

# An Introduction to Schematron

November 12, 2003

[Eddie Robertsson \(/pub/au/221\)](#)

## Table of Contents

- [Introduction to Schematron](#)
- [Schematron hierarchy](#)
- [Assertions](#)
- [Rules](#)
- [Patterns](#)
- [Namespaces and Schematron](#)
- [Schematron processing \(schematron.html#Schematron\\_processing\)](#)
- [Schematron processing using XSLT \(schematron.html#Schematron\\_processing\\_using\\_XSLT\)](#)
- [ISO Schematron \(schematron.html#ISO\\_Schematron\)](#)

The Schematron schema language differs from most other XML schema languages in that it is a rule-based language that uses path expressions instead of grammars. This means that instead of creating a grammar for an XML document, a Schematron schema makes assertions applied to a specific context within the document. If the assertion fails, a diagnostic message that is supplied by the author of the schema can be displayed.

One advantage of a rule-based approach is that in many cases modifying the wanted constraint written in plain English can easily create the Schematron rules. For example, a simple content model can be written like this: "The `Person` element should in the XML instance document have an attribute `Title` and contain the elements `Name` and `Gender` in that order. If the value of the `Title` attribute is 'Mr' the value of the `Gender` element must be 'Male'."

In this sentence the context in which the assertions should be applied is clearly stated as the `Person` element while there are four different assertions:

- The context element ( `Person` ) should have an attribute `Title`
- The context element should contain two child elements, `Name` and `Gender`
- The child element `Name` should appear before the child element `Gender`

- If attribute `Title` has the value 'Mr' the element `Gender` must have the value 'Male'

In order to implement the path expressions used in the rules in Schematron, [XPath \(http://www.w3.org/TR/xpath\)](http://www.w3.org/TR/xpath) is used with various extensions provided by [XSLT \(http://www.w3.org/TR/xslt\)](http://www.w3.org/TR/xslt). Since the path expressions are built on top of XPath and XSLT, it is also trivial to implement Schematron using XSLT, which is shown later in the section [Schematron processing \(schematron.html#Schematron\\_processing\)](http://schematron.html#Schematron_processing).

It has already been mentioned that Schematron makes various assertions based on a specific context in a document. Both the assertions and the context make up two of the four layers in Schematron's fixed four-layer hierarchy:

1. phases (top-level)
2. patterns
3. rules (defines the context)
4. assertions

## Schematron hierarchy

This introduction covers only three of these layers (patterns, rules and assertions); these are most important for using embedded Schematron rules in RELAX NG. For a full description of the Schematron schema language, see the [Schematron specification \(http://www.ascc.net/xml/resource/schematron/Schematron2000.html\)](http://www.ascc.net/xml/resource/schematron/Schematron2000.html).

The three layers covered in this section are constructed so that each assertion is grouped into rules and each rule defines a context. Each rule is then grouped into patterns, which are given a name that is displayed together with the error message (there is really more to patterns than just a grouping mechanism, but for this introduction this is sufficient).

The following XML document contains a very simple content model that helps explain the three layers in the hierarchy:

```
<Person Title="Mr">
  <Name>Eddie</Name>
  <Gender>Male</Gender>
</Person>
```

## Assertions

The bottom layer in the hierarchy is the assertions, which are used to specify the constraints that should be checked within a specific context of the XML instance document. In a Schematron schema, the typical element used to define assertions is `assert`. The `assert` element has a `test` attribute, which is an [XSLT pattern \(http://www.w3.org/TR/2000/WD-xslt11-20001212/#patterns\)](http://www.w3.org/TR/2000/WD-xslt11-20001212/#patterns). In the preceding example, there was four assertions made on the document in order to specify the content model, namely:

- The context element ( `Person` ) should have an attribute `Title`
- The context element should contain two child elements, `Name` and `Gender`
- The child element `Name` should appear before the child element `Gender`
- If attribute `Title` has the value 'Mr' the element `Gender` must have the value 'Male'

