Describing Document Types: The Schema Languages of XML Part 1

John Cowan

Copyright

- Copyright © 2003-05 John Cowan
- Licensed under the GNU General Public License
- ABSOLUTELY NO WARRANTIES; USE AT YOUR OWN RISK
- Portions written by James Clark; used by permission
- Black and white for readability
- The Gentium font available at *http://www.sil.org/~gaultney/gentium*

Abstract

This tutorial will teach you the basics of several XML schema languages: DTDs, RELAX NG, Schematron, and W3C XML Schema. You will end up understanding the principles of each and their advantages and disadvantages in various applications.

Prerequisites: An understanding of basic XML concepts. Knowledge of any XML or SGML schema languages is helpful but certainly not a requirement.

Part 1 Abstract

In this part of the tutorial you will learn how to use the RELAX NG schema language, one of the principal schema languages for XML. RELAX NG allows easy and intuitive descriptions of just what is and what is not allowed in an XML document. It is simple enough to learn in a few hours, and rich and flexible enough to support the design and validation of every kind of document from the very simple to the very complex.

We will also review DTDs, and talk briefly about Schematron (a rules-based schema language) and NVDL (a meta-schema language for compound documents).

Roadmap

- Review of DTDs (17)
- RELAX NG goals (21)
- Invoice example(10)
- Basic Patterns (13)
- More Patterns (13)
- Datatypes (11)
- Tools (14)
- NVDL (4)
- Schematron (12)

REVIEW OF DTDS

DTDs and documents

- DTDs can appear within a document, outside a document, or both
- The `DOCTYPE` declaration specifies a DTD
- Internal subset:
  `<!DOCTYPE root [[ ... ]]>`
- External subset:
  `<!DOCTYPE root SYSTEM "http://...">`
- External before internal

Functions of the DTD

- Document validation
- Default values for missing attributes
- Entity declaration and replacement
- Document documentation

`ELEMENT` declarations

- Specify what all the elements that share a common name can contain (the *content model*)
- No direct support for namespaces
- Elements can contain
  - Nothing
  - Text only
  - Text and child elements
  - Just child elements

`ELEMENT` declaration syntax

- Empty:

```
<!ELEMENT name EMPTY>
```

- Text only:

```
<!ELEMENT name (#PCDATA)>
```

- Text and child elements:

```
<!ELEMENT name
  (#PCDATA|elem1|elem2|...)*>
```

Element content models

- Sequence of child elements:

```
<!ELEMENT name (ch1, ch2)>
```

- Choice between child elements:

```
<!ELEMENT name (ch1|ch2)>
```

- Sequence of optional child elements:

```
<!ELEMENT name
  (ch?, ch2?, ch3?)>
```

Element content models

- Repeated child elements (zero or more):

```
<!ELEMENT name (child*)>
```

- Repeated child elements (one or more):

```
<!ELEMENT name (child+)>
```

- Any of these possibilities can be freely combined

Element content models

- Choice between sequences:
  `((a, b) | (c, d)`
- Sequence of choices:
  `((a | b), (c | d))`
- Optional sequence:
  `(a, b, c)?`
- You cannot mix `,` and `|` in one list; use parentheses to disambiguate

Restrictions on content models

- A mixed model cannot constrain how many times or in what order the child elements appear, only which elements are allowed
- An element-only content model requires that each child element in the instance match exactly one part of the content model
  - `(A?, A?)` is not a legal content model

Element wildcard

- `<!ELEMENT name ANY>` specifies an element that can contain any child elements declared elsewhere in the DTD
- May include text as well
- Cannot include undeclared elements

`ATTLIST` declarations

- Declares the allowed attributes of an element
-
```
<!ATTLIST ename
      attr1 type1 default1
      attr2 type2 default2
      ...>
```

`ATTLIST` declarations

- Any number of attributes for a single element type may be declared in a single ATTLIST
- Any number of ATTLIST declarations may be used for a given element type
- The typical case: one ATTLIST per element type

Attribute types

- DTDs support only a few attribute types
- `CDATA`: no constraints
- `ID`: an identifier for the element
- `IDREF` : must match some element's ID
- `NMTOKEN` : a name or number

