www.w3.org/TR/2003/REC-SVG11-20030114/>

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 CRDL

The Character Repertoire Description Language defined in Part 7 of this standard.

#### 3.2 Docbook

OASIS specification for marking-up books.

#### 3.3 DPML

Declarative Process Markup Language as defined for XML.

#### 3.4 DSRL

The Document Schema Renaming Language defined in Part 8 of this standard.

#### 3.5 DTD

Document Type Definition. There are two subtypes of DTD: SGML DTDs are defined in ISO 8879; XML DTDs are defined using a subset of SGML DTD standard that is defined in the W3C XML specification.

#### 3.6 HTML

The World Wide Web Consortium's HyperText Markup Language.

#### 3.7 (document) instance

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

#### 3.8 NVDL

The Namespace-based Validation Dispatching Language defined in Part 4 of this standard.

### 3.9 pipeline

A command that takes its input from the output of another command, or whose output is used as the input to a subsequent command.

### 3.10 RELAX-NG

The regular-grammar-based validation language defined in Part 2 of this standard.

### 3.11 SGML

The Standard Generalized Markup Language defined in ISO 8879.

### 3.12 URI

Uniform Resource Identifier as defined in RFC 2396.

### 3.13 validation management

A set of declarative commands that control the sequence in which multiple data structure and/or content validation functions are invoked.

### 3.14 validation test

A test undertaken to confirm the validity of the contents of one or more elements in an XML data stream.

### 3.15 XML

Extensible Markup Language.

## 4 Using Pipelining for Validation Management

To demonstrate the application of techniques described within this technical report for validation management, the following set of validation tests have been used to create examples of pipelines:

- Use NVDL to split out the parts of the document that are encoded using W3C's HTML, SVG and MathML specifications from the bulk of the document, whose tags are defined using a user-defined set of markup tags.
- Validate the HTML elements and attributes using the W3C HTML 4.01 DTD (W3C XML DTD).
- 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.
- Validate the SVG components of the file using the W3C XML Schema provided in the W3C SVG specification.
- 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.
- Validate the MathML components using the MathML schema (defined using RELAX-NG) to ensure that all maths fragments are valid. The schema makes use the datatype definitions in `check-maths.xml` to validate the contents of specific elements.
- Use `MathML-SVG.xslt` to transform the MathML segments to displayable SVG and replace each MathML fragment with its SVG equivalent.
- 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`.

- 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.
- 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) need to be recombined to produce a single stream that can be fed to a web browser. The flow is illustrated in Figure 1.

### 4.1 Validation Management using DPML

The Declarative Process Markup Language (DPML<sup>[1]</sup>) is a simple language that is used to compose NetKernel<sup>[2]</sup> hosted services: it is not explicitly an XML pipelining language, but is one, among many, of a set of runtime languages that can be used within 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 languages (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, loops, etc.

DPML provides a syntax for ordering 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.

The NetKernel Standard Edition software suite provides a set of XML technologies that can be invoked as simple URI addressable services. Using the higher-level languages activated by NetKernel different XML technologies can be composed into heterogeneous XML systems.

#### 4.1.1 Example of Linear DPML

In the DPML syntax a DPML instruction (an XML `<instr>` element) has two reserved sub-elements:

- `<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 'activate URI' request.

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

In NetKernel the arguments to a service are sourced with URIs of the form `this:param:xxxxx`. The argument `input` is passed from the process that is requesting the DSDL pipeline: it is referenced by `this:param:input`.

If a request fails with an exception the process flow will switch to the first `<exception>` block at the same level 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 at the current level in the processing tree the exception will percolate up to the calling parent process.

