Relax NG

Intruduction

RELAX NG is a schema language for XML. Main features are:

- is simple
- is easy to learn
- has both an XML syntax and a compact non-XML syntax
- does not change the information set of an XML document
- supports XML namespaces
- treats attributes uniformly with elements so far as possible
- has unrestricted support for unordered content
- has unrestricted support for mixed content
- has a solid theoretical basis
- can partner with a separate datatyping language (such W3C XML Schema Datatypes)

The RELAX NG specifications have been developed within OASIS by the RELAX NG Technical Committeee. RELAX NG is also an International Standard (ISO/IEC 19757-2). It is Part 2 of ISO/IEC 19757 DSDL (Document Schema Definition Languages), which is maintained by ISO/IEC JTC1/SC34/WG1. RELAX NG was based on TREX designed by James Clark and RELAX Core designed by MURATA Makoto.

Comparison with XML DTDs

RELAX NG provides functionality that goes beyond XML DTDs. In particular, RELAX NG

- uses XML syntax to represent schemas
- supports datatyping
- integrates attributes into content models
- supports XML namespaces
- supports unordered content
- supports context-sensitive content models

RELAX NG does not support features of XML DTDs that involve changing the infoset of an XML document. In particular, RELAX NG

- does not allow defaults for attributes to be specified; however, this is allowed by RELAX NG DTD Compatibility [Compatibility]
- does not allow entities to be specified
- does not allow notations to be specified
- does not specify whether whitespace is significant
- Also RELAX NG does not define a way for an XML document to associate itself with a RELAX NG pattern.

Software - Validators

Jing, RELAX NG validator. Supports both XML and compact syntaxes. Can be used as a library for validation with any SAX2 parser. Implemented in Java. Written by James Clark. Open source (BSD license).

RNV Relax NG Compact Syntax validator in ANSI C (under BSD license)

Libxml2, the XML C library for Gnome. Recent versions of libxml2 (at least 2.5.5) support RELAX NG validation. Only the XML syntax is supported. Libxml2 is included in most modern Linux distributions. Implemented in C. Written by Daniel Veillard. Open source (MIT license).

MSV, Sun Multi-Schema XML Validator. Validates RELAX NG. Supports XML syntax only. Also supports other schema languages including W3C XML Schema. Can be used as a library for validation and for accessing the schema. Implemented in Java. Written by Kohsuke KAWAGUCHI. Open source (BSD license).

Bali, RELAX NG validator compiler. Reads a RELAX NG schema and produces a validator that can validate documents with the given schema. Compiler uses MSV and is implemented in Java. Generated validator can be in Java, C++ for Win32 (using MSXML4) or C#. Written by Kohsuke KAWAGUCHI. Open source (BSD license).

Tenuto, RELAX NG validator for .NET. Implemented in C#. Open source (BSD license).

VBRELAXNG, ActiveX DLL for validating RELAX NG. RELAX NG Workshop sample client application also available. Uses MSXML4. Implemented in Visual Basic. Written by YONEKURA Koji. Freely downloadable.

XVIF, XML Validation Interoperability Framework. Proof of concept implementation of the idea of embedding XML processing pipelines in a grammar. Includes a partial implementation of RELAX NG. Implemented in Python. Written By Eric van der Vlist. Open source (MPL).

RelaxngValidatingReader, .NET XmlReader that validates using RELAX NG. Implemented in C#. Included in Mono. Written by Atsushi Enomoto. Open source (public domain).

On-line validator. It supports both RELAX NG Compact and XML syntax.

ManekiNeko, Xerces parser configuration that supports RELAX NG validation. Puts an XNI wrapper around Jing. Implemented in Java. Written by Andy Clark. Open source (Apache-style license).

Sun MSV Schematron add-on, allows a document to be validated against a RELAX NG grammar and also against Schematron constraints embedded as annotations in the RELAX NG. Uses MSV. Implemented in Java. Written by Kohsuke KAWAGUCHI. Open source (BSD license).

