O'REILLY® Short Cuts

Schematron

By Eric van der Vlist

Copyright © 2007 O'Reilly Media, Inc.
ISBN: 978-0-596-52771-6
Released: March 23, 2007

Schematron is a rule-based XML schema language, offering flexibility and power that W3C XML schema, RELAX NG, and DTDs simply can't match.

You need Schematron and can't settle for other languages if you have to check rules that go beyond checking the document structures (i.e., checking that an element `bar` is included in element `foo`) and their datatypes. Schematron is the right tool for checking conditions such as "`startDate` is earlier than or equal to `endDate`."

Schematron is also the right tool to use if you have to raise user-friendly error messages rather than depend on error messages that are generated by a schema processor and that are often obscure.

Schematron builds on XPath. You will need to understand XPath to to get the most from Schematron.

TABLE OF CONTENTS

SCHEMATRON 1

INTRODUCTION 2

SCHEMATRON, XSLT, AND XPATH 2

A SHORT HISTORY OF XSLT, XPATH, AND FRIENDS 2

WHICH SCHEMATRON DO YOU NEED? 4

SCHEMATRON 1.5 4
ISO SCHEMATRON 5
OTHER SCHEMATRON VERSIONS 6

LET'S START RULING 6

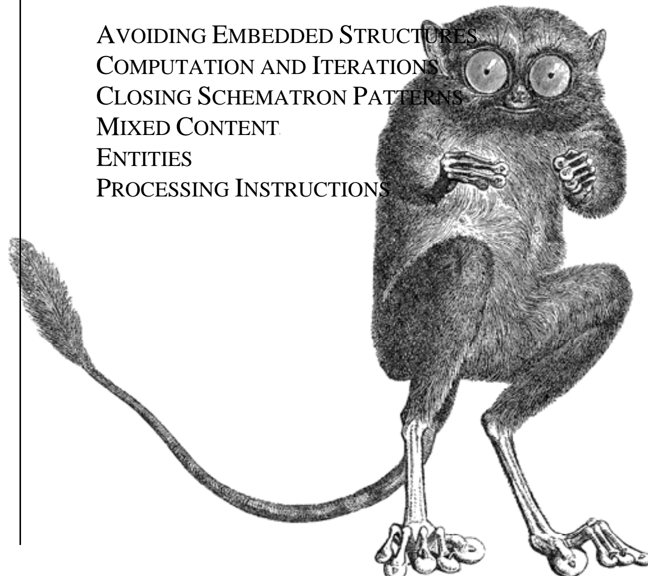
SETTING THE CONTEXT 6
TESTING 8
RICHER DIAGNOSTICS 10
ASSERTIONS VERSUS DIAGNOSTICS 12

ASSEMBLING RULES IN A SCHEMA 12

SCHEMATRON SCHEMAS 13
WHAT DOES THIS SCHEMA MEAN? 13
NAMESPACES 14
KEYS 16
PHASES 18
VARIABLES (ISO SCHEMATRON ONLY) 19
ABSTRACT RULES 21
ABSTRACT PATTERNS (ISO SCHEMATRON ONLY) 23
SCHEMA FRAGMENT INCLUSION (ISO SCHEMATRON) 24
FLAGS (ISO SCHEMATRON ONLY) 26
SIMPLE VALIDATION REPORTING LANGUAGE (ISO SCHEMATRON ONLY) 26

COOKBOOK 26

AVOIDING EMBEDDED STRUCTURES 26
COMPUTATION AND ITERATIONS 28
CLOSING SCHEMATRON PATTERNS 31
MIXED CONTENT 34
ENTITIES 40
PROCESSING INSTRUCTIONS 44



Introduction

Schematron is a simple tool that anyone working with XML should have in his toolbox. Its basic structures are basic, and its flexibility is immense.

If you are a W3C XML Schema user, you will find Schematron most useful in working around W3C XML Schema limitations, including issues such as checking for a specific root element, forbidding `xsi:type` attributes, dealing with co-occurrence constraints and handling unordered content models.

If you are a RELAX NG user, you will find Schematron especially suitable for implementing keys and key references in a very flexible way.

In both cases, Schematron will also help you keep your schemas simpler and define "business rules," such as checking that a start date is earlier than an end date.

If you are familiar with XPath but don't know any schema language, you'll be happy to learn that Schematron is based on XPath, and you will find Schematron much easier to learn than any other schema language.

In any case, you will appreciate Schematron's unique ability to let you define your own user-friendly error messages rather than having to rely on obscure error messages generated by a schema processor.

Schematron, XSLT, and XPath

As it will become clear in the rest of this article, Schematron is best described as a "hosting" language that defines rules that rely on a query language—Schematron itself defines only the logical structure of the tests to perform, and you rely on a query language to define the context of the tests to perform and the tests themselves.

To be used within Schematron, a query language needs to be able to select nodes within XML documents—this is the scope of XPath. However, XPath is hardly a standalone language, and a number of XPath versions and profiles have been defined. These flavors of XPath have a different set of features.

A Short History of XSLT, XPath, and Friends

Schematron relies on query languages to express its rules, so before we start discussing it, its versions, and its history, you have to understand some of the history of XSLT, XPath, and related. If you already know everything about these technologies, feel free to skip this section.

XSLT 1.0 and XPath 1.0 were developed by the W3C XSL Working Group to be, together with XSL-FO, a more powerful alternative to CSS stylesheets for formatting XML documents.

The definitions of these three recommendations is as follows:

XSL-FO (FO stands for Formatting Objects)
: A presentation format usually generated by XSLT transformations.

XSLT
: A general-purpose XML transformation language that operates on XML sources. It can be used to produce not only XSL-FO but any other XML format—even text or HTML.

XPath
: A query language used by XSLT to access the different information items that compose XML documents. XPath was separated from the XSLT recommendation with the expectation that it would be useful—and used—by other specifications. So, XPath 1.0 is used in XPointer, W3C XML Schema, and of course, Schematron.

Warning

As you can imagine, XPath plays a very important role in Schematron schemas. This short cut does not include an introduction to XPath, and if you don't know it, you should learn its basics (you'll find a number of good tutorials and excellent books on the subject) before you can fully understand this article.

Because of the split between XSLT and XPath, some functions that have been considered specific to XSLT are defined in the XSLT recommendation while the other functions are defined in XPath. This leaves us with two slightly different query languages, both of which can be used in Schematron: XPath 1.0 (which includes all the functions defined in the XPath 1.0 recommendation) and XSLT 1.0 (which includes the functions such as `document()` or `generate-id()` that are defined in XSLT 1.0).

XSLT 1.0 includes a mechanism for using extension functions identified by namespaces. Most XSLT implementations support a number of extension functions, such as the well-known `node-set()`,to transform result tree fragments into node sets. To improve interoperability between implementations, the EXSLT group (see **http://exslt.org**) has proposed using a shared namespace for common extensions. This creates a third candidate for query languages that can be used within Schematron.

XSLT implementations generally store in memory the complete XML documents that are to be manipulated. To avoid that overhead and facilitate "streaming" implementations, the STX project defines a language derived from XSLT that can

be implemented in a “streamable” way. This is a good candidate for enabling streamable implementations of Schematron.

The W3C recently published a second version of XPath and XSLT. Because these languages are no longer tied to stylesheet kind of usages, this has been a joint work between the XSL and XQuery Working Groups, and XPath 2.0 serves as the basis for XSLT 2.0 and XQuery 1.0. This makes three new candidates (XPath 2.0, XSLT 2.0 and XQuery 1.0) for query languages that can be used within Schematron.

XPath 2.0 and XSLT 2.0 are recent developments. You might think that XPath 2.0 and XSLT 2.0 would, sooner or later, deprecate XPath 1.0, XSLT 1.0, and EXSLT, and simplify the landscape, but things are not that easy. There are significant, complicated changes between XPath 1.0 and XPath 2.0, such as the support that the 2.0 specifications provide for W3C XML Schema. It is not likely that the 2.0 versions will replace XPath 1.0 and XSLT 1.0 anytime soon, if indeed it ever happens.

For the near future, this leaves us with seven possible query languages that can be used with Schematron: XPath 1.0, XSLT 1.0, EXSLT, STX, XPath 2.0, XSLT 2.0, and XQuery 2.0.

Which Schematron Do You Need?

Schematron is an open standard, and most of its implementations are open source. When you go shopping to download your free implementation of Schematron over the Internet, you'll find out that there are several flavors available!

Schematron was developed in Taiwan in late 1999 by Rick Jelliffe. For the next five years, many different implementations were created for different platforms, and the community of Schematron users and enthusiasts contributed many enhancements and implementation methods. The technique of using XPath assertions over data models expressed as XML has been widely adopted for many other problem areas, including programming, checking for Java, and semantic data using the RDF data model.

Schematron 1.5

The most common version of Schematron is known as “Schematron 1.5”.

Schematron 1.5 was built from a proposal by Rick Jelliffe and takes into account the number of contributions from a community of Schematron enthusiasts. Its home is on the Academia Sinica Computing Centre web site at **http://xml.ascc.net/schematron/**. It was designed to be implemented easily as a two-stage XSLT 1.0 transformation:

1. A first generic transformation that can be downloaded from the Schematron web page transforms Schematron schemas into XSLT transformations.
2. These transformations are run against instance documents to validate them.

This design decision means that you can use Schematron 1.5 wherever you have a XSLT processor available. That deployment simplicity has been one of the key reasons for the success of Schematron.

Schematon's style of design and implementation based on active users' and developers' communities is more similar to open source projects than to committee specifications. This has greatly contributed to its maturity and practicality: we can say that Schematron 1.5 has found the sweet spot between usability and ease of implementation.

As Schematron 1.5 is by far more common than any other version, this short cut focuses on this version, and unless explicitly mentioned, the examples presented in this article work with Schematron 1.5.

Warning

Schematron 1.5 supports XSLT 1.0 as a query language only. If you want to use another query language, you need to use ISO Schematron, discussed next.

ISO Schematron

Schematron has also been standardized by the ISO/DSDL project as ISO/IEC 19757 – Part 3.

The standard is available free from ISO as a Publicly Available Specification (**http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/ITTF.htm**) known as *ISO Schematron*. ISO Schematron consolidates ideas from the earlier and experimental versions of Schematron.

Even though the language is very similar, ISO Schematron differs from Schematron 1.5 in a number of critical ways:

- A new namespace URI is used in ISO Schematron.
- The ISO standard doesn't state that Schematron should be implementable as a two-stage XSLT transformation. However, it might still be possible to do so, and Rick Jelliffe is working on such an implementation. The future will tell, but it's not a formal requirement any longer.
- Schematron 1.5 limited the language used to define Schematron constraints to XPath 1.0 with XSLT 1.0 extension functions. ISO Schematron introduces a principle of "query language binding," which supports XPath 1.0, XSLT 1.0,

XSLT 1.1, EXSLT, STX, XPath 2.0, XSLT 2.0, and XQuery 1.0 (this list can be extended if needed).

