Schematron Overview

excerpted from Leigh Dodds' 2001 *XSLT UK* paper, "Schematron: validating XML using XSLT"

Leigh Dodds, ingenta ltd, xmlhack.com

April 2001

Abstract

Schematron [Schematron] is a structural based validation language, defined by Rick Jelliffe, as an alternative to existing grammar based approaches. Tree patterns, defined as XPath expressions, are used to make assertions, and provide user-centred reports about XML documents. Expressing validation rules using patterns is often easier than defining the same rule using a content model. Tree patterns are collected together to form a Schematron schema.

Schematron is a useful and accessible supplement to other schema languages. The open-source XSLT implementation is based around a core framework which is open for extension and customisation.

Overview

This paper provides an introduction to Schematron; an innovative XML validation language developed by Rick Jelliffe. This innovation stems from selecting an alternative approach to validation than existing schema languages: Schematron uses a tree pattern based paradigm, rather than the regular grammars used in DTDs and XML schemas. As an extensible, easy to use, open source tool Schematron is an extremely useful addition to the XML developers toolkit.

The initial section of this paper conducts a brief overview of tree pattern validation, and some of the advantages it has in comparison to a regular grammar approach. This is followed by an outline of Schematron and the intended uses which have guided its design. The Schematron language is then discussed, covering all major elements in the language with examples of their

usage. A trivial XML vocabulary is introduced for the purposes of generating examples.

The later sections in this paper provides an overview of the open source XSLT framework used to implement the Schematron language. The Schematron conformance language for custom implementation is also introduced. The paper completes with some suggestions of possible future extensions.

Introducing Tree Patterns as a Validation Mechanism

During the last few years a number of different XML schema languages have appeared as suggested replacements for the ageing Document Type Definition (DTD). The majority of these have taken the basic premise of recasting DTD functionality in XML syntax with the addition, in some cases, of other features such as data typing, inheritance, etc [XMLSchema]. The use of XML syntax provides additional flexibility through leveraging existing tools for markup manipulation, while the 'value added' features satisfy the requirements of developers looking for closer integration with databases and object-oriented languages.

Yet the fundamental approach adopted by these languages does not diverge greatly from the DTD paradigm: the definition of schemas using regular grammars. Less formally, schemas are constructed by defining parent-child and sibling relationships [Jelliffe1999a]. For example in a DTD one might write:

```
<!ELEMENT wall EMPTY>
<!ELEMENT roof EMPTY>
<!ELEMENT house (wall+, roof)>
```

This defines three elements, `wall`, `root`, and `house`. The parent-child relationship between `house` and `wall` elements is defined in the content model for `house`. A house may have several walls.

The sibling relationship between `wall` and `roof` is derived from the same content model, which defines them as legal sibling children of the `house` element.

However this means that DTDs, and similar derivatives, are unable to define (and hence constrain) the other kinds of relationships that exist amongst markup elements within a document. As the XPath specification [XPath] shows, there are many possible kinds of relationship, known as 'axes'.

Example 1. Tree relationships (axes) defined by XPath

- child
- descendant
- parent
- ancestor
- following-sibling
- preceding-sibling

- following
- preceding
- attribute
- namespace
- self
- descendant-or-self
- ancestor-or-self

While XML does include an ID/IDREF mechanism which allows for cross-referencing between elements, and hence another form of relationships, it only weakly binds those elements. There is no enforcement that an IDREF must point to an ID on a particular element type, simply that is must point to an existing ID, and further that all IDs must be unique.

Having highlighted the fact that the existing schema paradigm can only express constraints among data items in terms of the child and sibling axes, it is natural to consider whether an alternate paradigm might allow a schema author to exploit these additional relationships to define additional types of constraint amongst document elements. Tree patterns do just that, and XPath provides a convenient syntax in which to express those patterns.

Validation using tree patterns is a two-step process:

- Firstly the candidate objects (in XPath terms, nodes) to be validated must be identified. i.e. identify a *context*
- Secondly assertions must be made about those objects to test whether they fulfill the required constraints.