In the example shown below the 'nvdI' service has been written to return a multipart resource. Multipart fragmentation is used to select the required named parts later in the process.

```
<idoc>  
<seq>
```

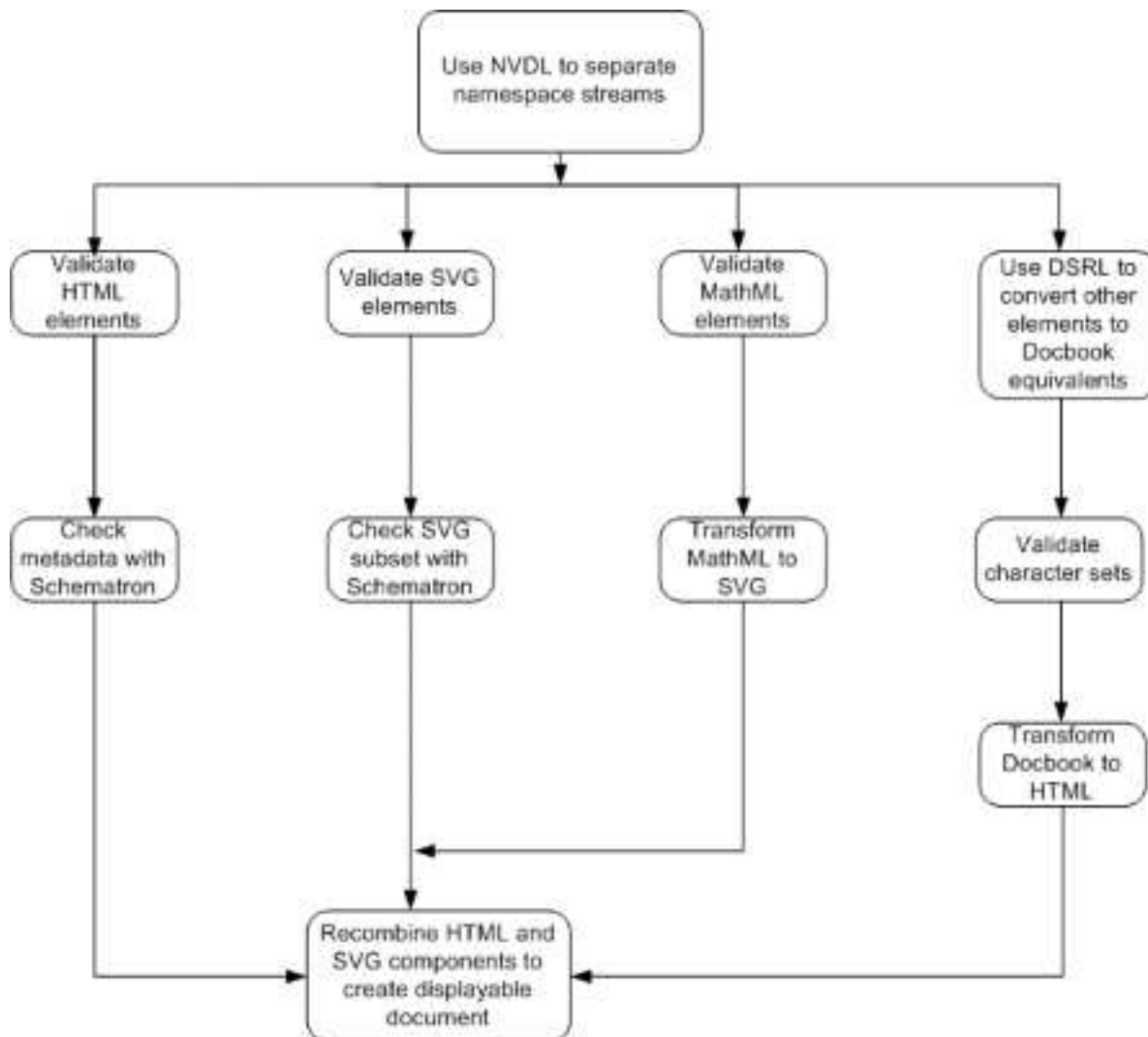


Figure 1: Pictorial Representation of Example Validation Management Sequence

```

<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.
  