- Variables and abstract patterns are now new features.
- The names of some attributes have been changed to improve the coherence of the language.

ISO Schematron is promising but still very young. Some of its new features need some smoothing, and there are few implementations available yet.

In this short cut, features or examples requiring ISO Schematron are explicitly labeled as such.

Other Schematron Versions

There are two other Schematron versions mentioned on the Internet that are not covered in this article:

- Schematron 1.3 is an older version, which you shouldn't have to bother with.
- Schematron 1.6 is an intermediary version between Schematron 1.5 and ISO Schematron. Because it's designed to be transitional, it isn't expected to be widely used.

Let's Start Ruling

You're probably impatient to see Schematron at work and want to start defining your first Schematron rules right now.

Setting the Context

Rules are the basic building block of Schematron schemas. Like XSLT templates, Schematron rules set the "context node" from which relative expressions are evaluated. They also act as containers to create groups of tests.

Warning

Although rules are similar to XSLT templates in the way that they set context nodes, you will see in a later section that the algorithm used to run across a document and select the rule matching each node is totally different from the algorithm used by XSLT to apply and select templates.

To specify which nodes a rule must be applied on, use the `context` attribute, as in the following:

```
<rule context="book">
<!--
Insert tests to be done on "book" elements here
-->
.../...
```

```
</rule>
```

The XPath expression in the `context` attribute can be more complex than this, and it can include all the constructions allowed in an XSLT pattern. (XSLT patterns are the XPath expressions that are allowed in `match` attributes.)

We could, for instance, write `context="book[@id]"` to define a rule that applies only to book elements with an `id` attribute, or write `context="*[@id]"` to define a rule that applies to any element with an `id` attribute, or write `context="library/book"` to define a rule that applies to book elements with a library parent element.

Warning

Schematron 1.5 applies rules to the following node types:

- Elements (`*`)
- Text nodes (`text()`)
- Root node (`/`)
- Comments (`comment()`)
- Processing instructions (`processing-instruction()`)

But it doesn't apply to attributes (`@*`).

In other words, a rule defined with `context="@id"` will never be fired by a Schematron 1.5 processor.

ISO Schematron applies rules to:

- Elements (`*`)
- Attributes (`@*`)
- Root node (`/`)
- Comments (`comment()`)
- Processing instructions (`processing-instruction()`)

But it doesn't apply rules to text nodes.

Note

Other schema languages such as W3C XML Schema and RELAX NG completely ignore XML comments and processing instructions.

If you need to validate comments or processing instructions (for instance, to check that a stylesheet is properly attached to a document), Schematron is the right tool to use!

Testing

Now that you've set the context, you can start testing things. Schematron gives you two different elements for testing something:

`assert`
: Is an assertion that generates a message when a condition is not met.

`report`
: Reports a message when a condition is met. The message may be an error, or it may be used positively to report information about structures found.

This is luxurious. We need only one of these elements, but it's handy to have two and saves us a lot of negations that would make our tests less readable.

These two elements expect a `test` attribute in which the test is expressed as an XPath expression. To test that you have an `id` attribute and an `isbn` number, and that the `id` attribute is derived from the `isbn` member by adding a prefix "b," write the following:

```
<rule context="book">
<assert test="@id">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be the
ISBN number with a prefix "b"</assert>
</rule>
```

The first two `assert` elements perform tests that can easily be done by any schema language: the first checks that there is an `id` attribute ,and the second checks that there is an `isbn` element. These asserts rely on the fact that `test` attributes are converted to boolean values following the casting rules defined in the XPath recommendations: `@id` and `isbn` are node lists, and their conversion into booleans is `true` if, and only if, these node lists contain at least one node.

Note that the second `assert` doesn't even test to make sure that there isn't more than one `isbn` element. To do so, you'd have to use the XPath `count()` function and write something such as `test="count(isbn) = 1"`.

The third `assert` element is more interesting. This is a typical example of business rules. These are easy enough to implement with Schematron but would be impossible to implement with grammar-based schema languages such as W3C XML Schema or RELAX NG.

These `assert` elements could all have been replaced by equivalent `report` elements such as `<report test="not(@id)">Missing "id" attribute.</report>`.

Choosing between `assert` and `report` is really a matter of style. You can argue indefinitely whether in `assert` is more readable because these is no negation in its `test` attribute or whether `report` is more natural because there is a clear relation between the test that says in XPath "we have no `id` attribute" and the error message that says the same thing in plain English.

Richer Diagnostics

In the previous examples, you wrote text to display when errors are raised in the `assert` and `report` elements themselves.

Note

Other schema languages such as W3C XML Schema and RELAX NG do not let the user define the text of his error messages. These messages are generated by the schema processors and can be, depending on the tool and the context, very hard to read—even for geeks. If it is very important for you to generate user-friendly error messages that end users can understand, consider using Schematron for this very reason.

It is often useful to copy information from the instance documents to make these error messages more useful.

For instance, you could write the following rule to test that `id` attributes are unique among all elements in a document:

```
<rule context="*[@id]">
<report test="preceding::*/@id = @id">Duplicated id attribute.</report>
</rule>
```

That would work fine, but it doesn't even say which element the error has been detected. To display the name of the current element, use a `name` element:

```
<rule context="*[@id]">
<report test="preceding::*/@id = @id">Duplicated id attribute in a
"<name/>" element.</report>
</rule>
```

As you might have guessed, the `name` element is replaced by the name of the context node in the error message, giving you something such as `"Duplicated id attribute in a "character" element."`

That's an improvement, but we could be still more helpful by mentioning the value of the duplicated `id` attribute. This isn't possible within the `assert` or `report` elements themselves, but you can write more detailed messages using `diagnostic` elements. These elements are detached from the `assert` and `report` elements and linked through `id` attributes. (It is thus possible to attach several `diagnostic` elements to an `assert` or `report` element and also to attach the same `diagnostic` to several `assert` and `report` elements).

To add a `diagnostic` to the `report` element, add a `diagnostics` attribute pointing to the diagnostic itself:

```
<rule context="*[@id]">
<report test="preceding::*/@id = @id" diagnostics="idAttribute ">Duplicated
id attribute in a "<name/>" element.</report>
</rule>
.../...
```

```
<diagnostic id="idAttribute"> The value of the duplicated id attribute
is "<value-of select="@id"/>".</diagnostic>
```

Within a `diagnostic` element, you can use `value-of`. This has the same meaning as the XSLT `xsl:value-of` instructions.

Warning

Both the ISO Schematron and Schematron 1.5 specifications clearly state that implementations are free to ignore `diagnostic` elements. The minimal XSLT reference implementation of Schematron 1.5 does ignore them, making this subject somewhat academic for its many users.

Warning

Although the Schematron 1.5 schema nominally forbids `value-of` elements in `assert` and `report` elements and `name` elements within `diagnostic` elements, the XSLT reference implementation of Schematron 1.5 does allow and support `value-of` elements in `assert` and `report` elements.

Relying on a feature marked as "Forbidden" in the specifications is always perilous. If you want to use it, do so at your own risk!

Note

ISO Schematron allows `value-of` elements within `assert`, `report`, and `diagnostic` elements but still forbids `name` elements within `diagnostic` elements.

This remaining restriction isn't a big deal because `name` elements can be replaced by `value-of` elements that use the XPath `name()` function.

One of the reasons you might want to attach multiple diagnostic elements to a single `report` or `assert` is to provide messages in different languages:

```
<rule context="*[@id]">
<report test="preceding::*/@id = @id" diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
.../...
<diagnostic id="idAttribute-en" xml:lang="en">The id attribute
"<value-of select="@id"/>" is duplicated in element
"<value-of select="name()"/>".</diagnostic>
<diagnostic id="idAttribute-fr" xml:lang="fr">L'attribut id
"<value-of select="@id"/>" de l'élément
"<value-of select="name()"/> est dupliqué".</diagnostic>
```

ISO Schematron also provides various attributes to give user interfaces more to work with, such as:

`icon`
: Associates the message with an informative icon.

`see`
: Associates the message with a link to a web resource.

`fpi`
: Provides a system-independent identifier for the assertion.

`role`
: Associates the assertion with a simple label that can be used for searching and sorting.

Assertions Versus Diagnostics

In this book, assertions contain simple error messages. This is appropriate for small-scale ad hoc validation. However, when Schematron is used for larger projects and for public specifications, it makes sense to adopt a more declarative style where:

- The `assertion` elements contain only positive natural language statements about what should occur in a context, optimally with the business reason (justification) explaining why the constraint exists, and preferably expressed in terms of the problem domain rather than in terms of the XML markup.
- The `diagnostics` elements contain more detailed statements to help locate and fix the problem, preferably expressed in terms of the XML markup.

This allows the most flexibility in the use of Schematron schemas. One way to test whether best practice is being followed is to consider "Would a domain expert or manager with only limited XML exposure be able to understand the constraints in the schema, if presented with the text of all the assertions in a bullet point form?"

One way to achieve this best practice is to adopt the following template for assertions: "An X should have a Y (because Z requires it)." If you are in an environment where traceability is not important, then just "An X should have a Y" is enough. Contrast this with the error message style "Error: this X does not have Y," which is not really an "assertion" at all.

Assembling Rules in a Schema

You've seen how to write `rule` and `diagnostic` elements. Now it's time to assemble these elements in a complete schema.

Schematron Schemas

In brief, `rule` elements go into `pattern` elements and `diagnostic` elements go into `diagnostics` elements, and all of those are packed within a `schema` element. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="main">
<rule context="book">
<assert test="@id">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be
the ISBN number with a prefix "b" </assert>
</rule>
<rule context="*[@id]">
<report test="preceding::*/@id = @id"
diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
</pattern>
<diagnostics>
<diagnostic id="idAttribute-en" xml:lang="en">The id attribute
"<value-of select="@id"/>" is duplicated in element
"<value-of select="name()"/>".</diagnostic>
<diagnostic id="idAttribute-fr" xml:lang="fr">L'attribut id
"<value-of select="@id"/>" de l'élément
"<value-of select="name()"/> est dupliqué".</diagnostic>
</diagnostics>
</schema>
```

Note

The namespace URI for Schematron 1.5 is **http://www.ascc.net/xml/schematron**.

The namespace URI for ISO Schematron is **http://purl.oclc.org/dsdl/schematron**.

What Does This Schema Mean?

This seems intuitive enough, but there are still important questions that need an answer. For instance, in the previous example, `book` elements obviously match the two rules (they are `book` elements and they are also elements with an `id` attribute). Which rule will be applied to them? In fact, we need to explain how Schematron processors are expected to understand this schema!

The ISO Schematron specification defines this semantics is a declarative way that is equivalent to the more programmatic style used in the Schematron 1.5 specification. Both can be summarized by saying that a Schematron processor acts as if it followed by this algorithm:

- Loop over the nodes in a document.
- For each node, loop over the patterns.
- For each `pattern`, find the first `rule` matching the current node.

- Test that all the `asserts` and `reports` in this rule are met for the current node.

The specifications say that the order in which this is performed is not fixed. The only things that are guaranteed are that the following:

- A document will not be considered valid until all these tests have been done.
- For each node, only the first matching `rule` of every `pattern` will be checked.

This second bullet point provides the answer to the question about `book` elements: `book` elements match your first rule. The second rule will never be evaluated for these elements. If you want the previous rules to be independent, with both tested on nodes that match their `context` attribute, you need to locate them within two different `pattern` elements, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="main">
<rule context="book">
<assert test="@id">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be
the ISBN number with a prefix "b" </assert>
</rule>
</pattern>
<pattern name="keys">
<rule context="*[@id]">
<report test="preceding::*/@id = @id"
diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
</pattern>
<diagnostics>
<diagnostic id="idAttribute-en" xml:lang="en">The id attribute
"<value-of select="@id"/>" is duplicated in element
"<value-of select="name()"/>".</diagnostic>
<diagnostic id="idAttribute-fr" xml:lang="fr">L'attribut id
"<value-of select="@id"/>" de l'élément
"<value-of select="name()"/> est dupliqué".</diagnostic>
</diagnostics>
</schema>
```

Now that the second rule is in a different pattern, it will be checked against elements with `id` attributes even whenthey match rules from other patterns (as with the `book` elements).