Attribute types

- `IDREFS`: one or more space-separated element IDs
- `NMTOKENS`: one or more space-separated names or numbers
- Enumerated tokens: value must be one

Attribute defaults

- Specific value required (supplied by parser):

  ```
  #FIXED "value"
  ```

- Attribute required but no specific value:

  ```
  #REQUIRED
  ```

- Attribute not required, default value:

  ```
  "value"
  ```

- Attribute not required, no default value:

  ```
  #IMPLICIT
  ```

`ENTITY` declarations

- Declare a name for text to be inserted into a document
- Entity references are of the form `&name;`
- The text is typically just one character long
- There is a long list of ISO-standardized character names
- General entities have *no equivalents* in other schema languages

Parameter entity declarations

- Declare a name for text (a single string or a whole file) to be inserted into a DTD;
- Allow parameterization of DTDs through included/ignored sections
- A trick involving parameter entities makes it possible to handle namespace prefixes
- Various specialized rules make parameter entities difficult to use

Obsolescent features

- Unparsed entities
  - Non-XML objects
  - Declared by `<!ENTITY ...>` declarations
  - Referred to by attributes of type `ENTITY` or `ENTITIES`
- `NOTATION` declarations
  - Specifies type of an unparsed entity
  - Specifies type of a text-only element using attributes of type `NOTATION`

RELAX NG GOALS

"DTDs on warp drive"

- An evolution/generalization of DTDs
- Shares the same basic paradigm
- Based on experience with SGML, XML
- Adds and subtracts features from DTDs
- DTDs can be automatically converted

Reusable knowledge

- Experts in designing SGML and XML DTDs will find their skills transfer easily
- Design patterns commonly used in XML DTDs can be reused
- Much more mature than if based on a completely new and different paradigm
- Higher degree of confidence in its design is possible

Easy to learn and use

- Allows schemas to be patterned after the structure of the documents they describe
- Allows definitions to be composed from other definitions in a variety of ways
- Treats attributes and elements as uniformly as possible

Namespaces

- DTDs are namespace-blind
- RELAX NG fully supports namespaces for elements and attributes
- Namespace support is purely syntactic, not tied to one schema per namespace
- Name classes support “any name” and “any name in specified namespace”

Datatyping

- Supports pluggable simple datatype libraries
- Basic library supports strings and tokens
- Full XML Schema Part 2 datatypes available (including facets)
- New libraries can be readily designed and built as needed.

Composability

- Schema languages provide atomic objects (elements, attributes, text, typed data) and methods of composing them (sequence, repetition, choice)
- All RELAX NG atomic objects can be composed with any available method
- Improves ease of learning, use, power; decreases complexity

Closure

- RELAX NG is closed under union
  - If two schemas exist describing two document types, then a schema describing the union of the two document types is trivial to create
  - Consequently, the content model of an element can be context-dependent

Two syntaxes, one language

- Provides two interconvertible syntaxes:
  - an XML one for processing
  - a compact non-XML one for human authoring
- We will learn the compact syntax
- One example of the XML syntax is provided to assist in learning it

Attributes

- “Elements or attributes?”
  - Reasonable people can differ
  - Attributes are treated as much like elements as possible
- Content models include elements as well as attributes
- Attribute defaulting is not done

Non-goal: attribute defaulting

- Attribute defaulting can only be done by DTDs or W3C XML Schema when the value does not depend on context
- Sensible attribute defaults often depend on context (inheritance of xml:lang, e.g.)
- Attribute defaulting is trivial for transformation languages such as XSLT

Non-goal: PSVI

- RELAX NG has no post-validation infoset enhancement
- Infoset enhancement can be done as a separate layer
- Sun's Multi-Schema Validator provides datatype information
- Separation of concerns promotes efficiency, flexibility

Annotations

- Annotations in the form of elements and attributes can be interspersed in RELAX NG schemas for various purposes:
  - DTD-style attribute defaults
  - documentation
  - embedded Java code
- Conforming RELAX NG validators ignore annotations

Mixed content

- SGML had problems with complex mixed-content models
- XML DTDs tightly restrict mixed-content models
- RELAX NG allows character content mixed with any content model