Software - Conversion tools

- **Trang**, multi-format schema converter based on RELAX NG. Trang supports the following languages: RELAX NG (both XML and compact syntax), XML 1.0 DTDs, W3C XML Schema. A schema written in any of the supported schema languages can be converted into any of the other supported schema languages, except that W3C XML Schema is supported for output only, not for input. Trang can also infer a schema from one or more example XML documents. Trang aims to produce human-understandable schemas; it tries to preserve all aspects of the input schema that may be significant to a human reader, including the definitions, the way the schema is divided into files, annotations and comments. Uses Jing. Implemented in Java. Written by James Clark. Open source (BSD license).

- **Sun RELAX NG Converter**. The Sun RELAX NG Converter is a tool to convert schemas written in various schema languages to their equivalent in RELAX NG. It supports schemas written in XML DTD, RELAX Core, RELAX namespace, TREX, W3C XML Schema, and RELAX NG itself. It does aims only to produce a RELAX NG schema that is equivalent to the input schema in the sense that it validates the same documents as the input schema; it does not aim to preserve information that is not significant for validation such as the use of definitions. Based on MSV. Implemented in Java. Written by Kohsuke KAWAGUCHI. Open source (BSD license).

- **XSD to RelaxNG**. web-based converter from W3C XML Schema to RELAX NG. The user types a WXS schema in this web form, and gets a RELAX NG schema with the convert button. The conversion is done by an XSLT stylesheet.

- **InstanceToSchema**, a tool to generate a RELAX NG schema from XML instances. Implemented in Java. Written by Didier Demany. Open source (BSD license).

- **rng2srng**, a tool to convert RELAX NG (XML or compact syntax) into the simple syntax. The simple syntax is a minimal subset of the XML syntax defined by the RELAX NG specification, into which any RELAX NG schema can be transformed. Uses Jing. Implemented in Java. Written by Kohsuke KAWAGUCHI. Open source (BSD license).

- **NekoDTD**, DTD to instance converter. Uses the Xerces Native Interface (XNI) to convert DTD into an XML document, which can then be converted into other formats. Includes XSLT stylesheets to convert XML output format into RELAX NG. Implemented in Java. Written by Andy Clark. Open source (Apache-style license).

Software - Code generators

- **Relaxer**, schema compiler. Can generate a collection of Java classes from a RELAX NG schema along with code to create instances of those classes from XML and vice-versa. Provides many other features useful for processing data described by a RELAX NG schema. Uses MSV. Implemented in Java. Written by ASAMI Tomoharu. Open source (GPL for compiler, BSD license for generated code, LGPL for runtime libraries).

- **RelaxNGCC**, RELAX NG Compiler Compiler. Tool for generating Java source code from a given RELAX NG grammar. By embedding code fragments in the grammar like yacc or JavaCC, you can take appropriate actions while parsing valid XML documents against the grammar. Uses MSV. Implemented in Java. Written by Daisuke OKAJIMA and Kohsuke KAWAGUCHI. Open source (GPL for compiler, public domain for generated code).

Software - XML editors

- **Stylus Studio** now supports RELAX NG!
- **Firedocs** is a browser based wysiwyg-xml-editor that has schema-driven auto-complete and uses Jing for validation. Licence: ASL 2.0
- **xmloperator**, an XML editor, suitable for editing data oriented documents. Allows any RELAX NG schema to be used to guide editing. Implemented in Java. Written by Didier Demany. Open source (BSD license).
- **Topologi Collaborative Markup Editor**, XML editor including support for RELAX NG. Commercial.
- **<oXygen/> XML editor**. Provides validation and completion using RELAX NG. Supports both XML and compact syntaxes. Multiplatform, implemented in Java. Commercial.
- **XMLBlueprint XML Editor** Provides validation and completion using RELAX NG. Commercial.
- **XMLBuddy Pro** Provides validation and completion using RELAX NG. Commertical.
- **RNGEdit** (in Japanese)

Getting started

For below XML:

```
<addressBook>
  <card>
    <name>John Smith</name>
    <email>js@example.com</email>
  </card>
  <card>
    <name>Fred Bloggs</name>
    <email>fb@example.net</email>
  </card>
</addressBook>
```