Namespaces

To be used in XPath expressions, namespaces must be declared using an `ns` element.

To validate an instance document such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<library xmlns="http://ns.xmlschemata.org/examples/">
<book id="b0836217462" available="true">
<isbn>0836217462</isbn>
.../...
</book>
</library>
```

you would write a Schematron schema such as:

```
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://ns.xmlschemata.org/examples/" prefix="x"/>
<pattern name="main">
<rule context="x:book">
<assert test="@id">Missing "id" attribute.</assert>
<assert test="x:isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', x:isbn)">The "id" attribute should be the ISBN number
with
a prefix "b" </assert>
</rule>
</pattern>
</schema>
```

Warning

When writing XPath expressions on documents with namespaces, you must remember that:

- XPath has no notion of a default namespace (elements from a namespace must always be prefixed in XPath expressions as done in the preceding example).
- The default namespace doesn’t apply to attributes. (Attributes without prefixes in the instance document must not be prefixes in XPath expressions.)

Note

Languages such as XSLT and W3C XML Schema use XML namespace declarations (through `xmlns` attributes) to define the prefixes used in the content of their attributes.

This practice, known as “QNames” (for *Qualified Names*), is very controversial because it creates a dependency between the markup and the document content, and you can’t change a namespace prefix in the markup without changing it in the content of the document.

Schematron has chosen to define its own mechanism to define its namespace prefix (through the `ns` element), which is a much safer practice even though you might find it weird to have to use a specific mechanism to define the namespaces manipulated by your schemas.

Also note that XPath and W3C XML Schema vary in whether an unprefixed name will be in the default namespace. Schematron uses XPath, which makes things simple: no prefix, no namespace.

Keys

If you are familiar with XSLT, then you’ll probably recognize the benefits of using keys. Keys improve speed and shorten XPath expressions in complex situations such as grouping data. XSLT keys can also bring these advantages to Schematron schemas.

One obvious use case is to optimize the identity constraint that you implemented using the `preceding` XPath axis in a previous example as shown here:

```
<pattern name="keys">
<rule context="*[@id]">
<report test="preceding::*/@id = @id">
Duplicated id in element "<name/>".
</report>
</rule>
</pattern>
```

The simplest schema that comes to mind to check this constraint is:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="id">
<rule context="*[@id]">
<key name="id" path="@id"/>
<assert test="count(key('id', @id)) = 1">Duplicated id in element "<name/>".</assert>
</rule>
</pattern>
</schema>
```

Note

The syntax of the Schematron `key` element is slightly different from the syntax of the XLST `xsl:key` declaration.

The equivalent `xsl:key` declaration would be:

```
<xsl:key name="id" match="*[@id]" use="@id"/>
```

Two out of the three attributes are different, but their content and semantics are the same :

- The `name` attribute is the same in both languages.
- The XSLT `use` attribute becomes the Schematron `path` attribute.
- The Schematron `context` attribute of the current `rule` is used as the XSLT `match` attribute.

This new version is both simpler and faster to execute.

It's not exactly equivalent to the previous version because it is raising a different error for each occurrence of a duplicated `id` attribute. Your previous version was raising a different error for the second and following occurrences of a duplicated `id`.

If you want to raise a single error for all occurrences of a duplicated key, use the XSLT 1.0 technique known as the Muenchian technique:

```
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="id">
<rule context="*[@id]">
<key name="id" path="@id"/>
<assert
test="count(key('id', @id)) = 1 or count(.|key('id', @id)[1]) = 2">
Duplicated id in element "<name/>".</assert>
</rule>
</pattern>
</schema>
```

This doesn't appear very readable, but if you take a closer look, you'll see that this isn't as horrible as it seems!

The idea is to add an `or` clause in the `assert` element to avoid raising errors after the `assert` elements first occurance. Let's look at this `or` clause. It has to distinguish whether or not this element is the first with this key value. A traditional way to check if two nodes are the same in XPath 1.0 is to test if they make one or two nodes when included in a node set. Here, we apply this technique and count the number of nodes in a node set composed of the current node (`.`) and the first node identified by this key (`key('id', @id)[1]`). The count after the `or` (`count(.|key('id', @id)[1])`) will thus return 1 if the current node is the first with this key. It will return 2 if that's not the case.

Warning

There is no `key` element in ISO Schematron: the ISO standards committee felt that `key` was a feature specific to XSLT implementations and not query-language-neutral. Fortunately, this doesn't mean that you can't use keys in ISO Schematron!

The specification mentions that: "The XSLT key element may be used, in the XSLT namespace, before the pattern elements."

This means that while you can't use the `key` element as shown in the previous examples, you can use it with its XSLT syntax before the `pattern` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="id" match="*[@id]" use="@id"/>
<pattern name="id">
<rule context="*[@id]">
<assert test="count(key('id', @id)) = 1">Duplicated id in element "<name/>".</assert>
</rule>
</pattern>
</schema>
```

Phases

Within a schema, there might be some rules that you don't want to always apply, because they are very heavyweight and long to test, you have different classes of documents with slightly different sets of constraints, or you want to allow incomplete documents at some stages in your processing.

If, for instance, you implement both key and key reference checks, you might be interested in switching off key reference checking for incomplete documents. This is a typical use case for phases.

To use this feature, you need to do two things:

- Locate the rules for checking keys and key references within two different patterns and give these rules `id` attributes.
- Define our phases using the `phase` elements located before the `pattern` elements.

The resulting schema would look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<phase id="full-integrity">
<active pattern="keys"/>
<active pattern="keyrefs"/>
</phase>
<phase id="keys-only">
<active pattern="keys"/>
</phase>
```

```
<pattern name="keys" id="keys">
<rule context="*[@id]">
<key name="id" path="@id"/>
<assert test="count(key('id', @id)) = 1">Duplicated id in element "<name/>".</assert>
</rule>
</pattern>
<pattern name="keyrefs" id="keyrefs">
<rule context="*[@idref]">
<assert test="count(key('id', @idref)) = 1">Reference not found in element
"<name/>".</assert>
</rule>
</pattern>
</schema>
```

When we run this schema without mentioning anything special to your Schematron processor, it considers that all the patterns are active and executes each of them, but if you specify a phase (either *full-integrity* or *keys-only*) it will only use the patterns marked as *active* in this phase.

Warning

The identification of the patterns is done using their `id` attributes. The `name` attribute is considered to be a human-readable name rather than an identifier.

Note that the way you define a phase is implementation-specific. Command-line implementations typically use a command-line parameter to do so.

Variables (ISO Schematron Only)

ISO Schematron supports XSLT-like "variables." As with XSLT variables, these variables can't be updated, and the assignment is also their definition.

The scope of a variable is the schema, phase, pattern, or rule within which it has been defined. As in XSLT, variables are most useful to simplify complex XPath expressions.

A typical example is date validation. If you need to validate an ISO date with the format "YYYY-MM-DD," you will find that Schematron 1.5 quickly gives you XPath expressions that are unreadable.

Even if you restrict yourself to just checking that you've got "four digits, followed by a "-", followed by two digits, followed by a "-", followed by two digits—which can already catch classical errors such as people using a non-ISO format for a valid date—you can find simple hacks. I often use the following rule:

```
<rule context="born|died">
<report test="string-length(.) != 10">Wrong date size (dates should be 10 character
long).</report>
<report test="translate(substring(., 1, 4), '0123456789', '') != ''">Wrong date format
(dates should start with four digits for the year)</report>
<report test="substring(., 5, 1) != '-' ">Wrong date format</report>
<report test="translate(substring(., 6, 2), '0123456789', '') != ''">Wrong date format
(dates should have two digits for the month)</report>
<report test="substring(., 8, 1) != '-' ">Wrong date format</report>
```

```
<report test="translate(substring(., 9, 2), '0123456789', '') != ''">Wrong date (dates
should end with two characters for the day)</report>
</rule>
```

Troubles begin when you want to do more elaborate tests such as checking days versus months. Even without bothering to test leap years, you rapidly end up with a lot of repetitions such as the following:

```
<report test="substring(., 9, 2) =0 or substring(., 9, 2) > 31">Wrong date (days
should
be between 1 and 31)</report>
<report
test="(substring(., 6, 2) = '04' or substring(., 6, 2) = '06' or substring(., 6, 2) =
'09' or substring(., 6, 2) = '11') and substring(., 9, 2) > 30"
>Wrong date (days should be between 1 and 30 in April, June, September and
November)</report>
<report test="substring(., 6, 2) = '02' and substring(., 9, 2) > 29">Wrong date (days
should be between 1 and 29 in February)</report>
```

You can make these expressions more readable using internal entities, but that still makes a lot of repetitions for the Schematron implementation that will most likely reevaluate the `substring()` function each time it finds one.

The same rule using ISO Schematron variables would be:

```
<rule context="born|died">
<let name="year" value="substring(., 1, 4)"/>
<let name="month" value="substring(., 6, 2)"/>
<let name="day" value="substring(., 9, 2)"/>
<report test="string-length(.) != 10">Wrong date size (dates should be 10 character
long).</report>
<report test="translate($year, '0123456789', '') != ''">Wrong date format (dates
should
start with four digits for the year)</report>
<report test="substring(., 5, 1) != '-' ">Wrong date format</report>
<report test="translate($month, '0123456789', '') != ''">Wrong date format (dates
should
have two digits for the month)</report>
<report test="substring(., 8, 1) != '-' ">Wrong date format</report>
<report test="translate($day, '0123456789', '') != ''">Wrong date (dates should end
with
two characters for the day)</report>
<report test="$day =0 or $day > 31">Wrong date (days should be between 1 and
31)</report>
<report
test="($month = '04' or $month = '06' or $month = '09' or $month = '11') and $day >
30"
>Wrong date (days should be between 1 and 30 in April, June, September and
November)</report>
<report test="$month = '02' and $day > 29">Wrong date (days should be between 1 and 29
in February)</report>
</rule>
```

This is much more readable, don’t you think?