Written using Schematron assertions this would be expressed as

```
<assert test="@Title">The element Person must have a Title attribute.
</assert>
<assert test="count(*) = 2 and count(Name) = 1 and count(Gender)=
1">The element Person should have the child elements Name and Gender.
</assert>
<assert test="*[1] = Name">The element Name must appear before element
Gender.</assert>
<assert test="(@Title = 'Mr' and Gender = 'Male') or @Title !=
'Mr'">If the Title is "Mr" then the gender of the person must be
"Male". </assert>
```

If you are familiar with XPath, these assertions are easy to understand, but even for people with limited experience using XPath they are rather straightforward. The first assertion simply tests for the occurrence of an attribute `Title`. The second assertion tests that the total number of children is equal to 2 and that there is one `Name` element and one `Gender` element. The third assertion tests that the first child element is `Name`, and the last assertion tests that if the person's title is 'Mr' the gender of the person must be 'Male'.

If the condition in the `test` attribute is not fulfilled, the content of the assertion element is displayed to the user. So, for example, if the third condition was broken (`*[1] = Name`), the following message is displayed:

```
The element Name must appear before element Gender.
```

Each of these assertions has a condition that is evaluated, but the assertion does not define where in the XML instance document this condition should be checked. For example, the first assertion tests for the occurrence of the attribute `Title`, but it is not specified on which element in the XML instance document this assertion is applied. The next layer in the hierarchy, the rules, specifies the location of the contexts of assertions.

## Rules

The rules in Schematron are declared by using the `rule` element, which has a `context` attribute. The value of the `context` attribute must match an XPath Expression (<http://www.w3.org/TR/xpath#section-Expressions>) that is used to select one or more nodes in the document. Like the name suggests, the `context` attribute is used to specify the context in the XML instance document where the assertions should be applied. In the previous example the context was specified to be the `Person` element, and a Schematron rule with the `Person` element as context would simply be

```
<rule context="Person"></rule>
```

Since the rules are used to group together all the assertions that share the same context, the rules are designed so that the assertions are declared as children of the `rule` element. For the previous example this means that the complete Schematron rule would be

```
<rule context="Person">
  <assert test="@Title">The element Person must have a Title
attribute.</assert>
  <assert test="count(*) = 2 and count(Name) = 1 and count(Gender) =
1">The element Person should have the child elements Name and Gender.
</assert>
  <assert test="*[1] = Name">The element Name must appear before
element Age.</assert>
  <assert test="(@Title = 'Mr' and Gender = 'Male' or @Title !=
'Mr')">If the Title is "Mr" then the gender of the person must be
"Male".</assert>
</rule>
```

This means that all the assertions in the rule will be tested on every `Person` element in the XML instance document. If the context is not all the `Person` elements, it is easy to change the XPath location path to define a more restricted context. The value `Database / Person` for example

sets the context to be all the `Person` elements that have the element `Database` as its parent.

## Patterns

The third layer in the Schematron hierarchy is the pattern, declared using the `pattern` element, which is used to group together different rules. The `pattern` element also has a `name` attribute that will be displayed in the output when the pattern is checked. For the preceding assertions, you could have two patterns: one for checking the structure and another for checking the co-occurrence constraint. Since patterns group together different rules, Schematron is designed so that rules are declared as children of the `pattern` element. This means that the previous example, using the two patterns, would look like

```
<pattern name="Check structure">
  <rule context="Person">
    <assert test="@Title">The element Person must have a Title
attribute.</assert>
    <assert test="count(*) = 2 and count(Name) = 1 and
count(Gender) = 1">The element Person should have the child elements
Name and Gender.</assert>
    <assert test="*[1] = Name">The element Name must appear before
element Age.</assert>
  </rule>
</pattern>
<pattern name="Check co-occurrence constraints">
  <rule context="Person">
    <assert test="(@Title = 'Mr' and Gender = 'Male') or @Title !=
'Mr' ">If the Title is "Mr" then the gender of the person must be
"Male".</assert>
  </rule>
</pattern>
```