Unordered content

- SGML's `&` operator allows unordered content models
  - `A & B` means `((A, B) | (B, A))`
- XML DTDs removed & to reduce implementation complexity
- RELAX NG restores & with improvements

Customization

- Definitions included from another schema can be overridden
- Multiple definitions from the same or different schemas can be intelligently combined
  - as if with |
  - as if with &

A real standard

- Standardized in OASIS by the RELAX NG Technical Committee
- A major component of ISO DSDL, the Document Schema Definition Languages umbrella-standard

Non-goal: inheritance

- Inheritance-based schemas only model single inheritance
- Modeling often requires multiple inheritance (at least for interfaces)
- Schema languages are really about syntactic details, not about models

Non-goal: identity constraints

- Identity constraints are not supported
- Identity constraints are still a developing research area
- Different applications have different requirements from simple to complex
- Some RELAX NG tools support DTD-style semantics for ID, IDREF(S)

Non-goal: schema binding

- There is no standard way for a document to specify “its schema”
- Receivers often want to verify against agreed-on schemas, not sender-specified ones
- Documents may be validated against different schemas for different purposes
- The validation model takes two inputs: a document and a schema
- Just part of the XML processing issue

Interoperability

- You can convert a DTD to RELAX NG, preserving modularity
- You can author in RELAX NG and deliver as a DTD or a W3C XML Schema or both
- RELAX NG allows embedded Schematron rules

Pronunciation

- "Relaxing" is the standard way
- Some people say "relax en gee"

AND NOW TO RELAX!

THE INVOICE EXAMPLE

An invoice in XML

```
<invoice number="640959-0" date="2002-03-12">
  <soldTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </soldTo>
  <shipTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </shipTo>
  <terms>Net 10 days</terms>
  <item ordered="6" shipped="6" unitPrice="7.812">
     Binder, D-ring, 1.5"</item>
  <item ordered="4" shipped="2" backOrdered="2"
     unitPrice="3.44">Fork, Plastic, Heavy, Medium</item>
</invoice>
```

The invoice schema (1)

```
element invoice {
  attribute number { text },
  attribute date { text },
  element soldTo {
    element name { text },
    element address { text }
    },
  element shipTo {
    element name { text },
    element address { text }
    },
```

The invoice schema (2)

```
  element terms { text },
  element item {
    attribute unitPrice { text },
    attribute ordered { text },
    attribute shipped { text },
    attribute backOrdered { text }?,
    text
    }*
}
```

Things to note

- The structure of the schema parallels the structure of the document
- Element content models include attributes as well as child elements
- The optional attribute is marked with `?`
- `text` is the equivalent of #PCDATA or CDATA

Things to note

- Commas separate multiple components of a content model when the components appear in the given order
- Of course, the order of attributes does not matter!
- Consequently, attributes can appear in the schema before, after, or mixed in with child elements

The XML format

```
<element name="invoice">
    <attribute name="number"/>
    <attribute name="date"/>
    <element name="soldTo">
      <element name="name">
        <text/>
      </element>
      <element name="address">
        <text/>
      </element>
    </element>
    <element name="shipTo">
      <element name="name">
        <text/>
      </element>
      <element name="address">
        <text/>
      </element>
    </element>
```

```
<element name="terms">
     <text/>
   </element>
   <zeroOrMore>
     <element name="item">
       <attribute name=
  "unitPrice"/>
       <attribute
  name="ordered"/>
       <attribute
  name="shipped"/>
       <optional>
         <attribute
  name="backOrdered"/>
       </optional>
     </element>
   </zeroOrMore>
  </element>
```

Definition form

```
start = invoice
element invoice {
   attribute number { text },
   attribute date { text },
   soldTo, shipTo, terms, item
   }
soldTo = element soldTo { name, address }
shipTo = element shipTo { name, address }
terms = element terms { text }
```

Definition form

```
item = element item {
     attribute unitPrice { text },
     attribute ordered { text },
     attribute shipped { text },
     attribute backOrdered { text }?,
     text
     }

name = element name { text }

address = element address { text }
```

Definition form notes

- In definition form, there must always be a definition of `start`
- You refer to a rule using just its name
- The order of the rules does not matter; use whatever order makes sense to you (top-down, bottom-up, alphabetical)
- Rule names are only relevant to the schema, and *never* appear in the document instance