Note

With an ISO Schematron processor supporting XSLT 2.0 as a query language, the preceding example could be dramatically simplified by using the XPath 2.0 `castable as` operator to test whether the element can be casted into an `xs:date` datatype.

The only tests that you would still have to do would be to check for a specific format—in our example, that would be to check that the year has four digits (`xs:date` allows any number of digits for the year), and that no time zone is specified.

That could be done using the tools that we used in the previous examples, but could also be simplified by using either the `fn:matches()` XPath 2.0 function that supports regular expressions or the EXSLT `regexp` module.

Abstract Rules

Let's say you have these two rules, shown here:

```
<rule context="book">
<assert test="@id">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be the ISBN number
with
a prefix "b" </assert>
</rule>
<rule context="*[@id]">
<report test="preceding::*/@id = @id" diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
```

A `book` element has an `id` attribute. Of course, you'd like to check the second rule for `book` elements. You've seen that, for each node in instance documents, the first matching rule is checked only, and that you can't write these two rules in the same pattern if you want to check both of them for `book` elements. You can fix that by locating the two rules in two different patterns. However, abstract rules give you a second option.

Abstract rules are rules without context attributes. They define sets of assertions and reports and are "extended" by rules that inherit their assertions and reports and that can add additional ones.

In the following case, you could define an abstract rule to implement identity checks:

```
<rule id="uniqueness" abstract="true">
<report test="preceding::*/@id = @id" diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
```

And then, you could extend this abstract rule in both rules:

```
<rule context="book">
```

```
<extends rule="uniqueness"/>
<assert test="@id">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be the ISBN number
with
a prefix "b" </assert>
</rule>
<rule context="*[@id]">
<extends rule="uniqueness"/>
</rule>
```

If you write these two rules in this order in a single pattern, the first will apply to `book` elements, and the second will apply to any other element with an `id` attribute. Both will perform the tests defined in the abstract rule.

Note

References to abstract rules are made using `extends` elements. This allows *multiple inheritance*, which means a single rule can extend more than one abstract rule.

Cascading inheritance is also possible: abstract rules can extend other abstract rules.

Warning

Despite their object-oriented metaphor, abstract rules are nothing more than macros: `extends` elements are recursively replaced by the content of the abstract rule that to which they refer.

For ISO Schematron, that statement means that the scope of variables defined in an abstract rule is not the abstract rule itself, but are the concrete rules that extend, directly or indirectly, the abstract rule.

Applied to the previous example, that means that a variable defined in the abstract rule can be used in the rule for the `book` element—for instance:

```
<rule id="uniqueness" abstract="true">
<let name="nbElementsWithThisId > 1" value="count(//*[@id=current()/@id])"/>
<report test="$nbElementsWithThisId" diagnostics="idAttribute-en idAttribute-fr "/>
</rule>
<rule context="book">
<extends rule="uniqueness"/>
<!-- Variable $nbElementsWithThisId is known here! -->
<assert test="$nbElementsWithThisId = 0">Missing "id" attribute.</assert>
<assert test="isbn">Missing "isbn" element.</assert>
<assert test="@id = concat('b', isbn)">The "id" attribute should be the ISBN number
with
a prefix "b" </assert>
</rule>
```

This is also true between abstract rules: the variables defined in an abstract rule are known in the abstract rules extended after this rule.

This situation also creates a risk of having conflicting variable definitions when extending abstract rules that have not been designed to work together. It is a good idea to avoid defining variables in abstract rules as much as possible unless you adopt a naming convention that limits the risk of having conflicting definitions (you can, for instance, prefix variable names with the identifier of the abstract rule).

These issues are the same for macro languages used in programming languages and will also be familiar to C programmers!

Abstract Patterns (ISO Schematron Only)

The abstract rules you saw in the previous section are useful, but they are nothing more than a rather crude macro language feature, and they work well when the physical structure expected by the abstract rule is the same as the one that is expected by the rules that extend them.

Abstract rules shouldn't be confused with *abstract patterns*, which is a feature specific to ISO Schematron that can be used to define patterns that can map different physical structures.

The typical example given in the ISO specification is an abstract pattern for table definitions:

```
<pattern abstract="true" id="table">
<rule context="$table">
<assert test="$row">A table has at least one row</assert>
</rule>
<rule context="$row">
<assert test="$cell">A table row has at least one cell</assert>
</rule>
</pattern>
```

This pattern is generic enough to be applied to HTML tables:

```
<pattern id="xhtml-basic-table" is-a="table">
<param name="table" value="table"/>
<param name="row" value="tr"/>
<param name="cell" value="td"/>
</pattern>
```

It also applies to CALS tables:

```
<pattern id="cals-table" is-a="table">
<param name="table" value="ctable"/>
<param name="row" value="tbody/row"/>
<param name="cell" value="entry"/>
</pattern>
```

It even applies to anything that looks more or less like a table:

```
<pattern is-a="table" id="calendar">
<param name="table" value="calendar/year"/>
<param name="row" value="week"/>
<param name="entry" value="day"/>
</:pattern>
```

In each of these cases, the abstract pattern is copied in the pattern that instantiates it, and the values of the parameters are replaced in the XPath expressions.

Schema Fragment Inclusion (ISO Schematron)

The last goody you need to see is also specific to ISO Schematron.

When you've defined a number of useful patterns, rules, and diagnostics, abstract or not, you will probably want to reuse them in different schemas. This can be done through the `include` element.

Unfortunately, this feature doesn't work as it does in other languages such as RELAX NG, XSLT, or W3C XML Schema. It does a straight inclusion of an external document that replaces the `include` element. In other words, this `include` statement has no Schematron-specific semantic and works like an external entity or an XInclude inclusion.

If you want to reuse your `table` abstract pattern, you would have to write a document with this pattern only:

```
<?xml version="1.0" encoding="UTF-8"?>
<pattern id="unique" abstract="true" xmlns="http://purl.oclc.org/dsdl/schematron">
<rule context="$item">
```

```
<report test="count($root/$item[$id=current()/$id]) > 1">Duplicated id "<value-of
select="."/>"</report>
</rule>
</pattern>
```

Then, you would include this document in schemas that use it through the `include` statement. If the name of the document with this pattern is *unique.sch*, the statement would be:

```
<include href="unique.sch"/>
```

Warning

This `include` statement is very different from the includes you may know from XML applications such as XSLT, RELAX NG, or W3C XML Schema. For these applications, `include` is about merging the contents of several transformations (or grammars or schemas): all the definitions of the document that is included are merged with the definitions of the current document.

For Schematron, `include` is a straight inclusion similar to what you can do with external entities or XInclude.

A Schematron `include` is less flexible than external entities, which support multirooted documents. This gives you the ability to group several patterns or rules in a single document, which a Schematron `include` doesn’t support. A Schematron `include` is also less flexible than Xinclude, which supports fragment identifiers. This allows you to pick one or more definitions in a schema that has multiple definitions. A Schematron `include` doesn't support this either.

The motivation of this feature is only to provide a minimally simple component for people who, for any reason, do not want to use external entities nor XInclude.

Flags (ISO Schematron Only)

Sometimes it is better to get the result of validation of a document in the form of a set of states rather than messages. To allow this, ISO Schematron provides an attribute `flag` that can appear on `assert` elements. The attribute contains a simple name, and multiple `assert` elements can potentially set the same flag. When validation is complete, a flag is considered true if any assertion with that flag fails.

Simple Validation Reporting Language (ISO Schematron Only)

ISO Schematron defines a simple report language, SVRL. Some implementations may expose the results of XML validation using SVRL. The SVRL can be organized easily into HTML or other formats.

Cookbook

Schematron is so simple that all of its basics fit in 20 pages! It is time to put this newfound knowledge into practice and get some real work done. The recipes that constitute this second part are derived from real world use cases. I hope that some of them will match cases that you have already encountered, or even better, that you need to solve right now. However, you will also run into cases that do not correspond to any of these recipes. If this happens, you should still be able to adapt the different techniques mentioned here. I invite you to go through these recipes, even if they solve issues that are different from those that you need to solve right now!

Avoiding Embedded Structures

In document-oriented applications such as XHTML, DocBook, TEI, DITA, and so on, it is often the case that some elements cannot be embedded within themselves. For instance, you probably want to forbid paragraphs that include a link within a link.

With a grammar-based schema language, the solution is to define two different content models: a content model for a paragraph and a content model for a link. These models are the same except that links are allowed in paragraphs but are forbidden within other links.

So far so good, but if you want to forbid bold within bold, you need to define four different content models: paragraph, link, bold, and bold in link, where both bold and link are forbidden. If you also want to forbid italic within italic, you need to define paragraph, link, bold, italic, bold in link, bold in italic, italic in link, bold in italic in link, and so on and so on. This is a good example of exponential blowup!

With Schematron, you need only one rule per element:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/xhtml" prefix="x"/>
<pattern name="embedding">
<rule context="x:a">
<report test=".//x:a">We don't want a's in a's!</report>
</rule>
<rule context="x:b">
<report test=".//x:b">We don't want b's in b's!</report>
</rule>
<rule context="x:i">
<report test=".//x:i">We don't want i's in i's!</report>
</rule>
</pattern>
</schema>
```

If that's still too long, the rule can be made generic and match all elements at once:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/xhtml" prefix="x"/>
<pattern name="embedding">
<rule context="x:a|x:b|x:i">
<report test=".//x:*[local-name()=local-name(current())]">We don't want <name/>'s in
<name/>'s!</report>
</rule>
</pattern>
</schema>
```

In this variant, you set the rule’s context node to match all three elements: `context="x:a|x:n|x:i"`. Now the test needs to check that you don’t find an element name that is the same than the current one. For an XML vocabulary that uses namespaces, you need to test that both the namespace URI and the local name are identical. However, if you know that we are working in the XHTML workspace, you can test the namespace by its prefix and write `x:*`. The test for local names is done through the predicate `[local-name()=local-name(current())]`. Lastly, to include the name of the element in the report, use the `name` element.