Getting started

DTD looks like as follows:

```
<!DOCTYPE addressBook [
<!ELEMENT addressBook (card*)>
<!ELEMENT card (name, email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
]>
```

Getting started

A RELAX NG pattern for this could be written as follows:

```
<element name="addressBook" xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

oneOrMore

If the addressBook is required to be non-empty, then we can use *oneOrMore* instead of *zeroOrMore*:

```
<element name="addressBook" xmlns="http://relaxng.org/ns/structure/1.0">
  <oneOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </oneOrMore>
</element>
```

Optional element

Let's each *card* to have an optional *note* element:

```
<element name="addressBook" xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
      <optional>
          <element name="note">
            <text/>
          </element>
      </optional>
    </element>
  </zeroOrMore>
</element>
```

Note that the text pattern matches arbitrary text, including empty text. Note also that whitespace separating tags is ignored when matching against a pattern. All the elements specifying the pattern must be namespace qualified by the namespace URI: `http://relaxng.org/ns/structure/1.0`

Namespace prefix

The examples above use a default namespace declaration `xmlns="http://relaxng.org/ns/structure/1.0"` for this. A namespace prefix is equally acceptable:

```
<rng:element name="addressBook" xmlns:rng="http://relaxng.org/ns/structure/1.0">
  <rng:zeroOrMore>
    <rng:element name="card">
      <rng:element name="name">
        <rng:text/>
      </rng:element>
      <rng:element name="email">
        <rng:text/>
      </rng:element>
    </rng:element>
  </rng:zeroOrMore>
</rng:element>
```

Choice

Now suppose we want to allow the *name* to be broken down into a *givenName* and a *familyName*, allowing an *addressBook* like this:

```
<addressBook>
  <card>
    <givenName>John</givenName>
    <familyName>Smith</familyName>
    <email>js@example.com</email>
  </card>
  <card>
    <name>Fred Bloggs</name>
    <email>fb@example.net</email>
  </card>
</addressBook>
```

Choice

We can use the following pattern:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <choice>
        <element name="name">
          <text/>
        </element>
        <group>
          <element name="givenName">
            <text/>
          </element>
          <element name="familyName">
            <text/>
          </element>
        </group>
      </choice>
      <element name="email">
        <text/>
      </element>
      <optional>
       <element name="note">
         <text/>
       </element>
      </optional>
    </element>
  </zeroOrMore>
</element>
```

This corresponds to the following DTD:

```
<!DOCTYPE addressBook [
<!ELEMENT addressBook (card*)>
<!ELEMENT card ((name | (givenName, familyName)), email, note?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT givenName (#PCDATA)>
<!ELEMENT familyName (#PCDATA)>
<!ELEMENT note (#PCDATA)>
]>
```

Attributes

Suppose we want the card element to have attributes rather than child elements. The DTD might look like this:

```
<!DOCTYPE addressBook [
<!ELEMENT addressBook (card*)>
<!ELEMENT card EMPTY>
<!ATTLIST card
  name CDATA #REQUIRED
  email CDATA #REQUIRED>
]>
```

Just change each element pattern to an attribute pattern:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <attribute name="name">
        <text/>
      </attribute>
      <attribute name="email">
        <text/>
      </attribute>
    </element>
  </zeroOrMore>
</element>
```

Order

In XML, the order of attributes is traditionally not significant. RELAX NG follows this tradition. The above pattern would match both

```
<card name="John Smith"
email="js@example.com"/>
```

and

```
<card
email="js@example.com"
name="John Smith"/>
```

In contrast, the order of elements is significant. The pattern

```
<element name="card">
  <element name="name">
    <text/>
  </element>
  <element name="email">
    <text/>
  </element>
</element>
```

would not match

```
<card><email>js@example.com</
email><name>John Smith</name></
card>
```

Note that an attribute element by itself indicates a **required** attribute, just as an element element by itself indicates a **required** element. To specify an **optional** attribute, use **optional** just as with element:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <attribute name="name">
        <text/>
      </attribute>
      <attribute name="email">
        <text/>
      </attribute>
      <optional>
        <attribute name="note">
          <text/>
        </attribute>
      </optional>
    </element>
  </zeroOrMore>
</element>
```