The name of the pattern will always be displayed in the output, regardless of whether the assertions fail or succeed. If the assertion fails, the output will also contain the content of the assertion element. However, there is also additional information displayed together with the assertion text to help you locate the source of the failed assertion. For example, if the co-occurrence constraint above was violated by having `Title = 'Mr'` and `Gender = 'Female'` then the following diagnostic would be generated by Schematron:

From pattern "Check structure":

From pattern "Check co-occurrence constraints":

```

    Assertion fails: "If the Title is "Mr" then the gender of the
    person must be "Male"." at
        /Person[1]
        <Person Title="Mr">...</>

```

The pattern names are always displayed, while the assertion text is only displayed when the assertion fails. The additional information starts with an XPath expression that shows the location of the context element in the instance document (in this case the first `Person` element) and then on a new line the start tag of the context element is displayed.

The assertion to test the co-occurrence constraint is not trivial, and in fact this rule could be written in a simpler way by using an XPath predicate (<http://www.w3.org/TR/xpath#predicates>) when selecting the context. Instead of having the context set to all `Person` elements, the co-occurrence constraint can be simplified by only specifying the context to be all the `Person` elements that have the attribute `Title = 'Mr'`. If the rule was specified using this technique the co-occurrence constraint could be described like this

```

<rule context="Person[@Title='Mr']">
  <assert test="Gender = 'Male'">If the Title is "Mr" then the
  gender of the person must be "Male".</assert>
</rule>

```

By moving some of the logic from the assertion to the specification of the context, the complexity of the rule has been decreased. This technique is often very useful when writing Schematron schemas.

This concludes the introduction of patterns; now all that is left to do to complete the schema is to wrap the patterns in the Schematron schema in a `schema` element, and to specify that all the Schematron elements used should be defined in the Schematron namespace, <http://www.ascc.net/xml/schematron>. The complete Schematron schema for the example follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
  <sch:pattern name="Check structure">
    <sch:rule context="Person">
      <sch:assert test="@Title">The element Person must have a
      Title attribute</sch:assert>
      <sch:assert test="count(*) = 2 and count(Name) = 1 and
      count(Gender) = 1">The element Person should have the child elements

```

```

Name and Gender.</sch:assert>
    <sch:assert test="*[1] = Name">The element Name must
appear before element Gender.</sch:assert>
    </sch:rule>
</sch:pattern>
<sch:pattern name="Check co-occurrence constraints">
    <sch:rule context="Person">
        <sch:assert test="(@Title = 'Mr' and Gender = 'Male') or
@Title != 'Mr'">If the Title is "Mr" then the gender of the person
must be "Male".</sch:assert>
    </sch:rule>
</sch:pattern>
</sch:schema>

```

## Namespaces and Schematron

Schematron can also be used to validate XML instance documents that use namespaces. Each namespace used in the XML instance document should be declared in the Schematron schema. The element used to declare namespaces are the `ns` element which should appear as a child of the `schema` element. The `ns` element has two attributes, `uri` and `prefix`, which are used to define the namespace URI and the namespace prefix. If the XML instance document in the example were defined in the namespace `http://www.topologi.com/example`, the Schematron schema would look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
    <sch:ns uri="http://www.topologi.com/example" prefix="ex"/>
    <sch:pattern name="Check structure">
        <sch:rule context="ex:Person">
            <sch:assert test="@Title">The element Person must have a
Title attribute</sch:assert>
            <sch:assert test="count(ex:*) = 2 and count(ex:Name) = 1
and count(ex:Gender) = 1">The element Person should have the child
elements Name and Gender.</sch:assert>
            <sch:assert test="ex:*[1] = ex:Name">The element Name must
appear before element Gender.</sch:assert>
        </sch:rule>
    </sch:pattern>
    <sch:pattern name="Check co-occurrence constraints">
        <sch:rule context="ex:Person">
            <sch:assert test="(@Title = 'Mr' and ex:Gender = 'Male')
or @Title != 'Mr'">If the Title is "Mr" then the gender of the person
must be "Male".</sch:assert>
        </sch:rule>
    </sch:pattern>
</sch:schema>