Since you have defined a generic rule to test that elements are not recursively embedded, this rule is a good candidate to be used as an abstract rule:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/xhtml" prefix="x"/>
<pattern name="embedding">
<rule abstract="true" id="embeddedElementsForbidden">
<report test=".//x:*[local-name()=local-name(current())]">We don't want <name/>'s in
<name/>'s!</report>
</rule>
<rule context="x:a">
<extends rule="embeddedElements"/>
<assert test="starts-with(@href, 'http://')">Our guidelines require to use HTTP URIs
in
XHTML links!</assert>
```

```
</rule>
<rule context="x:b">
<extends rule="embeddedElements"/>
</rule>
<rule context="x:i">
<extends rule="embeddedElements"/>
</rule>
</pattern>
</schema>
```

Compared to the first variant, this third one is more modular: if you need to update the report, for instance, to change its text, you'll have to do it only in the abstract rule. Furthermore, this abstract rule becomes a good candidate for a library of generic Schematron rules that you'll include using XInclude, external entities, or the ISO Schematron `include` element.

Compared to the second variant, the third is more verbose but can be considered easier to read since there is one rule per element. This also allows you to add additional specific rules for each element if you need to. This was done for the link (`x:a`) element, with an assertion that checks that the links are using HTTP URIs.

Computation and Iterations

Schematron is very handy for simple computation. There are a lot of checks that can be performed in a document such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice>
<item>
<ref>rf001</ref>
<number>5</number>
<unitPrice>10.1</unitPrice>
<totalPrice>50.5</totalPrice>
</item>
<item>
<ref>rf002</ref>
<number>1</number>
<unitPrice>20.5</unitPrice>
<totalPrice>20.5</totalPrice>
</item>
<total>
<number>2</number>
<totalPrice>71</totalPrice>
</total>
</invoice>
```

In this example, it is easy to check that the total prices are correct for each item line as well as for the grand total and for the number of items that match the actual number of item elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="Compute">
<rule context="item">
<assert test="totalPrice = number * unitPrice"/>
</rule>
```

```
<rule context="invoice">
<assert test="total/totalPrice = sum(item/totalPrice)"/>
<assert test="total/number = count(item)"/>
</rule>
</pattern>
</schema>
```

Such use cases are pretty common, and Schematron helps a lot in enforcing this kind of constraints. However, it is also often true that little differences make such cases impossible to validate with Schematron 1.5. You will see two examples in this section.

The first case is an example of what happens if you remove the total price elements from the item lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice>
<item>
<ref>rf001</ref>
<number>5</number>
<unitPrice>10.1</unitPrice>
</item>
<item>
<ref>rf002</ref>
<number>2</number>
<unitPrice>20.5</unitPrice>
</item>
<total>
<number>2</number>
<totalPrice>71</totalPrice>
</total>
</invoice>
```

This XML document still contains enough information to be able to check that the grand total price is the sum of the total prices of each item, but this requires you to compute these total prices, which is not doable with XPath 1.0. Since Schematron has no support for iteration within rules, you need to use ISO Schematron and a language binding that provides this missing help for iterations. This is done for both EXSLT processors that support the ESXLT Dynamic module (**http://www.exslt.org/dyn/index.html**), such as Amara's Scimitar, which uses 4Suite, and XPath 2.0, which supports `for` loops.

With EXSLT, you would use the `dyn:map()` function. This function takes a node set as its first argument and a string as its second element. It loops over the node set and replaces each node by the result of the evaluation of the string. In this case, the node set can be the `items` elements, and the XPath expression can be the multiplication of the number by the unit price. The result is a node set containing the total price of each item. It can be used as a parameter for the XPath `sum()` function. This is something that is easier to do than to explain; the corresponding schema will help you understand:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="exslt">
```

```
<ns prefix="dyn" uri="http://exslt.org/dynamic"/>
<pattern fpi="checks">
<rule context="/invoice">
<assert test="total/totalPrice = sum(dyn:map(item, 'number * unitPrice'))">The total
price
doesn't match</assert>
<assert test="total/number = count(item)"/>
</rule>
</pattern>
</schema>
```

With XSLT or XPath 2.0, you would use a `for` instruction instead of the `dyn:map()` function:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
<pattern fpi="checks">
<rule context="/invoice">
<assert test="sum(for $item in item return $item/number * $item/unitPrice) =
total/totalPrice"
>The total price doesn't match</assert>
<assert test="total/number = count(item)"/>
</rule>
</pattern>
</schema>
```

Note

As I am writing this (January 2007), there isn't yet any ISO Schematron implementation supporting XPath/XSLT 2.0.

Here's an example of what happens if numbers use a different decimal separator, such as this version of the first invoice, localized in French:

```
<?xml version="1.0" encoding="UTF-8"?>
<facture>
<ligne>
<ref>rf001</ref>
<nombre>5</nombre>
<prixUnitaire>10,1</prixUnitaire>
<prixTotal>50,5</prixTotal>
</ligne>
<ligne>
<ref>rf002</ref>
<nombre>1</nombre>
<prixUnitaire>20,5</prixUnitaire>
<prixTotal>20,5</prixTotal>
</ligne>
<total>
<nombre>2</nombre>
<prixTotal>71</prixTotal>
</total>
</facture>
```

The Schematron tests need to translate the commas into points before doing any computation, and, here again, the `sum()` function cannot be used any longer. The technique to use is exactly the same as in the previous examples and relies on the EXSLT `dyn:map()` function:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="exslt">
<ns prefix="dyn" uri="http://exslt.org/dynamic"/>
<let name"comma" value="','"/>
<let name"dot" value="'.'"/>
<pattern fpi="checks">
<rule context="ligne">
<assert test="translate(prixTotal, $comma, $dot) = translate(nombre, $comma, $dot) *
translate(prixUnitaire, $comma, $dot)"/>
</rule>
<rule context="/facture">
<assert test="translate(total/prixTotal, $comma, $dot) = sum(dyn:map(ligne,
'translate(prixTotal, $comma, $dot)'))">Le prix total ne correspond pas</assert>
<assert test="total/nombre = count(ligne)"/>
</rule>
</pattern>
</schema>
```

Or, it relies on an XPath 2.0 `for` loop:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
<ns prefix="dyn" uri="http://exslt.org/dynamic"/>
<pattern fpi="checks">
<rule context="ligne">
<assert test="translate(prixTotal, $comma, $dot) = translate(nombre, $comma, $dot) *
translate(prixUnitaire, $comma, $dot)"/>
</rule>
<rule context="/facture">
<assert test="translate(total/prixTotal, $comma, $dot) = sum(for $ligne in ligne
return translate($ligne/prixTotal, $comma, $dot))">Le prix total ne correspond
pas</assert>
<assert test="total/nombre = count(ligne)"/>
</rule>
</pattern>
</schema>
```

Note

Not so complex but very verbose XPath expressions are common in Schematron. You will see in a later section how they can often be simplified if you use XML entities.

Closing Schematron Patterns

Everything that isn't explicitly forbidden by an assertion is allowed by a Schematron schema, but that doesn't mean that you can't close a schema—or more precisely, a pattern—if you want or need to.

Closing a schema can be useful if you need to perform structure validation with Schematron. Grammar-based schema languages such as RELAX NG or W3C XML Schema are generally better suited for this task, but there are cases where you might want to use Schematron for that—for instance, when you want to have only a single schema to validate a document, when you want to control comments

or processing instructions, or when you want to control the messages that are raised.

The idea to “close a Schematron pattern” is that you define a rule for each node that is allowed and add a default rule that raises an error. To illustrate how this principle can be put in practice, let’s validate the `author` element:

```
<pattern name="Structure">
<rule context="author">
<assert test="true()"/>
</rule>
<rule context="author/name">
<assert test="true()"/>
</rule>
<rule context="author/born">
<assert test="true()"/>
</rule>
<rule context="author/died">
<assert test="true()"/>
</rule>
<rule context="*">
<assert test="false()"> This element (<name/>) is forbidden. </assert>
</rule>
</pattern>
```

The first four rules are in the example to escape from the default rule that raises an error. These rules define the XPath path of allowed elements. Unfortunately, the Schematron specification doesn’t allow empty rules. To work around this, you need to include a fake assert or report with a test that always returns true or false when you don’t have anything to test in a rule.

With Schematron 1.5, it is possible to use the same technique to define where text nodes are expected:

```
<pattern name="Structure+text">
<rule context="author">
<assert test="true()"/>
</rule>
<rule context="author/name">
<assert test="true()"/>
</rule>
<rule context="name/text()">
<assert test="true()"/>
</rule>
<rule context="author/born">
<assert test="true()"/>
</rule>
<rule context="born/text()">
<assert test="true()"/>
</rule>
<rule context="author/died">
<assert test="true()"/>
</rule>
<rule context="died/text()">
<assert test="true()"/>
</rule>
<rule context="*">
```

```
<assert test="false()"> This element (<name/>) is forbidden. </assert>
</rule>
<rule context="text()[normalize-space()]">
<assert test="false()"> This text is forbidden. </assert>
</rule>
</pattern>
```

Note

In this pattern, we were careful to specify that we want to exclude only text nodes—which are not made of whitespaces—through the `[normalize-space()]` condition. If we don't take this precaution, the schema would forbid whitespaces outside the `name`, `born`, and `died` elements!

Controlling whitespaces is yet another thing that Schematron can do that grammar-based schema languages can't.

That being said, imposing conditions on whitespaces is something that is very rarely needed and would upset most of your users!

Unfortunately, ISO Schematron doesn't support setting rule contexts to text nodes. If you want to make something portable in ISO Schematron, you need to test that there isn't whitespace text in each node that doesn't allow it. In the following example, there is only one such element (the `author` element), and the schema is less verbose, but its style is different:

```
<pattern name="Structure+text">
<rule context="author">
<report test="text()[normalize-space()]"> This text is forbidden. </report>
</rule>
<rule context="author/name">
<assert test="true()"/>
</rule>
<rule context="author/born">
<assert test="true()"/>
</rule>
<rule context="author/died">
<assert test="true()"/>
</rule>
<rule context="*">
<assert test="false()"> This element (<name/>) is forbidden. </assert>
</rule>
</pattern>
```

This technique can be extended to impose a location to comments and processing instructions if needed. (I am not suggesting that this is a good practice but just showing how this can be done, if required.) For instance, to impose that comments and processing instructions can only be found under the document root, you could write:

```
<pattern name="Structure+text+comment+PI">
.
.
```

```
.
<rule context="/processing-instruction()">
<assert test="true()"/>
</rule>
<rule context="/comment()">
<assert test="true()"/>
</rule>
<rule context="processing-instruction()">
<assert test="false()"> This processing instruction (<name/>) is forbidden here.
</assert>
</rule>
<rule context="comment()">
<assert test="false()"> This comment is forbidden here. </assert>
</rule>
</pattern>
```

With ISO Schematron, the same technique can be used for attributes. This doesn't work with Schematron 1.5, which doesn't support setting rule contexts to attributes.

Warning

Avoid setting context rules to `node()`. This produces different results between Schematron 1.5 (for which this matches the root, element, text, comment, and processing instruction nodes) and ISO Schematron (for which this does not match text nodes). This could even create interoperability issues between ISO Schematron implementations because the specification doesn’t state whether the implementations should raise errors or warnings or just silently ignore text nodes that would be matched by these rules.

Mixed Content

Content models where elements and text nodes can be mixed are called *mixed contents*. These are used frequently in document-oriented applications. A typical example of mixed content is the content model of the XHTML `p` element:

```
<p class="note"><b>Mixed content models</b> are so often used in document oriented
applications and so rarely used in data oriented applications that there presence has
become <em>one of the best indications</em> to check whether an <acronym
title="Extensible Markup Language">XML</acronym> application is document or data
oriented.</p>
```

Neither RELAX NG nor W3C XML Schema allows you to define any constraint on the text nodes in mixed content models. With both, you can define which subelements are allowed, but you cannot constrain the text nodes between these elements.