The **group** and **choice** patterns can be applied to attribute patterns in the same way they are applied to element patterns. For example, if we wanted to allow either a name attribute or both a *givenName* and a *familyName* attribute, we can specify this in the same way that we would if we were using elements:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <choice>
        <attribute name="name">
          <text/>
        </attribute>
        <group>
          <attribute name="givenName">
            <text/>
          </attribute>
          <attribute name="familyName">
            <text/>
          </attribute>
        </group>
      </choice>
      <attribute name="email">
        <text/>
      </attribute>
    </element>
  </zeroOrMore>
</element>
```

The **group** and **choice** patterns can combine element and attribute patterns without restriction. For example, the following pattern would allow a choice of elements and attributes independently for both the *name* and the *email* part of a *card*:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <choice>
           <element name="name">
             <text/>
           </element>
           <attribute name="name">
             <text/>
           </attribute>
      </choice>
      <choice>
           <element name="email">
             <text/>
           </element>
           <attribute name="email">
             <text/>
           </attribute>
      </choice>
    </element>
  </zeroOrMore>
</element>
```

As usual, the relative order of elements is significant, but the relative order of attributes is not. Thus the above would match any of:

```
<card name="John Smith" email="js@example.com"/>
<card email="js@example.com" name="John Smith"/>
<card email="js@example.com"><name>John Smith</name></card>
<card name="John Smith"><email>js@example.com</email></card>
<card><name>John Smith</name><email>js@example.com</email></card>
```

However, it would not match

```
<card><email>js@example.com</email><name>John
Smith</name></card>
```

because the pattern for card requires any email child element to follow any name child element.

There is one difference between attribute and element patterns: *<text/>* is the default for the content of an attribute pattern, whereas an element pattern is not allowed to be empty. For example,

```
<attribute name="email"/>
```

is short for

```
<attribute name="email">
  <text/>
</attribute>
```

It might seem natural that

```
<element name="x"/>
```

matched an *x* element with no attributes and no content. However, this would make the meaning of empty content inconsistent between the element pattern and the attribute pattern, so RELAX NG does not allow the element pattern to be empty. A pattern that matches an element with no attributes and no children must use *<empty/>* explicitly:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
      <optional>
        <element name="prefersHTML">
          <empty/>
        </element>
      </optional>
    </element>
  </zeroOrMore>
</element>
```

Even if the pattern in an element pattern matches attributes only, there is no need to use empty. For example,

```
<element name="card">
  <attribute name="email">
    <text/>
  </attribute>
</element>
```

is equivalent to

```
<element name="card">
  <attribute name="email">
    <text/>
  </attribute>
  <empty/>
</element>
```

Named patterns

For a non-trivial RELAX NG pattern, it is often convenient to be able to give names to parts of the pattern. Instead of

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

we can write

```
<grammar>

  <start>
    <element name="addressBook">
      <zeroOrMore>
         <element name="card">
              <ref name="cardContent"/>
         </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="cardContent">
    <element name="name">
      <text/>
    </element>
    <element name="email">
      <text/>
    </element>
  </define>

</grammar>
```

A grammar element has a single **start** child element, and zero or more **define** child elements.

The **start** and **define** elements contain patterns. These patterns can contain **ref** elements that refer to patterns defined by any of the **define** elements in that grammar element.

A **grammar** pattern is matched by matching the pattern contained in the **start** element.

We can use the **grammar** element to write patterns in a style similar to DTDs:

```
<grammar>

  <start>
    <ref name="AddressBook"/>
  </start>

  <define name="AddressBook">
    <element name="addressBook">
      <zeroOrMore>
        <ref name="Card"/>
      </zeroOrMore>
    </element>
  </define>

  <define name="Card">
    <element name="card">
      <ref name="Name"/>
      <ref name="Email"/>
    </element>
  </define>

  <define name="Name">
    <element name="name">
      <text/>
    </element>
  </define>

  <define name="Email">
    <element name="email">
      <text/>
    </element>
  </define>