Three schema designs

- "Russian Doll": a single pattern for the whole document
- "Salami Slice": one definition for each differently-named element (DTD style)
- "Venetian Blind": one definition for each pattern specifying the content of an element, plus one for the document element

Venetian Blind schema

```
start = element invoice {invoice}
invoice =
   attribute number { text },
   attribute date { text },
   element soldTo { name-addr },
   element shipTo { name-addr },
   element terms { text },
   element item { item }*
```

Venetian Blind schema

```
item =
   attribute unitPrice { text },
   attribute ordered { text },
   attribute shipped { text },
   attribute backOrdered { text }?,
   text
name-addr =
   element name { text },
   element address { text }
```

BASIC PATTERNS

Patterns

- Patterns are the basic building blocks of RELAX NG schemas and rules
- Some kinds of patterns can contain sub-patterns enclosed in braces ({ … })

Element patterns

- Syntax: `element` *`name`* `{ ... }`
- The content model (child elements and attributes) is contained within the braces
- Content models consist of one or more patterns

Attribute patterns

- Syntax: `attribute` *`name`* `{ ... }`
- The content model is contained within the braces
- Content models consist of one or more patterns
- You can't have child elements or attributes within attributes, of course!

Attribute patterns

- So what patterns can be inside attributes?
  - The `text` pattern - equivalent to `CDATA`
  - Datatypes (details later)
  - Literal strings in quotes:
    `attribute country { "US" }`
    means the `country` attribute must have the value `US`.

Element patterns

- So what patterns can be inside elements?
  - The `text` pattern - equivalent to `#PCDATA`
  - Datatypes (details later)
  - Literal strings in quotes:
    `element country { "US" }`
    means the `country` element must have the content `US`.

The `text` pattern

- Matches any amount of arbitrary text, possibly broken up by child elements
- Equivalent to `#PCDATA` in elements or `CDATA` in attributes
- `text*`, `text?`, `text+` all mean the same as `text`

Namespaces

- To declare elements and attributes in namespaces, use QNames in element and attribute patterns
- Namespace prefixes are declared like this:

  ```
  namespace foo = "(some URI)"
  ```

- Namespace declarations must come first in the schema

Default namespaces

- You can declare a namespace for unprefixed elements (not attributes) like this:
```
default namespace =
    "(some URI)"
```
- If you want the default namespace to have a prefix too, use:
```
default namespace foo =
    "(some URI)"
```

Namespaces

Here's an example:

```
namespace one = "http://example.com/one"
namespace two = "http://example.com/two"
default namespace = "http://example.com"
element para {
  attribute one:class { text },
  attribute two:class { text },
  element line { text }*
  }
```

Choice

- Two patterns separated by | represent a choice between them; the document can match one pattern or the other, not both
- Arbitrary patterns are allowed in a choice: you can have a choice between attributes, between elements, or even between an element and an attribute!

Choice

- A useful case:

```
element data {
   (element id { text } |
    attribute id { text }),
   text
   }
```

- You cannot mix `,` and `|` in one list; use parentheses to disambiguate

Choice

Enumerated values use choice like this:

```
element font {
  attribute size {
    "10" | "12" | "14" | "16"
  }
}
```

Quantifiers

- You can place an `*`, `?`, or `+` after any pattern to allow it to be repeated:
  - `*` means zero or more times
  - `?` means zero or one times
  - `+` means one or more times
- These mean the same as in DTD content models, but can be used after any pattern, not just rule names

MORE PATTERNS

Interleave

- Interleave is a cross between choice and sequence
- When patterns are combined with &, they all must appear but it can be in any order (as in SGML) …
- … or even mixed together!

Interleave

- So this schema ...

```
element head {
   element meta { empty }* &
   element title { text }
   }
```

- … matches a `head` element that has any number of `meta` child elements (including zero) and a required `title` child element *mixed in anywhere*.