There are still some cases where you may need to define constraints on the text nodes of mixed content. One such example is when you want to restrict the characters that can be used in a text. Another example is a vocabulary in which a mixed content model is used for data. For instance, you could define that you want

to accept both `<price><currency>USD</currency>200</price>` and `<price>200<currency>Euros</currency></price>`.

This vocabulary would be considered a bad practice by most of the XML community, which would instead advise either to use an attribute for the currency (`<price currency="USD>200</price>`) or to wrap the value in an element (`<price><currency>USD</currency><value>200</value></price>)`.

Compared to using attributes, your mixed content solution has the advantage of defining whether the currency should be displayed before or after the value. Compared to wrapping the value in an element, you could argue that showing the currency in this way is more concise.

These two examples are interesting because they show the two different kinds of difficulties that you may encounter when dealing with text nodes with Schematron.

When you want to test character sets in XPath, the main points are to express the fact that each character belongs to a specific set of characters, and the way to express this test is different in XPath 1.0 and XPath 2.0.

Warning

The encoding defined in the XML declaration of a document shouldn't be taken as a guarantee that the characters in the document belong to a specific character set. This encoding is nothing more than the physical encoding that is used to encode the XML document, and the XML recommendation makes it clear that any Unicode character (except a few control characters) may be included as a numeric character entity reference in an XML document independently of its encoding.

In other words, a XML document encoded as US-ASCII—which is the most limited encoding supported by common XML parsers—can still include any Chinese character that is defined in Unicode!

In XPath 1.0, a common hack to test that all the characters of a string belong to a set of characters is to remove the allowed characters from the string using the `translate()` function and to see if there is anything remaining. To test that a string contains only the characters "aeiouy," you would write: `translate(myString, 'aeiouy') = ''`.

Note

If your boss complains that this is a poor hack, you can answer that a generalization of this method—removing everything that is allowed and looking at what is left—is known as "Brzozowski derivatives" and is behind some of the most efficient algorithms for implementing regular expressions and grammar-based XML schema languages.

The remaining problem is to define the set of characters that needs to be allowed. XML is based on Unicode, which means that Unicode is the best normative reference for characters used in XML documents. The Unicode specifications are published under a number of formats that include machine processable "Unicode character tables." You can use these tables directly or through your favorite programming language (which probably has some level of Unicode support built in). This will help you to get the list of characters that you will allow in a form that you can copy and paste in an argument to the translate function. For instance, the set of Unicode characters for the Basic Latin bloc would be the following string:

```
!&quot;#$%&amp;'()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
```

This Unicode block is more complex than others because you have a single and a double quote at the same time, but you can't include both at the same time in an XPath literal. To work around this, you can remove one of them with a first call to the translate function and remove all the other characters through a second pass. The resulting pattern, which checks that a document uses only Basic Latin characters, would be as follows:

```
<pattern name="Character set">
<rule context="text()">
<assert
test="translate(translate(., &quot;'&quot;, ''), ' &#xa;&#xd;!&quot;#$%&amp;()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
', '') = ''"
>Non Basic Latin characters found!</assert>
</rule>
</pattern>
```

This works fine, but in real life, you might want applications to define different sets of allowed characters, depending on conditions such as `xml:lang` attribute values. To say that, for instance, you allow characters from the Latin-1 Supplement to text written in French, you need to express the fact that a text has an `xml:lang` attribute starting with `fr`. (By matching attributes that start with `fr` instead of only than those equal to `fr`, we include `fr` (French) and all its variants, such as `fr-FR`, `fr-BE`, etc.)

The first step is to deal with the fact that the `xml:lang` attribute is inherited. The condition will thus apply to the first ancestor element with a `xml:lang` attribute. The XPath expression to express that is as follows:

```
text()[ancestor::*[xml:lang][1]]
```

Now that we know which ancestor has an `xml:lang` attribute, you need to test whether this attribute starts with `fr`. Use the `starts-with` function. Be aware that `xml:lang` attributes can include leading whitespaces and can normalize the spaces before calling `starts-with`. The resulting pattern is:

```
<pattern name="Character set">
<rule
context="text()[starts-with(normalize-
space(ancestor::*[@xml:lang][1]/@xml:lang),'fr')]">
<assert
test="translate(translate(., &quot;'&quot;, ''), ' &#xa;&#xd;!&quot;#$%&amp;()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
¡¢£¤¥¦§¨©ª«¬-
®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþ',
'') = ''"
>Only Basic Latin or Latin-1 Supplement characters are allowed!</assert>
</rule>
<rule context="text()">
<assert
test="translate(translate(., &quot;'&quot;, ''), ' &#xa;&#xd;!&quot;#$%&amp;()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
', '') = ''"
>Non Basic Latin characters found!</assert>
</rule>
</pattern>
```

This pattern relies on setting rule contexts on node texts and doesn’t work with ISO Schematron. A pattern that works with both ISO Schematron and Schematron 1.5 must set the rules on the parent elements and report an error if there exists a node test for which the condition isn’t met:

```
<pattern name="Character set">
<rule
context="*[starts-with(normalize-space(ancestor-or-
self::*[@xml:lang][1]/@xml:lang),'fr')]">
<report
test="text()[translate(translate(., &quot;'&quot;, ''), '
&#xa;&#xd;!&quot;#$%&amp;()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
¡¢£¤¥¦§¨©ª«¬-
®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþ',
'') != '']"
>Only Basic Latin or Latin-1 Supplement characters are allowed!</report>
</rule>
<rule context="*">
<report
test="text()[translate(translate(., &quot;'&quot;, ''), '
&#xa;&#xd;!&quot;#$%&amp;()*+,-
./0123456789:;&lt;=&gt;?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
', '') != '']"
>Non Basic Latin characters found!</report>
</rule>
</pattern>
```

Warning

There are a couple of traps that you should be aware of around these tests.

The first is that we could be tempted to test against the current node rather than against its text nodes, and write: `test="translate(translate(., "'", ''), ...) != '']"`. This would be a valid XPath expression, but the current node (.) would be converted to text following the XPath casting rules, which are used in this case to concatenate the text nodes of the current node and all its descendants. That would not only be terribly inefficient, since the same text nodes would be tested one time per ancestor, but also inaccurate, since that would test a subnode with a different `xml:lang` attribute with the rules defined for the `xml:lang` for the current node.

The second trap is that in avoiding that first problem, we might be tempted to write: `test="translate(translate(text(), "'", ''), ...) != '']"`. This would also be a valid XPath expression, but the rule to convert a set of nodes into text for the first parameter of the `translate` function is to keep only the first node. In a situation in which the current element has several children text nodes(which can happen for mixed content models or even when there is a comment or processing instruction in a simple content element), the test would be done only on the first text node.

To avoid these two traps, you need to use this rather complex form for your reports as well as write an XPath expression that reads as “Report an error if there is a text node that matches the condition...”.

Now that you have seen how ISO Schematron makes your schema more complex, it is time to see how it can simplify it. This happens when you are using a Schematron implementation that supports the EXSLT, XPath 2.0, or XSLT 2.0 binding.

Note

The rest of this section assumes that you are familiar with regular expressions. If this is not the case, you can have a look at one of the many tutorials available online (including the one that I have written for my RELAX NG book, available at http://books.xmlschemata.org/relaxng/relax-CHP-9.html) or read *Mastering Regular Expressions* by Jeffrey E.F. Friedl (O’Reilly), which is the reference in this domain.

If the EXSLT implementation that you are using supports it, you can use the “regexp” module to test character ranges. The selection of rules remains the same, but the regular expressions provide a much more concise alternative to using `translate`. The complete schema would become:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="exslt">
<ns prefix="regexp" uri="http://exslt.org/regular-expressions"/>
<pattern fpi="Character set">
<!--
These rules work only with an ISO Schematron
implementation supporting the EXSLT regexp module
such as 4Suite "Scimitar".
-->
<rule
context="*[starts-with(normalize-space(ancestor-or-
self::*[@xml:lang][1]/@xml:lang),'fr')]">
<report test="text()[regexp:test(., '[^&#xa;-&#xff;]')]"><name/>: Only Basic
Latin or Latin-1 Supplement characters are allowed!</report>
</rule>
<rule context="*">
<report test="text()[regexp:test(., '[^&#xa;-&#x7f;]')]"><name/>: Non Basic
Latin character(s) found!</report>
</rule>
</pattern>
</schema>
```

XPath 2.0 and XSLT 2.0 give you access to the `matches()` regular expression function, which can be used with the same regular expression as the EXSLT version that you just saw. However, the regular expressions used by the `matches()` function are the same as those of W3C XML Schema. One of the good things about regular expressions is that they give direct access to Unicode blocks. For instance, the regular expression that is used to test whether a text belongs to the Basic Latin block is `^\p{IsBasicLatin}*$`. If you prefer testing to check whether a text doesn't contain any non-Basic Latin characters (which is equivalent), you can write `[^\p{IsBasicLatin}]`. Using this feature is more verbose than using ranges, but it can be considered more readable since the names of the Unicode blocks appear clear. (In the next section, you will see that you could use XML entities to obtain a similar effect with EXSLT.) A schema for an ISO Schematron supporting XPath 2.0 or XSLT 2.0 would be:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
<pattern fpi="Character set">
<!--
These rules work only with an ISO Schematron
implementation supporting the XSLT 2.0 query binding.
-->
<rule
context="*[starts-with(normalize-space(ancestor-or-
self::*[@xml:lang][1]/@xml:lang),'fr')]">
<report test="text()[matches(., '[^\p{IsBasicLatin}\p{IsLatin-
1Supplement}]]')]"><name/>: Only Basic
Latin or Latin-1 Supplement characters are allowed!</report>
</rule>
<rule context="*">
<report test="text()[matches(., '[^\p{IsBasicLatin}]')]"><name/>: Non Basic
Latin character(s) found!</report>
</rule>
</pattern>
</schema>
```

Warning

Here is what John Cowan, who has taught courses on Unicode and enjoys obscure alphabets, wrote about this topic when he reviewed my RELAX NG book:

“It’s important to note that Unicode blocks are a very crude mechanism for discrimination: not everything needed to write Greek is in the Greek block, and there are no less than five Latin blocks, one of which (Basic Latin; i.e., ASCII) contains many script-independent symbols. The blocks were originally created solely for internal Unicode organizational purposes, and have spread to the outside world somewhat randomly.”

The five Latin blocks mentioned by John are BasicLatin, Latin1Supplement, LatinExtended-A, LatinExtendedAdditional, and LatinExtended-B.

Entities

Good old XML entities are very handy for shortening complex XPath expressions. This is most useful with Schematron 1.5, which has no support for variables. Onec you start using entities, you’ll find them useful, even with ISO Schematron.

Warning

XML entities are inherited from the SGML days and are more than mature. Their support by XML parsers is fairly good, and there are very few remaining interoperability issues.

You should note, though, that XML entities are forbidden by SOAP and that XML documents defining (and using) XML entities cannot be sent in SOAP messages.

