June 21, 2002

Schemachine

(C) 2002 Rick Jelliffe

A framework for modular validation of XML documents

This note specifies a possible framework for supporting modular XML validation. It has **no official status whatsover.**[1] It is for discussion purposes only. Review comments are welcome to ricko@topologi.com

The strawman has the following features:

- based on XML Pipeline structures (http://www.w3.org/TR/xml-pipeline/), but with rearrangement and renaming,
- embedded in Schematron-like superstructure with titles and phases,
- a minimal implementation is possible, where all validators and translators are command-line executable programs, and the framework document is translated into BAT files or Bourne shell scripts (i.e., validators etc. are treated as black boxes) ,
- the purpose is validation rather than declarative description per se. (In particular, the further down a transformation chain that data gets, the more difficult it will be to tie the effect of a schema to the original document.)
- this framework supports both validation of explicit structure and validation of complex data values. It leaves issues of simple datatyping to particular validators,
- validation is a tree of processes,
- supports inband signalling (@exclude) and out-of-band signalling (@haltOnFail).

The element names used here are very process-oriented. They could be renamed to be more declarative: for example <validate> to <constraint> and <pass> to <seq>

1. This has been developed as a strawman for the ISO DSDL effort. For another strawman using a different basis, see Eric van der Vlist's **Xml Validation Interoperability Framework (xvif),** at http://downloads.xmlschemata.org/python/xvif/xvif.html

Basic Structure

A Framework document has four parts:

- optional **head** section,
- optional **phases** section,
- required **pipelines** section, and
- optional **schemas** section.

A simple example is:

```
<schemachine xmlns=”....”>
   <title>Example Schema</title>
   <pass>
      <validate engine=”schemachine:xsd” />
      <validate engine=”schemachine:schematron”>
         <param name=”schema” href=”a Schematron schema”/>
      </validate>
   </pass>
</schemachine>
```

This example specifies that that we will validate using an XSD schema and a Schematron schema. The XSD schema is not specified (the document’s *schemaLocation* attribute will be used); the Schematron schema is specified.

Another simple example is:

```
<schemachine xmlns=”....”>
   <title>Another Example Schema</title>
   <ns prefix=”html” url=”...” />

   <pass>
      <select engine=”schemachine:namespace_selector”>
         <param name=”pattern”>html:body</param>
         <output name=”htmlbody” />
      </select>

      <validate engine=”schemachine:relax_ng”>
         <param name=”schema” href=”....”/>
         <param name=”feasible”>true</param>
         <input name=”htmlbody”/>
      </validate>
   </pass>
</schemachine>
```

In this schema, there is a single **pass** defined. First the data from the input goes through a **selector** (which uses the eternally-provided document as input by default), then a **validator** which take its schema from an external source.

Note that there is no nesting of validator and selector. All connections are by named references. Note that one output may feed more than one input (i.e. a *tee*) but every input can only connect to a single output. This keeps the schema flatter, which allows easier integration with GUIs and easier diagramming.

Head

The head section follows Schematron. It allows

- a <title> containing various version metadata attributes
- <ns> elements for passing namespaces
- <p> elements for documentation

```
## RELAX NG compact schema
default namespace this = inherit

## <schemachine> is the top-level element
## Metadata is omitted for clarity
schemamachine = element schemachine {
  title, ns*, p*, phase*, pass+,
  (element * - this:*)*
}

## The <title> is for humans and pretty printing
title = element title { text }


## The <ns> allows definition of the namespaces
## used in the framework document. These values
## are not visible to the engines.
ns = element ns {
  attribute prefix { string },
  attribute uri { string },
  empty
}

## The <p> element is for humans and pretty printing
p = element p { text }
```

Phases

The phase section follows Schematron. It allows different phases to be specified.

```
## A <phase> enables various selectors.
phase = element phase {
       active+
}
## The <active> element specifies the selectors
```

```
active = element active {
       attribute selectors { string },
       empty
}
```

Pipelines

The schema pipelines are based on XML Pipelines, used more or less as an architecture. It allows:

- <pass> elements, which start discrete passes on the input stream
- <select> elements, which select or sort documents for use using different engines,
- <tokenize> elements, which tokenize text input using different simple engines, and
- <validate> elements, which specify the validator to use, using different engines.