Both the candidate object selection, and the assertions can be defined in terms of XPath expressions. More formally, the nodes and arcs within a graph of data can be traversed to both identify nodes, and then make assertions about the relationships of those nodes to others within the same graph. Assertions are therefore the mechanism for placing constraints on the relationships between nodes in a graph (elements and attributes in an XML document).

For example, we may select all `house` nodes within a document using the expression:

```
//house
```

And then assert that all `house`s have `wall`s by confirming that the following pattern selects one or more child nodes (within the context defined by the previous selection):

```
child::wall
```

Regular grammars, as used in DTDs, can then be viewed as tree patterns where the only available axis is the parent-child axis [Jelliffe1999e]. Full use of tree pattern validation provides the maximum amount of freedom when modelling constraints for a schema. This comes at very little cost: XPath is available in most

XML environments. For example the following types of constraint are hard, or impossible to express with other schema languages.

Example 2. Examples of 'difficult' constraints

- Where attribute X has a value, attribute Y is also required
- Where the parent of element A is element B, it must have an attribute Y, otherwise an attribute Z
- The value of element P must be either "foo", "bar" or "baz"

Tree patterns are the schema paradigm underpinning Schematron as a validation language.

There are reasons to believe that tree-pattern validation may be more suitable in an environment where documents are constructed from elements in several namespaces (often termed 'data islands'). As many consider that the future of XML document interchange on the Internet will involve significant mixing of vocabularies, a flexible approach may bring additional benefits.

Introducing Schematron

Background

Schematron [Schematron] is an XML schema language designed and implemented by Rick Jelliffe at the Academia Sinica Computing Centre, Taiwan. It combines powerful validation capabilities with a simple syntax and implementation framework. Schematron is open source, and is (at the time of writing) being migrated to SourceForge to better manage its development by a rapidly growing community of users.

Schematron traces its ancestry [Jelliffe1999f] indirectly from SGML DTDs via Assertion Grammars [Raggett], Groves and Property Sets [Arciniegas]. A recent review of six current schema languages [Lee] supports this view, declaring Schematron to be unique in both its approach and intent. Before discussing the details of the Schematron language it is worth reviewing the design goals which have been highlighted by its author.

Design Goals

There are several aims which Rick Jelliffe which believed were important during the design and specification of Schematron [Schematron], [Jelliffe2001]:

- Promote natural language descriptions of validation failures, i.e. diagnose as well as reject
- Reject the binary valid/invalid distinction which is inherent in other schema languages
- Aim for a short learning curve by layering on existing tools (XPath and XSLT)

- Trivial to implement on top of XSLT
- Provide an architecture which lends itself to GUI development environments
- Support workflow by providing a system which understands the phases through which a document passes in its lifecycle

Target Uses

Jelliffe has also suggested [Jelliffe2001] several target environments in which Schematron is intended to add value:

- Document validation in software engineering, through the provision of interlocking constraints
- Mining data graphs for academic research or information discovery. Constraints may be viewed as hypotheses which are tested against the available data
- Automatic creation of external markup through the detection of patterns in data, and generation of links
- Use as a schema language for "hard" markup languages such as RDF.
- Aid accessibility of documents, by allowing usage constraints to be applied to documents

How It Works

The implementation of Schematron derives from the observation that tree pattern based validators can be trivially constructed usings XSLT stylesheets [Jelliffe1999b], [Norton]. For example, a simple stylesheet that validates that `house`s must have `wall`s can be defined as follows:

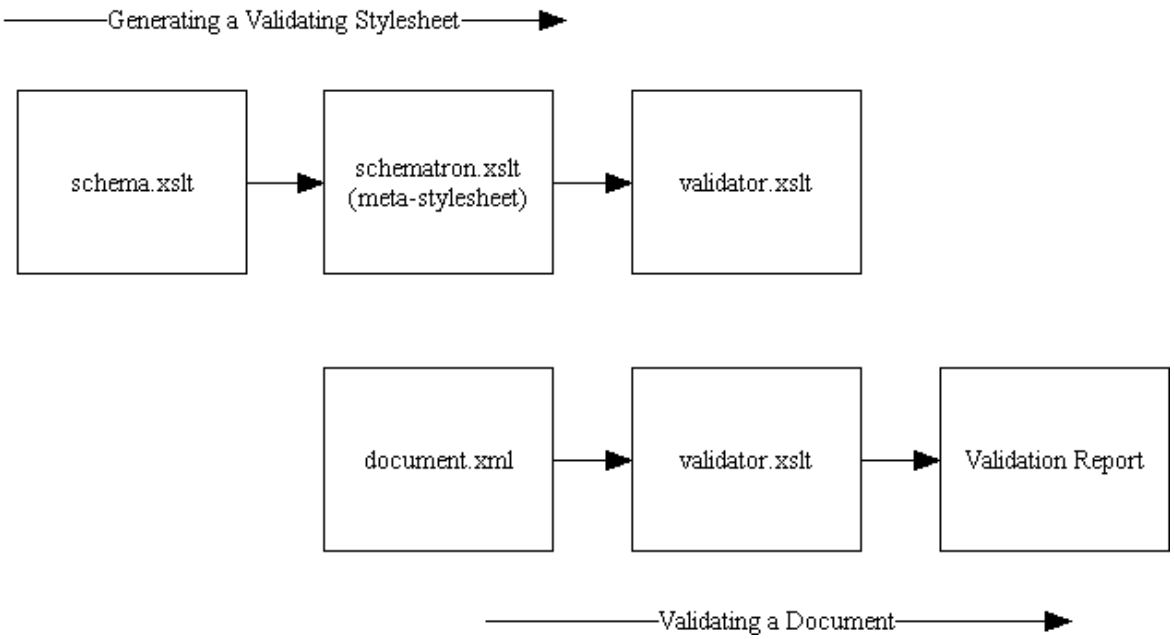
```
    <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

      <!-- select all houses -->
      <xsl:template match="//house">
        <!-- test whether it has any walls -->
        <xsl:if test="not(wall)">
          This house has no walls!
        </xsl:if>
      </xsl:template>

  </xsl:stylesheet>
```

It should be obvious from the above that if a `house` does not have any `wall`s, a simple error message will be displayed to the user.

Schematron takes this a natural step further by defining a schema language which, when transformed through a meta-stylesheet (i.e. a stylesheet which generates other stylesheets), produces XSLT validators similar to the above. The following diagram summarises this process.

Figure 1. Generating Validators Using Schematron

Editor’note: most Schematron users would call the first box “schema.sch”.



Schematron is therefore a simple layer above XPath and XSLT allowing it to leverage existing tools, and benefit from a framework which is already familiar to XSLT developers. Yet from a user perspective, the details of XSLT are hidden; the end-user need only grapple with the XPath expressions used to define constraints.

The following section outlines the Schematron assertion language which is used to define Schematron schemas. The last section in the paper provides information on the Schematron implementation (i.e. the metastylesheet) which will be of interest to implementors seeking to customise Schematron for particular needs.

The Schematron Assertion Language

This section introduces the Schematron assertion language which can be used to generate XSLT validators using the Schematron implementation. All following examples conform to a simple XML vocabulary introduced in the next section.

DTD For Example Content

The examples used within this section will refer to a fictional XML language for describing building projects. While the examples could have been couched in terms of an existing schema language, the intention is to provide a simple vocabulary which does not assume any prior knowledge on behalf of the user. It should be stressed that, while the examples themselves may be trivial this should not be taken to indicate any specific limitation in Schematron, which is capable of handling much more complex schemas.