One of the reasons to forbid XML entities is that they can potentially cause security issues or at least cause out-of-memory errors. The following code shows a simple example known as “the billion laughs attack”:

```
<?xml version="1.0"?>
<!DOCTYPE billion [
<!ELEMENT billion (#PCDATA)>
<!ENTITY laugh0 "ha">
<!ENTITY laugh1 "&laugh0;&laugh0;">
<!ENTITY laugh2 "&laugh1;&laugh1;">
<!ENTITY laugh3 "&laugh2;&laugh2;">
<!ENTITY laugh4 "&laugh3;&laugh3;">
<!ENTITY laugh5 "&laugh4;&laugh4;">
<!ENTITY laugh6 "&laugh5;&laugh5;">
<!ENTITY laugh7 "&laugh6;&laugh6;">
<!ENTITY laugh8 "&laugh7;&laugh7;">
<!ENTITY laugh9 "&laugh8;&laugh8;">
<!ENTITY laugh10 "&laugh9;&laugh9;">
<!ENTITY laugh11 "&laugh10;&laugh10;">
<!ENTITY laugh12 "&laugh11;&laugh11;">
<!ENTITY laugh13 "&laugh12;&laugh12;">
<!ENTITY laugh14 "&laugh13;&laugh13;">
<!ENTITY laugh15 "&laugh14;&laugh14;">
<!ENTITY laugh16 "&laugh15;&laugh15;">
<!ENTITY laugh17 "&laugh16;&laugh16;">
<!ENTITY laugh18 "&laugh17;&laugh17;">
<!ENTITY laugh19 "&laugh18;&laugh18;">
<!ENTITY laugh20 "&laugh19;&laugh19;">
<!ENTITY laugh21 "&laugh20;&laugh20;">
<!ENTITY laugh22 "&laugh21;&laugh21;">
<!ENTITY laugh23 "&laugh22;&laugh22;">
<!ENTITY laugh24 "&laugh23;&laugh23;">
<!ENTITY laugh25 "&laugh24;&laugh24;">
<!ENTITY laugh26 "&laugh25;&laugh25;">
<!ENTITY laugh27 "&laugh26;&laugh26;">
<!ENTITY laugh28 "&laugh27;&laugh27;">
<!ENTITY laugh29 "&laugh28;&laugh28;">
<!ENTITY laugh30 "&laugh29;&laugh29;">
]>
<billion>&laugh30;</billion>
```

Through the repeated expansion of the `laugh` entities, this code will rapidly exhaust the memory of the virtual machine or computer in which you parse it!

Let's revisit the date example used to introduce ISO Schematron variables. With ISO Schematron, you define variables as follows:

```
<let name="year" value="substring(., 1, 4)"/>
```

These are reused as XPath variables:

```
<report test="$day =0 or $day > 31">Wrong date (days should be between 1 and
31)</report>
```

The principle is the same with entities except that entities are defined in a DTD (Document Type Definition) that can either be internal or external to the document (variables can be defined pretty much everywhere in a document).

Their scope is always global to the whole document (the scope of variables is limited to the schema, phase, pattern, or rule in which they are defined).

You can reference entities anywhere in the schema (variables can be referenced only within XPath expressions).

The adapted schema is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema [
<!ENTITY year "substring(., 1, 4)">
<!ENTITY month "substring(., 6, 2)">
<!ENTITY day "substring(., 9, 2)">
]>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="dates">
<rule context="born|died">
<report test="string-length(.) != 10">Wrong date size (dates should be 10 character
long).</report>
<report test="translate(&year;, '0123456789', '') != ''">Wrong date format (dates
should
start with four digits for the year)</report>
<report test="substring(., 5, 1) != '-' ">Wrong date format</report>
<report test="translate(&month;, '0123456789', '') != ''">Wrong date format (dates
should
have two digits for the month)</report>
<report test="substring(., 8, 1) != '-' ">Wrong date format</report>
<report test="translate(&day;, '0123456789', '') != ''">Wrong date (dates should end
with
two characters for the day)</report>
<report test="&day; =0 or &day; > 31">Wrong date (days should be between 1 and
31)</report>
<report
test="(&month; = '04' or &month; = '06' or &month; = '09' or &month; = '11') and &day;
> 30"
>Wrong date (days should be between 1 and 30 in April, June, September and
November)</report>
<report test="&month; = '02' and &day; > 29">Wrong date (days should be between 1 and
29
in February)</report>
</rule>
</pattern>

</schema>
```

Note

The readability of this schema is close to the readability of the equivalent ISO Schematron schema that uses variables, but this schema is slower to execute on nonoptimized implementations that recompute the value of the substrings at each occurrence of the expanded entities.

The previous example shows how entities can be used for text replacement in XPath expressions. That's a common use case, but entities are not restricted to pseudovariables for Schematron 1.5! Being a general XML mechanism, they can be used to replace text anywhere in a Schematron schema (for instance, in the text of `report`, `assert`, and `diagnostic` elements), and they can also replace elements or sets of elements.

Using entities to replace `assert` or `report` elements isn't very useful since that would compete with abstract rules, which are more readable. But, there are other cases where this can be handy. We've seen, for instance, that the `name` element is forbidden in a `diagnostic` element and that you need to use `<value-of select="name()"/>` instead. To illustrate how entities can be used to replace elements, let's see how we could replace this element with a `&diagName;` entity.

The entity definition is created as it was in the previous example; however, now the definition contains a full element:

```
<!ENTITY diagName '<value-of select="name()"/>'>
```

The < character in the entity value shouldn't be escaped, and the only precautions you have to take is to make sure that the entity value is well-formed and that there is no conflict between the quotes used to delimit the entity value (here, a single quote) and the quotes used within your entity value (here, double quotes to delimit the value of the `select` attribute).

The entity can then be referred as usual through `diagName`. The complete schema would look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema [
<!ENTITY year "substring(., 1, 4)">
<!ENTITY month "substring(., 6, 2)">
<!ENTITY day "substring(., 9, 2)">
<!ENTITY diagName '<value-of select="name()"/>'>
<!ENTITY diagValue '<value-of select="."/>'>
]>
<schema xmlns="http://www.ascc.net/xml/schematron">
<pattern name="dates">
<rule context="born|died">
.../...
<report test="translate(&year;, '0123456789', '') != ''"
diagnostics="dateYear.en dateYear.fr">Wrong date format (dates should start with
four digits for the year)</report>
.../...
```

```
</rule>
</pattern>
<diagnostics>
<diagnostic id="dateYear.en" xml:lang="en">ISO dates must start by four digits for the
year.
This is not the case for the value "&diagValue;" found in element
"&diagName;".</diagnostic>
<diagnostic id="dateYear.fr" xml:lang="fr">ISO dates must start by four digits for the
year.
This is not the case for the value "&diagValue;" found in element
"&diagName;".</diagnostic>
</diagnostics>

</schema>
```

Processing Instructions

Schema languages such as RELAX NG and W3C XML Schema just ignore processing instructions and drop them before validating a document. This isn't the case with Schematron, which makes Schematron the language of choice if you need to validate processing instructions.

Let's say that you want to check that there is a single `xml-stylesheet` processing instruction before the document element with a `type` pseudo attribute equal to `text/xsl`, and that this processing instruction points to a XSLT transformation.

> **Note**
>
> Although the syntax of a processing instruction such as `<?xml-stylesheet href="dummy.xsl" type="text/xsl"?>` looks like the syntax of an element with a set of attributes, these attributes are not real XML attributes, which is why we call them *pseudo-attributes*.
>
> Following this principle, the XPath data model considers that this processing instruction is a node with a type "processing instruction," a name equal to `xml-stylesheet` and a value equal to the string `href="dummy.xsl" type="text/xsl"`.
>
> The consequence is that you will have to parse this value in XPath to extract these pseudo-parameters. This is a good exercise that is worth undertaking even if you're not interested in validating processing instructions.

With Schematron 1.5

The exercise is more complex than it looks. Start by trying to see how far you can go with Schematron 1.5.

To validate that there is a processing instruction under the document root, define a rule in which context is the document root (`/`) and test that you have a processing instruction:

```
<rule context="/">
<assert test="processing-instruction()">There should be a processing
instruction</assert>
</rule>
```

Easy enough, but how do you check that is coming before the root element? Use the `preceding-sibling` XPath axis:

```
<rule context="/">
<assert test="processing-instruction()[not(preceding-sibling::*)]">There should be a
processing instruction before the root
element.</assert>
</rule>
```

Now you need to specify that the name of the processing instruction is `xml-stylesheet`, which is done by appending a new predicate such as `[name()='xml-stylesheet']` and checking that the content of the stylesheet contains the string `type="text/xsl"`, which can be tested by adding the predicate `[contains(., ' type="text/xsl"')]`.

The assertion is then:

```
<assert test="processing-instruction()[not(preceding-sibling::*)][name()='xml-
stylesheet'][contains(., 'type=&quot;text/xsl&quot;')]">There should be a 'xml-
stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
```

The XPath expression is beginning to get difficult to read, so it is time to start using entities. With entities, you can nicely express the progression in the different steps that you've just gone through, by the followoing:

```
<!DOCTYPE schema[
<!ENTITY PI "processing-instruction()">
<!ENTITY LeadingPI "&PI;[not(preceding-sibling::*)]">
<!ENTITY LeadingXmlStylesheetPI "&LeadingPI;[name()='xml-stylesheet']">
<!ENTITY LeadingXmlStylesheetXslPI "&LeadingXmlStylesheetPI;[contains(.,
'type=&quot;text/xsl&quot;')]">
```

In these definitions, each step just adds a new predicate to the previous one, as we did to explain the process. And since each step is defined as an entity, it is possible to debug them step by step. After the last step, we are expressing that there is at least one processing instruction that matches our first steps of requirement. To say that there is exactly one, we need to count the number of these expressions. The rule becomes:

```
<rule context="/">
<assert test="count(&LeadingXmlStylesheetXslPI;) = 1">There should be one and only one
'xml-stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
</rule>
```

We have expressed the first requirement, which is to check that there is a single `xml-stylesheet` processing instruction before the document element with a

type pseudo-attribute equal to text/xsl. The second requirement is to test whether this processing instruction points to a XSLT stylesheet. To implement it, we will write a second rule in the context of this stylesheet: `<rule context="/&LeadingXmlStylesheetXslPI;">`. I hope you feel lucky that you were able to define this entity, because if not, we would have had to copy the same verbose XPath expression!

Your next task is to extract the value of the href pseudo-attribute. With XPath 1.0, this is typically done by cutting the value of the processing instruction after the string type=" and then cutting the result before the character ". Again, you can use entities to do so:

```
<!ENTITY tailHref "substring-after(., 'href=&quot;')">
<!ENTITY href "substring-before(&tailHref;, '&quot;')">
```

Now that we have the value of the href pseudo attribute, we use the document() function to retrieve the document referenced by this URI. Before that, you need to test that the value you have retrieved isn't the empty string—per the XSLT 1.0 recommendation, the document("") function call returns the XSLT transformation itself, which, at least when the implementation is based on XSLT, will by definition always be a XSLT transformation.