```

```
        </sch:rule>
    </sch:pattern>
</sch:schema>
```

Note that all XPath expressions that test element values now include the namespace prefix `ex` .

This Schematron schema would now validate the following instance:

```
<ex:Person Title="Mr" xmlns:ex="http://www.topologi.com/example">
  <ex:Name>Eddie</ex:Name>
  <ex:Gender>Male</ex:Gender>
</ex:Person>
```

## Schematron processing

There are currently a few different Schematron processors available. In general these processors are divided into two groups: XSLT-based and XPath-based processors.

Since the Schematron specification is built on top of XSLT and XPath, all you really need to perform Schematron validation is an XSLT processor. Validation is then performed in two stages: the Schematron schema is, first, turned into a validating XSLT stylesheet that is, second, applied to the XML instance document to get the validation results. Since XSLT processors are available in most programming languages and on most platforms and operating systems, this validation technique will be explained in more detail in the next section.

For more API-like validators there currently exists two Schematron processors that are built on top of XPath. The first one is a Java implementation by Ivelin Ivanov that is part of the Cocoon project. This implementation can be accessed through the SourceForge.NET [website \(http://prdownloads.sourceforge.net/freebuilder/sch-java.zip\)](http://prdownloads.sourceforge.net/freebuilder/sch-java.zip). The second XPath implementation is Daniel Cazzulino's [Schematron.NET \(http://sourceforge.net/projects/dotnetopensrc/\)](http://sourceforge.net/projects/dotnetopensrc/) using the Microsoft .NET platform which is also available at SourceForge.

XPath implementations of Schematron are generally faster than the XSLT approach because they do not need the extra step of creating a validating XSLT stylesheet and have less functionality. This also means that the functions in Schematron that are XSLT-specific (for example, *document()* and *key()* functions) are unavailable in the XPath implementation. This means, for example, that constraints between XML instance documents cannot be checked using an XPath implementation