```

The NVDL process creates a multipart response containing 'stripped-out' resources identified using 'html', 'svg', 'mathml' and 'other'. These streams are later accessed through multipart URIs.

```

*****
</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>
*****
Any exception in the process will be trapped by the
following code, which simply loggs the cause before
rethrowing it to the process which started the validation.
*****
</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

The following example shows how the 'pipeline' shown above could be re-written as an asynchronous process pattern. Each sub-part of the use-case is executed asynchronously in forked sub-processes.

To illustrate how XML pipelining can be handled in a way that is not language specific this example has been written sot that the different sub-processes are processed using different languages. In practice it is frequently 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 using the URI 'this:param:parameter'.

```

<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>
<instr>
<type>nvdl</type>
<source>this:param:input</source>
<rules>nvdl-docbook-processing-rules.xml</rules>
<target>var:docbook</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: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 be 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>
*****
This example catches any exception in a pipeline,
or any of the asynchronous child pipelines.
As before, the application is simply logging the
exception and then rethrowing it to the process
which called the pipeline.
*****
</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("nvd1");
req.addArgument("source", "this:param:parameter");
req.addArgument("rules", "nvd1-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("nvd1")
req.addArgument("source", "this:param:parameter")
req.addArgument("rules", "nvd1-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 when a file whose location and name match a specific pattern is 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 `map:component`, `map:view` and `map:pipelines` elements. Multiple `map:pipeline` elements and `map:match` rules can be defined within the `map:pipelines` element. Matching, which is 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, which would invoke a DSDL validation process to produce one or more in-memory document trees, e.g.:

```
<map:generate type="nvd1" nvd1-rules="rules.nvd1" src="sample.doc"/>
```

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

**Can Cocoon resynchronize tracks? Cocoon is based on the processing of files. Can it be used effectively with fragmented files stored as DOM trees that can be fully resynchronized after validation of data in different namespaces?**

```
<map:pipelines name="multitrack-validation-example">
  <map:pipeline name="recombine-multiple-tracks">
    <map:pipeline name="check-html">
      <map:generate type="nvd1" rules="test.nvd1" mode="html"
        src="{1}"/>
      <map:transform type="schematron" src="test-html.sch"/>
    </map:pipeline>
    <map:pipeline name="check-svg">
      <map:generate type="nvd1" rules="test.nvd1" 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.nvd1" mode="mathml"
        src="{1}"/>
      <map:transform type="xslt" src="mathml2svg.xsl"/>
    </map:pipeline>
    <map:pipeline name="check-rest">
      <map:generate type="nvd1" rules="test.nvd1" 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 `map:pipeline` element to recombine the multiple fragments is only conjecture. It is unclear, at present, how the serialized tree 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

An XML Pipelining Language (XPL<sup>[3]</sup>) defines orchestrated sequences of operations on XML Information Sets (Infosets). Individual operations are encapsulated within components called XML processors. Operations involve the production, consumption, and transformation of XML Infosets. An XPL program can support unconditional operations, conditions, loops and change of control following runtime errors.

An XPL program consists of:

- *Input and output parameters.* Each input or output has a name, and may either provide the XPL program with an XML Infoset (as an input), or produces an XML Infoset (as an output). An XPL sequence can, when used as an adjunct to other processes, have no specific input or output.
- *A sequence of statements.*

An XPL statement is an element with the following characteristics:

- *Scoped XML Infoset Identifiers.* A set of XML Infoset identifiers in scope at the point where the statement occurs in the XPL program. The set may be empty.
- *Exposed XML Infoset Identifiers.* A set of XML Infoset identifiers made available (exposed) by the statement. There cannot be an intersection between the scoped XML Infoset identifiers and the exposed XML Infoset identifiers. For example, a statement can attempt to expose an XML Infoset identifier already present in the set of scoped XML Infoset identifiers. Such a condition must raise a static error. This is referred to as the 'no-collision' rule.

The set of XML Infoset identifiers in scope for a statement within an XPL sequence, unless specified otherwise, consists of the union of the identifiers in scope for the previous statement and the identifiers exposed by the previous statement in document order.

If there is no such previous statement, the set of XML Infoset identifiers in scope for a statement of an XPL program, unless specified otherwise, consists of the set of identifiers specified by the `infoset` attributes of the XPL statements. In other words, this condition applies to the first statement of an XPL program.

**Is the above statement correct, or should "infoset attributes" read "param elements"?**

The following example suggests how the DSDL multitrack scenario might be coded using XPL:

```

<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"/>

  <!--

```

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="svg-stream"/>
  <p:output name="mathml-stream" id="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"/>
<!--
```

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 use 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>
```

## 5 Managing Multiple Input Sources

When transforming documents prior to validation it is sometimes necessary to combine data from more than one source. Where data stream changes are not indicated by the use of an XInclude declaration, the files to be combined will need to be identified as part of the validation management process.

How can validation management identify the order in which documents will be processed? Would we need to define extensions to existing pipelining languages to provide this functionality?

To be completed

## 6 Validating Included Fragments

The W3C XInclude language provides a widely adopted means of integrating data from different resources into a single, unified, data model. In some cases the included modules will need to be validated separately, because they contain elements in a different namespaces. In other cases the included elements will be in the same namespace as the document they are included within, so the document cannot be validated until after the inclusion has been resolved. An XML validation management suite must provide a mechanism that allows the way in which validation is to be performed to be determined from the namespace of the root element of the inclusion.

What happens when an included element itself contains inclusions? Do we need a separate table per incusion (perhaps using nested elements within a declaration) or should there be a single list that provides instructions for all inclusions found under an initial root element?

Would we need to define extensions to existing pipelining languages to provide this functionality?

To be completed

## 7 Assigning Default Values

Validation is often dependent on the inclusion of default values for attributes or elements that are derived from a schema or DTD. In the case of SGML-encoded documents omitted markup may need to be added by parsing the document against a DTD prior to undertaking other aspects of validation management. To ensure that a data stream is complete before validation it may be necessary to require pre-processing to add default data and markup.

Should adding default data/markup be considered a pre-processing stage?

Would we need to define extensions to existing pipelining languages to provide this functionality?

To be completed

## 8 Validating Document Fragments

In some cases only parts of a document need to be validated. In this case the validation request needs to include a fragment identifier that identifies the element whose contents are to be validated.

Should it be possible to reference to a fragment in another document?

To be completed

## 9 Managing Encrypted Fragments

Where documents have been encrypted for safer interchange it is necessary that they are unencrypted prior to validation. For this to be possible the validation manager needs to be provided with the keys needed to unencrypt the source file.

To be completed

## **Annex A** (informative)

### **Example**

The following example shows how the techniques described in this Technical Report can be used when processing a sample file.

To be completed

## Bibliography

- [1] *D P M L            Q u i c k            R e f e r e n c e            G u i d e* ,  
[http://www.1060research-server-1.co.uk/docs/2.0.2/book/declarative/doc\\_guide\\_dpml\\_quick\\_reference.html](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>
- [3] *XML Pipeline Language (XPL) Version 1.0*, <http://www.orbeon.com/ops/2005/xpl/>

## Summary of editorial comments:

### [4.2] Validation Management using Cocoon

Can Cocoon resynchronize tracks? Cocoon is based on the processing of files. Can it be used effectively with fragmented files stored as DOM trees that can be fully resynchronized after validation of data in different namespaces?

### [4.3] Validation Management using XPL

Is the above statement correct, or should "infoset attributes" read "param elements"?

### [5] Managing Multiple Input Sources

How can validation management identify the order in which documents will be processed? Would we need to define extensions to existing pipelining languages to provide this functionality?

### [6] Validating Included Fragments

What happens when an included element itself contains inclusions? Do we need a separate table per inclusion (perhaps using nested elements within a declaration) or should there be a single list that provides instructions for all inclusions found under an initial root element?

Would we need to define extensions to existing pipelining languages to provide this functionality?

### [7] Assigning Default Values

Should adding default data/markup be considered a pre-processing stage?

Would we need to define extensions to existing pipelining languages to provide this functionality?

### [8] Validating Document Fragments

Should it be possible to reference to a fragment in another document?