These elements have subelements for <input>, <output>, <param>eters. The haltOnFail attribute prevents useless validation. The values of parameters can be specified directly as an attribute value, using a ref attribute to some declaration in the schemas section, or using an href attribute to an external document.

We speak of <pass>, <select>, <tokenize> and <validate> *processors* which use different *engines*.

The particular engines involved are the subjects of other parts of DSDL or external specifications. The parameters passed to an engine primarily specify an engines *processing strategy*. The validation strategy for a schema language may have been specified as part of the schema language (such as XML Schema's *lax* validation, or Schematron *phases*) but it may not have been specified as part of that schema language, but be implementation-dependent (such as *feasible* validation.)

```
## A <pass> is a single iteration through a document
## It allows the user to get one set of validation
## over before attempting a more specialized set.
pass = element pass {
      attribute onEmpty { ”continue” | ”fail” | ”error” }?,
      exclude*, { select | tokenize | validate )*
}


## A <select> statement scans the input and picks out
## various elements of interest as XML documents.
## If there is no <input> element, the top-level
## input is used. The <output> names the pipe to
## which one or more subsequent processors are attached.
## The engine attribute selects which selector to use:
## the typical selector is an XPath.
select = element select {
      attribute onEmpty { ”continue” | ”fail” | ”error” }?,
      attribute id { string }?,
      attribute engine { string },
      attribute haltOnFail ( ”true” | ”false”)?,
      exclude*, param*, input*, output*,
      (select | tokenize | validate )*
}


## A <tokenize> statement scans the input (which is a
## document probably with one element only) and parses
## its data content, returning the result as another
## XML document.
## If there is no <input> element, the top-level
## input is used. The <output> names the pipe to
## which one or more subsequent processors are attached.
## The engine attribute selects which tokenizer to use.
tokenize = element tokenize {
      attribute onEmpty { ”continue” | ”fail” | ”error” }?,
      attribute id { string }?,
      attribute engine { string },
      attribute haltOnFail ( ”true” | ”false”)?,
      exclude*, param*, input*, output*,
      ( select | tokenize | validate )*
}


## A <validate> statement validates its input.
## If there is no <input> element, the top-level
## input is used. The output of validators and its
## use is implementation-dependent.
## The engine attribute selects which validator to use.
validate = element validate  {
      attribute onEmpty { ”continue” | ”fail” | ”error” }?,
      attribute id { string }?,
      attribute engine { string },
      attribute haltOnFail ( ”true” | ”false”)?,
      exclude*, param*, input*
}

## The <param> element specifies parameters passed to
## the validator, tokenizer, or selector.  The name
## schema is reserved for use as the schema, but all other
## names are available. The value can be sourced inline
## as the contents of this element, or by a ref attribute
## giving and idref to some schema etc in the Schemas
## section, or by href attribute to an external resource.
param = element param {
      attribute name { string },
      (   ( attribute ref { string }, empty ) |
          ( attribute href { string }, empty) |
          ( element * - this:* ) )
}


## The <input> element names the pipe used as input for
## the current process.
input = element input {
     attribute name { string },
     empty
}


## The <output> element names the pipe used as the output
## for the current process.
output = element output {
     attribute name { string },
     empty
}

## The <exclude> element gives a namespace URI. Elements or
## attributes in that namespace should be removed from the
## document before the engine performs its work. A terminal
## process might merely ignore the information rather than
## strip it.
exclude = element exclude {
     attribute namespace { string }
}
```

Schemas

This section is designed to hold schemas, controlled vocabularies and other elements. I have not put details in here, for clarity.