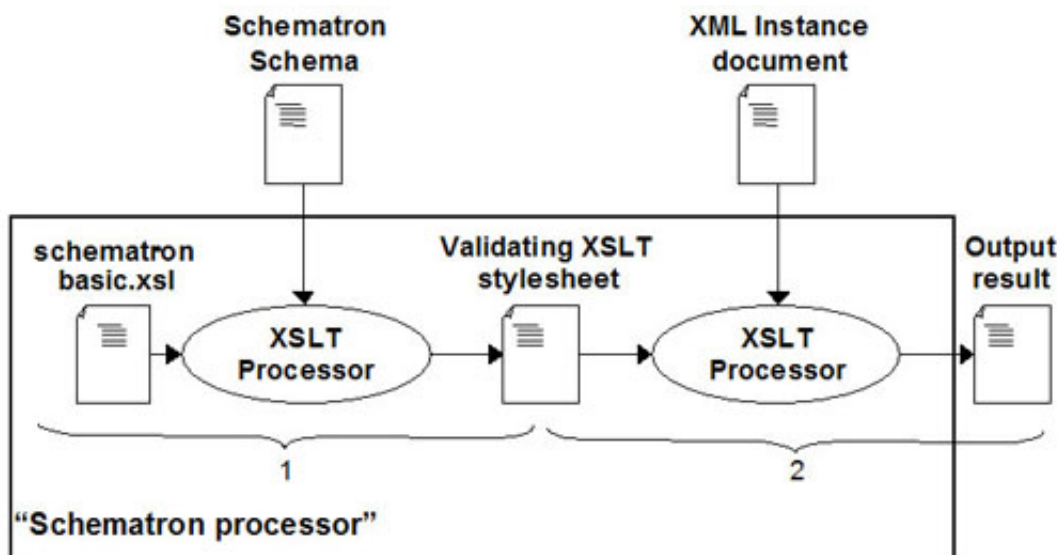


of Schematron. Since Schematron is still a fairly young schema language, many implementations differ in functionality and typically most XPath implementations only implement a subset of the Schematron functionality.

### Schematron processing using XSLT

Schematron processing using XSLT is trivial to implement and works in two steps:

1. The Schematron schema is first turned into a validating XSLT stylesheet by transforming it with an XSLT stylesheet provided by Academia Sinica Computing Centre. These stylesheets (*schematron-basic.xsl*, *schematron-message.xsl* and *schematron-report.xsl*) can be found at the [Schematron site \(http://www.ascc.net/xml/schematron/1.5/\)](http://www.ascc.net/xml/schematron/1.5/) and the different stylesheets generate different output. For example, the *schematron-basic.xsl* is used to generate simple text output as in the example already shown.
2. This validating stylesheet is then used on the XML instance document and the result will be a report that is based on the rules and assertions in the original Schematron schema.



This means that it is very easy to setup a Schematron processor because the only thing needed is an XSLT processor together with one of the Schematron stylesheets. Here is an example of how to validate the example used above where the XML instance document is called *Person.xml* and the Schematron schema is called *Person.sch*. The example use [Saxon \(http://saxon.sourceforge.net/\)](http://saxon.sourceforge.net/) as an XSLT processor:

```
>saxon -o validate_person.xsl Person.sch schematron-basic.xsl
```

```
>saxon Person.xml validate_person.xsl
```

```
From pattern "Check structure":
```

```
From pattern "Check co-occurrence constraints":
```

```
Assertion fails: "If the Title is "Mr" then the gender of the
person must be "Male"." at
    /Person[1]
    <Person Title="Mr">...</>
```

## ISO Schematron

Version 1.5 of Schematron was released in early 2001 and the next version is currently being developed as an ISO standard. The new version, ISO Schematron, will also be used as one of the validation engines in the DSDL (<http://www.dSDL.org/>) (Document Schema Definition Languages) initiative.

ISO Schematron evaluates the functionality implemented in existing implementations of Schematron 1.5. Functionality that is not implemented at all or only in a few implementations will be evaluated for removal while some requested features will be added. Some of these new features are briefly explained below, but it should be noted that these changes are not final. More information can be found in the Schematron upgrade document (<http://www.topologi.com/resources/schematronUpgrades.html>).

### Include mechanism

An include mechanism will be added to ISO Schematron that will allow a Schematron schema to include Schematron constructs from different documents.

### Variables using <let>

In Schematron it is common for a rule to contain many assertions that test the same information. If the information is selected by long and complicated XPath expressions, this has to be repeated in every assertion that uses the information. This is both hard to read and error prone.

In ISO Schematron a new element `let` is added to the content model of the `rule` element that allows information to be bound to a variable. The `let` element has a `name` attribute to identify the variable and a `value` attribute used to select the information that should be bound to the

variable. The variable is then available in the scope of the rule where it is declared and can be accessed in assertion tests using the \$ prefix.