The following DTD defines the building project vocabulary:

Example 3. Simple XML language for illustrative purposes

```
<!ELEMENT wall EMPTY/>
 <!ELEMENT roof EMPTY/>
 <!ELEMENT street #PCDATA>
 <!ELEMENT town #PCDATA>
 <!ELEMENT postcode #PCDATA>
 <!ELEMENT firstname #PCDATA>
 <!ELEMENT lastname #PCDATA>
 <!ELEMENT certification EMPTY>
 <!ATTLIST certification number CDATA #REQUIRED>
 <!ELEMENT telephone #PCDATA>

 <!ELEMENT address (street, town, postcode)>
 <!ELEMENT builder (firstname, lastname, certification)>
 <!ELEMENT owner (firstname, lastname, telephone)>
 <!ELEMENT house (wall+, roof?, address, (builder|owner)?>
```

This schema allows us to describe a house consisting of a number of walls and a roof. The roof may not be present if the house is still under construction.

A house has an address which consists of a street name, town and a postcode.

A house should have either a builder who is currently assigned to its construction (and all builders must be certified), or an owner. Certification numbers of builders, and telephone numbers of owners are also recorded for adminstrative purposes.

A sample document instance conforming to this schema is:

Example 4. Sample document instance

```
<house>
  <wall/>
  <wall/>
  <wall/>
  <wall/>
  <address>
    <street>1 The High Street</street>
    <town>New Town</town>
    <postcode>NT1</postcode>
  </address>
  <builder>
    <firstname>Bob</firstname>
    <lastname>Builder</lastname>
    <certification number="123"/>
  </builder>
</house>
```

Core Elements: Assert and Report

The basic building blocks of the schematron language are the `assert` and `report`

elements. These define the constraints which collectively form the basis of a Schematron schema. Constraints are assertions (boolean tests) that are made about patterns in an XML document; these patterns and tests are defined using XPath expressions.

The best illustration is a simple example:

Example 5. A simple assertion

```
<assert test="count(walls) = 4">This house does not have four walls</assert>
```

This demonstrates a simple assertion which counts the number of `walls` in the current context. Recall that validation is a two step process of identification and followed by assertion. The identification step generates the context in which assertions are made. This is covered in the next section.

If there are not four walls then the assertion fails and a message, the content of the `assert` element, is displayed to the user.

Asserts therefore operate in the conventional way: if the assertion evaluates to false some action is taken. The `report` element works in the opposite manner. If the test in a report element evaluates to true then action is taken.

While reports and asserts are effectively the inverse of one another, the intended uses of the two elements are quite different. An assert is used to test whether a document conforms to a particular schema, generating actions if deviations are encountered. A report is used to highlight features of the underlying data:

Example 6. A simple report

```
<report test="not(roof)">This house does not have a roof</assert>
```

The distinction may seem subtle, especially when grapplying with a constraint which may be expressed simpler in one way or the other. However Schematron itself does not define the action which must be taken on a failed assert, or successful report, this is implementation specific. The default behaviour is to simply provide the user with the provided message. An implementation may choose to handle these two cases differently.

It is worth noting that there is a trade-off to be made when defining tests on these elements. In some cases a single complex XPath expression may accurately capture the desired constraint. Yet it is closer to the 'spirit' of Schematron's design to use several smaller tests that collectively describe the same constraint. Specific tests can more accurately provide feedback to a user, than a single general test and associated message.

Assert and Report elements may contain a `name` element which has an optional path attribute. This element will be substituted with the name of the current element before a message is passed to the user. When supplied the path attribute

should contain an XPath expression referencing an alternate element. This is useful for giving additional feedback to the user about the specific element that failing an assertion.

Schematron 1.5, released in January 2001, adds the ability to provide detailed diagnostic information to users. Assert and report messages should be simple declarative statements of what is, or should be. Diagnostics can include detailed information that can be provided to the user as appropriate to the Schematron implementation. Diagnostic information is grouped separately to constraints, and is cross-referenced from a diagnostic attribute.

Example 7. Diagnostic Example

```
<assert test="count(walls) = 4" diagnostics="1">This house does not have four walls</assert>

...

<diagnostics>
  <diagnostic id="1">
    Its an odd house which has more or less than four walls! Consult your architect...
  </diagnostic>

</diagnostics>
```

Writing Rules

As noted earlier, constraints must be applied within a context. The context for constraints is defined by grouping them together to form `rule`s.