Processors

The kinds of processors involved could be:

SELECTORS

- RELAN Namespaces/XPath-based selector
- APEX architectural-form-based processor (does limited renaming)
- Inheritence-based annotator (does #DEFAULT, #CURRENT, + inheritence effects)

TOKENIZER

- Regular Fragmentations: regular expression based
- Picture based (e.g. YYYY-MM-DD)
- Unit matching tokenizer

VALIDATOR

The various validators given in the other parts of DSDL + extensions:

- RELAX NG
- Schematron
- Super DTDs

- Link processors

Note that datatyping is a layer inside schema processors, in this framework.

Validity

An invocation of a processor may result in:

- *pass*,
- *fail*, or
- *error.*

An *error* result occurs because of some programming or environment problem, rather than because of a document problem. For example, a parameter value may be wrong or a schema may itself be invalid.

Validity is defined in terms of phases. A document is valid in particular phase if

- no processor results in *fail* or *error.*

In other words, all processors act as validators.

It is implementation-dependent which other diagnostic outcomes are produced as well as validity. The halt-on-fail mechanism reduces one error being reported multiple times.

NOTE: Even though processors are defined in terms of XML documents, an implementation may work by passing the parsed information set in order to preserve the original location of errors.

If the input to a processor is *empty*, i.e. there is no input document, each processor can be set (by an attribute) to

- *continue*,
- *ignore*,
- *fail*, or
- *error.*

If the processor is set to *continue,* then the engine used by that processor will determine how it treats empty input. Note that a processor may not necessarily invoke the next processors in the pipeline; this is why validity is defined negatively.

If the processor is set to *allow*, then an empty input will not cause a *fail*, or *error*, but the processor will not proceed further. For example, a selector may find no matching input. If this is allowed, the selector will result in *pass* and the subsequent processor can be set to *continue* if the subsequent processor can handle empty input or *ignore* if the subsequent processor cannot. If matching input was required, if the selector does not not signal *error* then the subsequent processor can be set to *fail*. This gives the flexibility to simplify selection patterns by factoring out empty-handling (if that is what the schema-writer chooses to use.)

Example

Here is an example of the features of the language.