Interleave

- Note: In the case of attributes, sequence and interleave are the same thing, because attributes don't have ordering
- So you can use either `,` or `&` according to what is the most convenient
- The pattern `mixed { ... }` is a synonym for `(text & ( ... ))`

Multiple element and attribute patterns

- An element or attribute pattern with multiple names separated by `|` matches elements bearing any of those names:

```
element h1|h2|h3|h4|h5|h6
{ heading.model }
```

Element wildcards

- An element pattern with `*` instead of a name matches an element with any name
- To match any name in a particular namespace, use *`foo`*`:*` where *`foo`* is the prefix declared for the namespace
- To match any name except `foo` and `bar`, use `* - (foo|bar)`

Attribute wildcards

- Attribute wildcards are declared just like element wildcards
- An attribute wildcard pattern must be followed by `*` or `+`
- It makes no sense to specify an element with "just one wildcard attribute"

ANY?

- There is no built-in ANY content model, corresponding to `empty` for empty content models or `notAllowed` for forbidden models
- Here's how it can be done:

```
ANY = element * {
   attribute * {text}*
   & text & ANY*
}
```

Multiple schemas

- `external` incorporates one pattern document into another
- `include` incorporates one set of rules into another, and allows for overriding any of the included rules by name
  - In particular, overriding the start rule is usually necessary
- Rules with identical names can also be combined by choice or by interleave

Context sensitivity

The first paragraph cannot have footnotes; the remainder can:

```
start = element doc {first, other*}

first = element para { text }

other = element para {
  mixed { element footnote {text}*}
}
```

Restrictions on schemas

- The obvious XML ones: no elements or attributes within attributes, only one top-level element, etc. etc.
- Attributes can't have conflicting definitions in a single element

Restrictions on schemas

- Interleave doesn't allow an element with a given name, or text either, to be used in more than one interleaved subpattern
- The following are *illegal:*
  - (a, b, c) & (a, d, e)
  - (

A few lexical details

- Strings (used as values and parameter values) can be wrapped in double quotes or single quotes
- Multi-line strings are wrapped in `"""` or `'''` (as in Python)
- Rule names that are the same as syntactic keywords must be preceded by `\`

Comments

- Ordinary comments begin with `#`
- Documentation comments begin with `##` and are copied (in groups) into the XML syntax as `a:documentation` elements
- The `a:` prefix represents the namespace of the DTD Compatibility extension to RELAX NG

DATATYPES

Datatypes

- A type is a named set of values
- An datatype provides a standardized, machine-checkable representation of a type

Schema datatypes

- DTDs have only a few datatypes for attributes and only one datatype for elements
- XML Schema provides a long, but fixed, list of datatypes
- RELAX NG can work with any datatype library, including the XSD (XML Schema Datatypes) library

RELAX NG datatypes

- Datatype patterns are written using QNames
- This use of QNames can't be confused with QNames for elements or attributes, because those are only recognized after the words `element` and `attribute`
- The built-in datatypes `string` and `token` don’t have prefixes and are recognized by all implementations

Declaring datatype libraries

- A prefix is declared like this:
  `datatypes lib = "(some URI)"`
- Datatype library declarations must come first
- RELAX NG processors recognize a system-dependent list of datatype library URIs

Useful datatypes

- The `xsd` prefix is predeclared for XML Schema Datatypes
- `xsd:integer` represents an integer of arbitrary length
- There are `xsd:` equivalents for all the DTD attribute types (`ID`, `IDREF`, etc.)
- We'll discuss all the `xsd` types later.

Typed values

- `"0"` and `token "0"` match a “0” character with possible surrounding whitespace
- `string "0"` matches a “0” character exactly
- `xsd:integer "0"` matches “0” or “00” or “000” or “-0” or ...

Simple and complex content

- Element patterns containing a datatype or a value (or a list) specify *simple content*
- Element patterns containing child elements or text or both specify *complex content*
- No element pattern can contain both
  - A choice between simple and complex content is legal

Datatype exceptions

- `xsd:nonNegativeInteger` and `xsd:nonPositiveInteger` are existing types
- How do we say “non-zero integer”? `xsd:integer - xsd:integer "0"`
- We can likewise express a string that is not a name: `xsd:string - xsd:Name`

Parameters

- Parameters restrict the values of datatypes
- Each datatype has specific parameters are legal with it
- An integer between 0 and 999 inclusive:

```
xsd:integer {
  minInclusive = "0"
  maxInclusive = "999"
}
```