Example 8. Defining the context for assertions

```
<rule context="house">
  <assert test="count(wall) = 4">A house should have four walls</assert>
  <report test="not(roof)">This house does not have a roof</assert>
</rule>
```

The context attribute for a `rule` contains an XPath expression. This identifies the candidate nodes to which constraints will be applied. The above example checks that a `house` contains 4 `wall` child elements, and provides feedback to the user if it is missing a `roof`.

Schematron 1.5 add a simple macro mechanism for rules which is useful when combining constraints. To do this, a rule may be declared as 'abstract'. The contents of this rule may be included by other rules as necessary. This is achieved through the use of the `extends` element.

Example 9. Using abstract rules and the extends element

```
<rule abstract="true" id="nameChecks">
   <assert test="firstname">A person must have a first name</assert>
```

```
   <assert test="lastname">A person must have a last name</assert>
</rule>

<rule context="builder">
   <extends rule="nameChecks"/>

   <!-- builder specific constraints -->
   ...
</rule>

<rule context="owner">
   <extends rule="nameChecks"/

   <!-- owner specific constraints -->
   ...
</rule>
```

In the above example an abstract rule is defined, and assigned the id "*nameChecks*". Two assertions are associated with this abstract rule: checking that an element has a firstname and a lastname. These assertions are imported by the other non-abstract rules and will be applied along with the other constraints specific to that element. An abstract rule may contain assert and report elements but it cannot have a context. Assertions from an abstract rule obtain their context from the importing rule.

Producing Patterns and Schemas

The next most important element in a Schematron schema is `pattern`. Patterns gather together a related set of rules. A particular schema may include several patterns that logically group the constraints.

Example 10. Grouping rules using patterns

```
<pattern name="Adminstration Checks"
         see="http://www.buildingprojects.org/admin/procedures.html">

   <rule abstract="true" id="nameChecks">
      <assert test="firstname">A person must have a first name</assert>
      <assert test="lastname">A person must have a last name</assert>
   </rule>

   <rule context="builder">
      <extends rules="nameChecks"/>

      <!-- builder specific constraints -->
      ...
   </rule>

   <!-- additional rules -->
   ...

</pattern>

<pattern name="Other Constraints">
   <!-- other rules -->
```

```
</pattern>
```

A pattern should have a name and may refer to additional documentation using a URL. A Schematron implementation can then furnish the user with a link to supporting documentation.

Patterns defined within a schema will be applied sequentially (in lexical order). Nodes in the input document are then matched against the contexts defined by the rules contained within each pattern. If a node is found to match the context of a particular rule, then the assertions which it contains will be applied. Within a pattern a given node can only be matched against a *single* rule. Rules within separate patterns may match the same node, but only the first match within a pattern will be applied. An example of an incorrect schema is given below.

Example 11. Incorrect use of rule contexts

```
<pattern name="Adminstration Checks">

   <!-- this rule WILL be matched -->
   <rule context="builder">
      <assert test="firstname">A person must have a first name</assert>
   </rule>

   <!-- this rule WILL NEVER be matched -->
   <rule context="builder">
      <assert test="lastname">A person must have a last name</assert>
   </rule>

   <!-- additional rules -->
   ...

</pattern>

<!-- rules in this pattern will be checked after the above -->
<pattern name="Other Constraints">

   <!-- this rule WILL be matched -->
   <rule context="builder">
      <assert test="certification">A builder must be certified</assert>
   </rule>

   <!-- additional rules -->
   ...

</pattern>
```

Care should be taken when defining contexts to ensure that these circumstances never arise.

The last step in defining a Schematron schema is to wrap everything up in a `schema` element.

Example 12. Grouping patterns to create a schema

```
<schema xmlns="http://www.ascc.net/xml/schematron">
   <title>A Schematron Schema for Validating Building Project Documents</title>

   <ns uri="http://www.buildingprojects.org/admin/" prefix="bld"/>

   <pattern name="Adminstration Checks">
   <!-- rules -->
   </pattern>

   <pattern name="Construction Checks">
   <!-- rules -->
   </pattern>

</schema>
```

There are several points to note about the above schema. Firstly it introduces the namespace for Schematron documents, which is "http://www.ascc.net /xml/schematron". Secondly a schema may have a title; this is recommended.