For example, say that a simple `time` element should be validated so that the value always match the HH:MM:SS format where  $0 \leq \text{HH} \leq 23$ ,  $0 \leq \text{MM} \leq 59$  and  $0 \leq \text{SS} \leq 59$ :

```
<time>21:45:12</time>
```

Using the new `let` element this can be implemented like this in ISO Schematron:

```
<sch:rule context="time">
  <sch:let name="hour" value="number(substring(.,1,2))"/>
  <sch:let name="minute" value="number(substring(.,4,2))"/>
  <sch:let name="second" value="number(substring(.,7,2))"/>

  <!-- CHECK FOR VALID HH:MM:SS -->
  <sch:assert test="string-length(.)=8 and substring(.,3,1)=':' and
substring(.,6,1)=':'">The time element should contain a time in the
format HH:MM:SS.</sch:assert>
  <sch:assert test="$hour>=0 and $hour<=23">The hour must be a
value between 0 and 23.</sch:assert>
  <sch:assert test="$minute>=0 and $minute<=59">The minutes must
be a value between 0 and 59.</sch:assert>
  <sch:assert test="$second>=0 and $second<=59">The second must
be a value between 0 and 59.</sch:assert>
</sch:rule>
```

### <value-of> in assertions

A change requested by many users is to allow `value-of` elements in the assertions so that value information can be shown in the result. The `value-of` element has a `select` attribute specifying an XPath expression that selects the correct information.

In the above schema the assertion that for example checks the hour could then be written so that the output result contain the erroneous value:

```
<sch:assert test="$hour>=0 and $hour<=23">Invalid hour: <sch:value-
of select="$hour"/>. The value should be between 0 and 23.
</sch:assert>
```

The following instance

```
<time>25:45:12</time>
```

would then generate this output:

```
Assertion fails: "Invalid hour: 25. The value should be between 0 and 23."
```

## Abstract patterns

Abstract patterns are a very powerful new feature that allows the user to identify a specific pattern in the data and make assertions about this pattern. If we keep to the example above the abstract pattern that should be validated is the definition of a time with three parts: hour, minute and second. In ISO Schematron an abstract pattern like the following can be written to validate this time abstraction:

```
<sch:pattern name="Time" abstract="true">
  <sch:rule context="$time">
    <sch:assert test="$hour>=0 and $hour<=23">The hour must be
a value between 0 and 23.</sch:assert>
    <sch:assert test="$minute>=0 and $minute<=59">The minutes
must be a value between 0 and 59.</sch:assert>
    <sch:assert test="$second>=0 and $second<=59">The seconds
must be a value between 0 and 23.</sch:assert>
  </sch:rule>
</sch:pattern>
```

Instead of validating the concrete elements used to define the time this abstract pattern instead work on the abstraction of what makes up a time: hours, minutes and seconds.

If the XML document use the below syntax to describe a time

```
<time>21:45:12</time>
```

the concrete pattern that realises the abstract one above would look like this:

```
<sch:pattern name="SingleLineTime" is-a="Time">
  <sch:param formal="time" actual="time"/>
  <sch:param formal="hour" actual="number(substring(.,1,2))"/>
  <sch:param formal="minute" actual="number(substring(.,4,2))"/>
```

```
<sch:param formal="second" actual="number(substring(.,7,2))"/>
</sch:pattern>
```

If the XML instead uses a different syntax to describe a time the abstract pattern can still be used for the validation and the only thing that need to change is the concrete implementation. For example, if the XML looks like this

```
<time>
  <hour>21</hour>
  <minute>45</minute>
  <second>12</second>
</time>
```

the concrete pattern would instead be implemented as follows:

```
<sch:pattern name="MultiLineTime" is-a="Time">
  <sch:param formal="time" actual="time"/>
  <sch:param formal="hour" actual="hour"/>
  <sch:param formal="minute" actual="minute"/>
  <sch:param formal="second" actual="second"/>
</sch:pattern>
```

This means that the abstract pattern that performs the actual validation will stay the same independent of the actual representation of the data in the XML document.

The include mechanism makes it possible to define a separate Schematron schema that defines the validation rules as abstract patterns. Multiple "concrete schemas" can then be defined for each instance document that uses a different syntax for the abstractions. Each of these "concrete schemas" simply includes the schema with the abstract patterns and defines the mapping from the abstraction to the concrete elements.