Lists

- List patterns specify that simple content is to be divided by whitespace into tokens
- The pattern `list {xsd:integer*}` matches a list of zero or more white-space-separated integers
- This example needs * because `list` itself does not imply repetition

Lists

- The pattern

  ```
  list {(xsd:integer, token)+}
  ```

  matches the string

  "32 foo 45 bar 76 baz"
- Lists within lists are not allowed

TOOLS

The Jing validator

- Written by James Clark, principal author of RELAX NG
- Java based command-line tool
  - Validates schemas
  - Validates documents against schemas
- Accepts either compact or XML syntax
- Optionally enforces DTD ID/IDREF

The Jing validator

- Also usable as a validation library within a Java program
- Provides JAXP (Sun-standard) interface
- Provides native interface
- Validates against other schema languages:
  - W3C XML Schema
  - Schematron
  - Namespace Routing Language

The Trang translator

- Another James Clark product
- Translates schemas:
  - Input: XML syntax, compact syntax, DTDs
  - Output: XML syntax, compact syntax, DTDs, W3C XML Schema
- Output schemas may be looser than the input schema (accept a superset of what the input accepts)
- DTD and W3C XML Schema output is imperfect

Sun RELAX NG translator

- Translates other schema languages into RELAX NG in XML syntax:
  - DTDs
  - RELAX Core/Namespace
  - TREX (predecessor of RELAX NG)
  - Subset of XML Schema
- Does not preserve schema structure

Instance to schema

- InstanceToSchema generates a RELAX NG schema from one or more XML instances
- Examplotron *(www.examplotron.org)* is a schema language that resembles an instance with optional annotations and is translated into RELAX NG

Sun Multi-Schema Validator

- By Kohsuke KAWAGUCHI, a major RELAX NG contributor
- Validates documents (command-line or library) using any schema language supported by the Sun RELAX NG Translator
- Also handles stand-alone or embedded Schematron rules

Validation in .NET

- Tenuto is a C# implementation of validation for the Common Language Runtime environment
- Supports XML syntax, XSD library
- Does not support ID/IDREF semantics
- RelaxngValidatingReader is an unrelated implementation of XMLReader that validates input against a RELAX NG schema

The Bali validator generator

- Accepts a RELAX NG schema and generates special-purpose code to validate documents against that particular schema
- The generated validator is faster and smaller than a general-purpose validator
- Generates Java, C++, or C#

The VBRELAXNG validator

- A validator for the XML syntax written in Visual Basic 6.0
- Provides validation as an ActiveX control
- Requires MSXML 4.0 parser
- Topologi Schematron Validator uses this control

The RelaxNGCC compiler

- Accepts a subset of RELAX NG (no ambiguous grammars) with annotations; RelaxMeter tool checks for ambiguities
- Compiles specified RELAX NG rules into Java classes
- Embedded Java code can refer to the values matched by datatype, text, and list patterns
- Analogous to JavaCC

The RelaxNGCC compiler

- The generated code requires a source of SAX events (typically a parser)
- Objects of the generated classes are data bindings for RELAX NG rules considered as types
- The killer app for RELAX NG?

The RELAXER compiler

- Generates Java objects from RELAX NG schemas without embedded code
- Imposes slightly different restrictions on schemas (weaker if DOM available)
- Provides serialization from Java as well as parsing to Java
- Analogous to JAXB
- The killer app for RELAX NG?

Editors

- oXygen and Topologi editors validate against RELAX NG schemas
- xmloperator and an Emacs mode allow editing guided by a RELAX NG schema
- Vim syntax coloring for writing schemas in the compact syntax

The libxml2 library

- Provides parsing and (RELAX NG) validation services for C programs
- Packaged with xmllint, an XML parser and RELAX NG validator
- Part of the GNOME desktop, but available separately

NAMESPACE-BASED VALIDATION DISPATCHING LANGUAGE

What it is

- An NVDL schema specifies how to divide a document into parts and apply multiple subschemas to them
- Basically it's a map between namespace names and schema URIs
- Implementations aren't stable yet, but the concepts are important