Conclusions

Schematron is unique amongst current schema languages in its divergence from the regular grammar paradigm, and its user-centric approach.

Schematron is not meant as a replacement for other schema languages; it is not expected to be easily mappable onto database schemas or programming language constructs. It is a simple, easy to learn language that can perform useful functions in addition to other tools in the XML developers toolkit.

It is also a tool with little overhead, both in terms of its learning curve and its requirements. XSLT engines are regular components in any XML application framework. Schematrons use of XPath and XSLT make it instantly familiar to XML developers.

A significant advantage of Schematron is the ability to quickly produce schemas that can be used to enforce house style rules and, more importantly, accessibility guidelines without alteration to the schema to which a document conforms. An XHTML document is still an XHTML document even if it does not meet the Web Accessibility Initiative Guidelines [WAI]. These kind of constraints describe a policy which is to be enforced on a document, and can thus be layered above other schema languages. Indeed in many cases it may be impossible for other languages to test these kinds of constraints.

Bibliography

[Arciniegas] Fabio Arciniegas. 19/04/2000. *Groves Explained* .

[Conformance] Rick Jelliffe. *Schematron Conformance Language* .

[Dodds] Leigh Dodds. 10/1/2001. *Old Ghosts: XML Namespaces* .

[Holman] Ken Holman. *Practical Transformations using XSLT and XPath* .

[Jelliffe1999a] Rick Jelliffe. 1999. *From Grammars To The Schematron* .

[Jelliffe1999b] Rick Jelliffe. 24/01/1999. *Using XSL as a Validation Language* .

[Jelliffe1999c] Rick Jelliffe. 11/06/1999. *How to Promote Organic Plurality on the WWW* .

[Jelliffe1999d] Rick Jelliffe. 30/07/1999. *Weak Validation* .

[Jelliffe1999e] Rick Jelliffe. 01/08/1999. *Axis Models & Path Models: Extending DTDs with XPaths* .

[Jelliffe1999f] Rick Jelliffe. 1999. *Family Tree of Schema Languages for Markup Languages* .

[Jelliffe2000] Rick Jelliffe. 19/10/2000. *Getting Information Into Markup: the Data Model Behind the Schematron Assertion Language* .

[Jelliffe2001] Rick Jelliffe. 31/1/2001. *The Schematron Assertion Language 1.5* .

[Lee] . *Comparative Analysis of Six XML Schema Languages* .

[Norton] Francis Norton. 20/5/1999. *Generating XSL for Schema Validation* .

[OgbujiC] Chimezie Ogbuji. *Validating XML with Schematron* .

[OgbujiU] Uche Ogbuji. *Introducing the Schematron* .

[Quick] Rick Jelliffe. *Schematron Quick Reference* .

[Raggett] Dave Raggett. May 1999. *Assertion Grammars* .

[RDDL] Jonathan Borden and Tim Bray. 22/1/2001. *Resource Directory Description Language* .

[Schematron] Rick Jelliffe. *Schematron Home Page* .

[WAI] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. 5/5/1999. *Web Content Accessibility Guidelines 1.0* .

[XHTML] Mark Baker, Masayasu Ishikawa, Shinichi Matsui, Peter Stark, Ted Wugofski, and Toshihiko Yamakami. 19/12/2000. *XHTML Basic* .

[XMLNames] Tim Bray, Dave Hollander, and Andrew Layman. 14/1/1999. *Namespaces in XML* .

[XMLSchema] David Fallside. 24/10/2000. *XML Schema Part 0: Primer* .

[XPath] James Clark and Steven DeRose. November 1999. *XML Path Language (XPath) Version 1.0* .

[XSLT] James Clark. November 1999. *XML Transformations (XSLT) Version 1.0* .

[Zvon] Miloslav Nic. *Zvon Schematron Tutorial* .