Finally, how do you check whether a document is a XSLT transformation? In their more complete flavors, XSLT transformations can have two different root elements. They can also have a simplified form in which XSLT instructions are embedded in documents. In this case, the root element must have an xsl:version attribute.

Your resulting rule thus looks like:

```
<rule context="/&LeadingXmlStylesheetXslPI;">
<assert
test="&href; != '' and document(&href;, .)/*[self::xsl:stylesheet or
self::xsl:transformation or @xsl:version]"
>The href attribute should point to a XSLT transformation and that doesn't appear to
be the case!</assert>
</rule>
```

At this point, the complete schema is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema[
<!ENTITY PI "processing-instruction()">
<!ENTITY LeadingPI "&PI;[not(preceding-sibling::*)]">
<!ENTITY LeadingXmlStylesheetPI "&LeadingPI;[name()='xml-stylesheet']">
<!ENTITY LeadingXmlStylesheetXslPI "&LeadingXmlStylesheetPI;[contains(.,
'type=&quot;text/xsl&quot;')]">
<!ENTITY tailHref "substring-after(., 'href=&quot;')">
<!ENTITY href "substring-before(&tailHref;, '&quot;')">
]>
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/XSL/Transform" prefix="xsl"/>
<pattern name="PIs">
```

```
<rule context="/">
<assert test="count(&LeadingXmlStylesheetXslPI;) = 1">There should be one and only
one 'xml-stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
</rule>
<rule context="/&LeadingXmlStylesheetXslPI;">
<assert
test="&href; != '' and document(&href;, .)/*[self::xsl:stylesheet or
self::xsl:transformation or @xsl:version]"
>The href attribute should point to a XSLT transformation and that doesn't appear to
be the case!</assert>
</rule>
</pattern>
</schema>
```

There are a couple of criticisms that can be made against this schema:

- The parsing of pseudo-attributes is very basic and matches strings such as `<?xml-stylesheet myhref="dummy.xsl"mytype="text/xsl"?>`, which are not valid stylesheet attachments.
- It only accepts pseudo-attributes with values that are delimited by double quotes.

You can improve the parsing to partially fix the first bullet point by checking that the pseudo-attributes are whitespace-separated. The common trick is to normalize the spaces of the value of the processing instruction and to add a leading and a trailing space. This is enough to be safe when we test that the pseudo-attributes need to be followed and preceded by a space character.

A quick hack to improve the second bullet point is to use the `translate` function to replace single quotes with double quotes. These two features can be implemented by a single entity definition that you can call `safeContent`. The final schema is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema[
<!ENTITY safeContent "concat(' ', normalize-space(translate(., &quot;'&quot;,
'&quot;')), ' ')">
<!ENTITY PI "processing-instruction()">
<!ENTITY LeadingPI "&PI;[not(preceding-sibling::*)]">
<!ENTITY LeadingXmlStylesheetPI "&LeadingPI;[name()='xml-stylesheet']">
<!ENTITY LeadingXmlStylesheetXslPI "&LeadingXmlStylesheetPI;[contains(&safeContent;, '
type=&quot;text/xsl&quot; ')]">
<!ENTITY tailHref "substring-after(translate(&safeContent;, &quot;'&quot;, '&quot;'),
' href=&quot;')">
<!ENTITY href "substring-before(&tailHref;, '&quot; ')">
]>
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/XSL/Transform" prefix="xsl"/>
<pattern name="PIs">
<rule context="/">
<assert test="count(&LeadingXmlStylesheetXslPI;) = 1">There should be one and only
one 'xml-stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
```

```
</rule>
<rule context="/&LeadingXmlStylesheetXslPI;">
<assert
test="&href; != '' and document(&href;, .)/*[self::xsl:stylesheet or
self::xsl:transformation or @xsl:version]"
>The href attribute should point to a XSLT transformation and that doesn't appear to
be the case!</assert>
</rule>
</pattern>
</schema>
```

If your colleagues say you are old-fashioned because you're using entities, you can show them the same schema without entities, which should be enough to convince them:

```
<schema xmlns="http://www.ascc.net/xml/schematron">
<ns uri="http://www.w3.org/1999/XSL/Transform" prefix="xsl"/>
<pattern name="PIs">
<rule context="/">
<assert test="count(processing-instruction()[not(preceding-sibling::*)][name()='xml-
stylesheet'][contains(concat(' ', normalize-space(translate(., &quot;'&quot;,
'&quot;')), ' '), ' type=&quot;text/xsl&quot; ')]) = 1">There should be one and only
one 'xml-stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
</rule>
<rule context="/processing-instruction()[not(preceding-sibling::*)][name()='xml-
stylesheet'][contains(concat(' ', normalize-space(translate(., &quot;'&quot;,
'&quot;')), ' '), ' type=&quot;text/xsl&quot; ')]">
<assert test="substring-before(substring-after(translate(concat(' ', normalize-
space(translate(., &quot;'&quot;, '&quot;')), ' '), &quot;'&quot;, '&quot;'), '
href=&quot;'), '&quot; ') != '' and document(substring-before(substring-
after(translate(concat(' ', normalize-space(translate(., &quot;'&quot;, '&quot;')), '
'), &quot;'&quot;, '&quot;'), ' href=&quot;'), '&quot; '), .)/*[self::xsl:stylesheet
or self::xsl:transformation or @xsl:version]">The href attribute should point to a
XSLT transformation and that doesn't appear to
be the case!</assert>
</rule>
</pattern>
</schema>
```

This schema isn't perfect and will still accept processing instructions that are not valid stylesheet attachments. For instance, it will accept a processing instruction with two `href` or `type` pseudo-attributes, one with pseudo-attributes that are forbidden, or one with pseudo-attributes whose value is delimited by one simple quote and one double quote.

You can continue to improve the schema, but we seem to be reaching the limits of what is reasonable with XPath 1.0.

With ISO Schematron

The ISO Schematron variables will not help that much compared to our entities because we need to include a pretty complex condition in the `rule context` attribute that variables cannot be used in `context` attributes (have a look at its expanded version in the preceding example if you don't trust me).

Note

The content of the `context` attribute is the same subset of XPath that is allowed in an XSLT `template match` attribute.

One of the restrictions of XPath expressions in these attributes is that you can't use variables.

Regular expressions are very convenient to further improve this schema, and they are available both through the EXSLT regexp" module (**http://www.exslt.org/regexp/**) and through XPath 2.0. The main difference between these two implementations is that the EXSLT regexp module can generate node sets from a regular expressions group while XPath 2.0 cannot.

Note

Generating node sets from regular expression matching groups is supported by XSLT 2.0 instructions, which are not available in ISO Schematron.

There is no reason to break something that is working, and using regular expressions would not dramatically improve the readability of what is already tested by this schema. Just use them to improve the test coverage of the rule applied to the leading `xml-stylesheet` PI.

The first step is to write the pattern that describes a processing instruction pseudo-attribute. Since you are only updating the previous schema, you can continue to work on the `safeContent` entity, which adds a leading space to the PI value and translates single quotes into double quotes. This makes much easier, and this pattern can be something such as:

```
( [a-z]+=&quot;[^&quot;]*&quot;).
```

With this pattern, we can do two different kinds of things: we can check that the value is only made of such patterns, and we can split the value into pseudo-attributes that we can check individually. Following these principles, the full schema can be:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema[
<!ENTITY safeContent "concat(' ', normalize-space(translate(., &quot;'&quot;,
'&quot;')), ' ')">
<!ENTITY PI "processing-instruction()">
<!ENTITY LeadingPI "&PI;[not(preceding-sibling::*)]">
<!ENTITY LeadingXmlStylesheetPI "&LeadingPI;[name()='xml-stylesheet']">
<!ENTITY LeadingXmlStylesheetXslPI "&LeadingXmlStylesheetPI;[contains(&safeContent;, '
type=&quot;text/xsl&quot; ')]">
<!ENTITY atom "( [a-z]+=&quot;[^&quot;]*&quot;)">
]>
<?oxygen RNGSchema="file:/home/vdv/cvs-private/presentations/en/Schematron/iso-
schematron.rnc" type="compact"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="exslt">
<ns prefix="regexp" uri="http://exslt.org/regular-expressions"/>
```

```
<pattern fpi="PIs">
<rule context="/">
<assert test="&PI;">There should be a processing instruction</assert>
<assert test="&LeadingPI;">There should be a processing instruction before the root
element.</assert>
<assert test="&LeadingXmlStylesheetPI;">There should be a 'xml-stylesheet' processing
instruction before the root element.</assert>
<assert test="&LeadingXmlStylesheetXslPI;">There should be a 'xml-stylesheet'
processing
instruction with type="text/xsl" before the root element.</assert>
<assert test="count(&LeadingXmlStylesheetXslPI;) = 1">There should be one and only one
'xml-stylesheet' processing instruction with type="text/xsl" before the root
element.</assert>
</rule>
<rule context="/&LeadingXmlStylesheetXslPI;">
<!--
This rule works only with an ISO Schematron
implementation supporting the EXSLT regexp module
such as 4Suite "Scimitar".
-->
<let name="normalized" value="&safeContent;)"/>
<let name="pseudoAttributes" value="regexp:match($normalized, '&atom;', 'g')"/>
<let name="href" value="$pseudoAttributes[starts-with(., ' type=')]"/>
<assert test="regexp:test($normalized, '^&atom;+$')"> Syntax error <value-of
select="."/>. </assert>
<assert test="$href"> href pseudo-attribute missing <value-of select="."/>
</assert>
<report
test="$pseudoAttributes[not(starts-with(., ' type=')) and not(starts-with(., '
href='))]">
Non expected pseudo-attribute <value-of select="."/>
</report>
<assert
test="document(substring-before(substring-after($href, ' href=&quot;'), '&quot;'),
.)/*[self::xsl:stylesheet or self::xsl:transformation or @xsl:version]"
>The href attribute should point to a XSLT transformation and that doesn't appear to
be the
case!</assert>
</rule>
</pattern>
</schema>
```

In the new version of the rule, start by storing the normalized value into a variable to avoid computing it several times. Then, you split this value into pseudo-attributes using the `regexp:match()` function and store the result (which is a node set) into a variable. The last variable is for storing the type pseudo-attribute that you'll use a couple of times. With these three variables ready, you can start testing. The first `assert` uses the `regexp:test()` function to check that the value is only composed of iterations of the pattern. This is needed because `regexp:match()` silently stops at the first character that doesn't match the pattern. The second `assert` checks that there is an `href` pseudo-attribute. The `report` checks that we have no other pseudo-attribute other than `type` and `href`. Finally, the last `assert` tests that the document referenced by the `href` pseudo-attribute is an XSLT transformation.

Note

To remain as concise as possible, this schema has been simplified. In order to be compatible with the W3C Recommendation, which describes the `xml-stylesheet` PI (**http://www.w3.org/TR/xml-stylesheet/**), a more complete one should allow the optional pseudo-attributes `media`, `charset`, and `alternate`.