How it works

- The document is carved up into subtrees whose elements have a common namespace name
- The subschema associated with that name is used to validate the subtree
- Subschemas can be in any schema language including XML Schema and even DTD

Concurrent validation

- Multiple schemas can be applied to a subtree to cause concurrent validation
- Example: XHTML validation with RELAX NG requires a check that no `a` element has another `a` element as its ancestor
- A small schema that must *fail* validation can be used to enforce this limtation

Other features

- A subordinate namespace can be attached to a superior namespace so that they will be validated together by the superior namespace's schema
- Attribute-only schemas can be supported using a dummy element name
- Validation can begin at a specific element specified by a small subset of XPath

Tools

- Currently there are only a few validators
- Jing actually validates Namespace Routing Language, an earlier version of NVDL
- This situation should improve

THE SCHEMATRON

Rule-based validation

- Grammars can't do it all
- The Schematron allows you to write rules that test almost anything about a document
- Schematron implementations display natural-language diagnostics when:
  - An assertion is violated
  - The schema specifies a report

An invoice in XML

```
<invoice number="640959-0" date="2002-03-12">
  <soldTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </soldTo>
  <shipTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </shipTo>
  <terms>Net 10 days</terms>
  <item ordered="6" shipped="6" unitPrice="7.812">
     Binder, D-ring, 1.5"</item>
  <item ordered="4" shipped="2" backOrdered="2"
     unitPrice="3.44">Fork, Plastic, Heavy, Medium</item>
</invoice>
```

Schematron example (1)

```
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern>
    <rule context="/invoice">
     <assert test="soldTo/name = shipTo/name">Sold-to and
ship-to names must match.</assert>
    </rule>
  </pattern>
...
```

Patterns and rules

- A Schematron pattern is a list of rules to apply against each element in the document
- Rules are applied in the order they appear
- Only the first successful rule in a pattern is applied
- Each rule specifies an XSLT pattern in its `context` attribute

- Schemas can contain multiple patterns

Schematron example (2)

```
  <pattern>
    <rule context="/invoice/item">
     <report test="@ordered = @shipped" diagnostic="d1">
       <emph>Partial shipment</emph></assert>
     <report test="true()">Complete shipment.</report>
    </rule>
  </pattern>
  <diagnostics>
    <diagnostic id="d1"><value-of select="@ordered -
@shipped"/> items to follow.</diagnostic>
  </diagnostics>
...
```

Reports vs. assertions

- Both reports and assertions specify an Xpath expression to test in a `test` attribute
- Reports display a message if the test passes
- Assertions display a message if the test fails
- The `emph` element marks up emphatic parts of the message

Diagnostics

- Assertions and reports can use the `diagnostic` attribute to point to a `diagnostic` element by ID
- Diagnostics are placed in the top-level `diagnostics` element
- Diagnostics can contain `value-of` child elements to output computed values

Internationalization

- `diagnostic` elements can use an `xml:lang` attribute to specify the language of the diagnostic
- `assert` and `report` elements can point to multiple diagnostics in different languages
- The diagnostic which matches the locale will be used and the rest ignored

Namespaces

- The top-level `ns` element has `prefix` and `uri` attributes with the obvious meanings
- This specifies the mapping used inside the Xpaths; unprefixed names are in no namespace, as usual
- Direct namespace declarations in the schema may also work but are implementation-dependent

Phases

- A single schema can contain multiple phases
- At run-time a phase is chosen and only the patterns that are active in that phase are considered
- A pattern may be active in multiple phases
- A `phase` element specifies a phase using `active` child elements that point to `pattern` elements by ID.

Other features

- The `include` element brings in externally stored schema portions
- The `title` and `p` elements allow embedded documentation
- Elements and attributes from foreign namespaces are allowed
- The `let` element binds a variable for use in XPath (ISO extension)

Abstract rules and patterns

- ISO extensions not yet implemented by most Schematron systems
- Abstract rules are ignored, but ordinary rules can inherit assertions and report elements from them
- Abstract patterns are ignored, but concrete patterns can inherit rules from

MORE INFORMATION

http://www.relaxng.org

http://www.dsdl.org

http://www.schematron.com

http://www.ccil.org/~cowan/