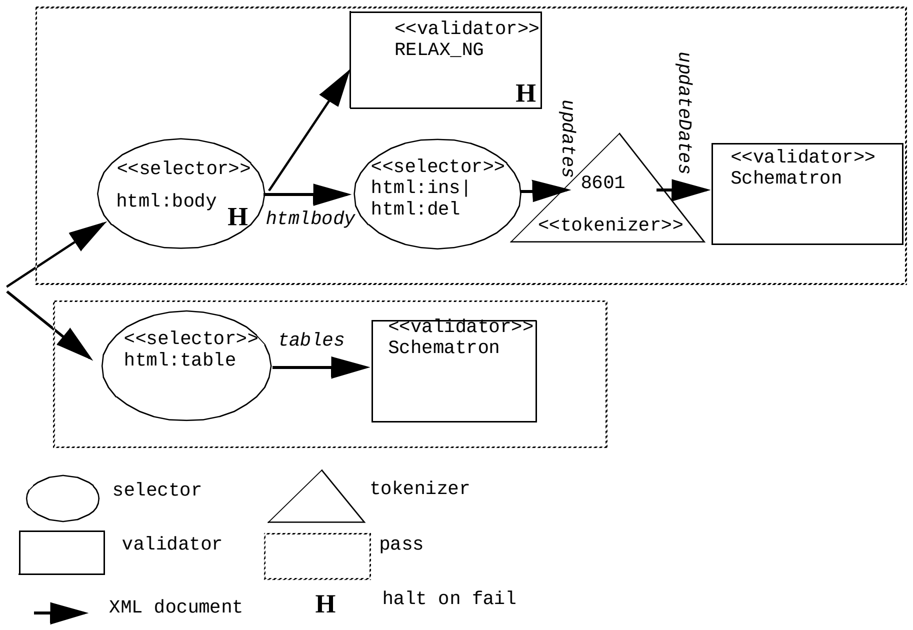
There are two design points brought out by this schema. First, that dates are validated by tokenizing them and then validating them with Schematron: to enable manipulation of data values like this, our selector mechanism needs a “wrap” function to wrap the data in an element. Second, that this kind of validation of complex text is too cumbersome to be considered as the mechanism for simple datatyping.

```
<schemachine xmlns=”....”>
  <title>larger Example Schema</title>
  <ns prefix=”html” url=”...” />

  <p>This is a larger example. The user can select two kinds of process-
ing: “basic” or “full” to validate XHTML.</p>

  <p>In “basic” processing, the html:body element from the input document
is validated using a RELAX NG schema.</p>

  <p><![CDATA[In “full” processing, the first pass has an extra stage
where the html:body element is further processed to select the datetime
attributes of any html:ins or html:del elements.  These are wrapped in a
datetime> element for passing as a single-element document, then tokenized
into another document using COBOL-style picture-tokenizing rules, before
being passed to a Schematron validator (which would presumably make some
complex checks on each individual date.)]]></p>

  <p>If the RELAX NG validation during the first pass has no errors, then
another pass is made and the tables in the document are validated against
a Schematron schema. The Schematron schema is invoked in its “cols-check”
phase, which presumably checks column counts.
</p>

 <phase name=”basic”>
    <active selector=”s1” />
  </phase>

 <phase name=”full”>
    <active selector=”s1” />
    <active selector=”s2” />
    <active selector=”s3” />
  </phase>

  <pass>
    <select engine=”schemachine:namespace_selector”
        haltOnFail=”true”  id=”s1”>
        <param name=”pattern”>html:body</param>
        <output name=”htmlbody” />
    </select>

    <validate engine=”schemachine:relax_ng” haltOnFail=”true”>
        <param name=”schema” href=”....”/>
        <input name=”htmlbody”/>
    </validate>
```

Example

```
    <select engine=”schemachine:namespace_selector” id=”s2”>
        <param name=”pattern”>html:ins/@datetime
            | html:del/@datetime</param>
        <param name=”wrap”>datetime</param>
        <input name=”htmlbody” />
        <output name=”updates” />
    </select>

    <tokenize engine=”schemachine:picture-tokenizer” onEmpty=”ignore”>
        <param name=”picture”>YYYY-MM-DD</param>
        <param name=”delimiters”>-</param>
        <input name=”updates” />
        <output name=”updateDates” />
    </tokenize>

    <validate engine=”schemachine:Schematron”>
        <param name=”schema” href=”...schematron that validates dates” />
        <param name=”year”>YYYY</param>
        <param name=”month”>MM</param>
        <param name=”day”>DD</param>
    <input name=”updateDates” />
    </validate>

  </pass>

 <pass>
    <select engine=”schemachine:namespace_selector” id=”s3”>
        <param name=”pattern”>html:table</param>
        <output name=”tables” />
    </select>

    <validate engine=”schemachine:schematron” onEmpty=”ignore”>
        <param name=”schema” href=”....”/>
        <param name=”phase”>cols-check</param>
        <input name=”tables”/>
    </validate>

  </pass>
</schemachine>
```

Here is an informal block diagram of this.



To clarify, here are the documents at various pipes.

```
<!-- Input document-->
<html xmlns=” namespace for XHTML”>
        <head><title>EG</title></head>
        <body>
                <ins datetime=”2002-06-17”><p>EG</p></ins>
        </body>
</html>
```

<!-- **htmlbody** document-->

```
        <body  xmlns=” namespace for XHTML”>
                <ins datetime=”2002-06-17”><p>EG</p></ins>
        </body>
```

<!-- **updates** document-->

```
<datetime=”2002-06-17”>
```

<!-- **updateDates** document-->

```
<datetime><YYYY>2002</YYYY>-<MM>06</MM>-<DD>17</DD></datetime>
```

<!-- **tables** document is *empty* -->