</grammar>
```

Recursive references are allowed. For example,

```
<define name="inline">
  <zeroOrMore>
    <choice>
      <text/>
      <element name="bold">
        <ref name="inline"/>
      </element>
      <element name="italic">
        <ref name="inline"/>
      </element>
      <element name="span">
        <optional>
          <attribute name="style"/>
        </optional>
        <ref name="inline"/>
      </element>
    </choice>
  </zeroOrMore>
</define>
```

However, recursive references must be within an element. Thus, the following is not allowed:

```
<define name="inline">
  <choice>
    <text/>
    <element name="bold">
      <ref name="inline"/>
    </element>
    <element name="italic">
      <ref name="inline"/>
    </element>
    <element name="span">
      <optional>
         <attribute name="style"/>
      </optional>
      <ref name="inline"/>
    </element>
  </choice>
  <optional>
    <ref name="inline"/>
  </optional>
</define>
```

Datatyping

RELAX NG allows patterns to reference externally-defined datatypes, such as those defined by [*W3C XML Schema Datatypes*]. RELAX NG implementations may differ in what datatypes they support. One must uses datatypes that are supported by the implementation one plans to use.

The **data** pattern matches a string that represents a value of a named **datatype**. The **datatypeLibrary** attribute contains a URI identifying the library of datatypes being used. The datatype library defined by [*W3C XML Schema Datatypes*] would be identified by the URI http://www.w3.org/2001/XMLSchema-datatypes.

The **type** attribute specifies the name of the datatype in the library identified by the **datatypeLibrary** attribute. For example, if a RELAX NG implementation supported the datatypes of [*W3C XML Schema Datatypes*], you could use:

```
<element name="number">
  <data type="integer" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"/>
</element>
```

It is inconvenient to specify the **datatypeLibrary** attribute on every *data* element, so RELAX NG allows the **datatypeLibrary** attribute to be inherited.

The **datatypeLibrary** attribute can be specified on any RELAX NG element. If a data element does not have a **datatypeLibrary** attribute, it will use the value from the closest ancestor that has a **datatypeLibrary** attribute.

Typically, the **datatypeLibrary** attribute is specified on the root element of the RELAX NG pattern. For example,

```
<element name="point" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <element name="x">
    <data type="double"/>
  </element>
  <element name="y">
    <data type="double"/>
  </element>
</element>
```

If the children of an element or an attribute match a data pattern, then complete content of the element or attribute must match that data pattern. It is not permitted to have a pattern which allows part of the content to match a data pattern, and another part to match another pattern.

For example, the following pattern is not allowed:

```
<element name="bad">
  <data type="int"/>
  <element name="note">
    <text/>
  </element>
</element>
```

However, this would be fine:

```
<element name="ok">
  <data type="int"/>
  <attribute name="note">
    <text/>
  </attribute>
</element>
```

Note that this restriction does not apply to the **text** pattern.

Datatypes may have parameters.

For example, a **string** datatype may have a parameter controlling the **length** of the string.

The parameters applicable to any particular datatype are determined by the **datatyping vocabulary**.

Parameters are specified by adding one or more *param* elements as children of the *data* element.

For example, the following constrains the email element to contain a string at most 127 characters long:

```
<element name="email">
  <data type="string">
    <param name="maxLength">127</param>
  </data>
</element>
```

Enumerations

Many markup vocabularies have attributes whose value is constrained to be one of set of specified values. The **value** pattern matches a string that has a specified **value**. For example,

```
<element name="card">
  <attribute name="name"/>
  <attribute name="email"/>
  <attribute name="preferredFormat">
    <choice>
      <value>html</value>
      <value>text</value>
    </choice>
  </attribute>
</element>
```

The **value** pattern is not restricted to attribute values.
For example, the following is allowed:

```
<element name="card">
  <element name="name">
    <text/>
  </element>
  <element name="email">
    <text/>
  </element>
  <element name="preferredFormat">
    <choice>
      <value>html</value>
      <value>text</value>
    </choice>
  </element>
</element>
```

The prohibition against a **data** pattern's matching only part of the content of an element also applies to value patterns.

By default, the **value** pattern will consider the string in the pattern to match the string in the document if the two strings are the same after the **whitespace** in both strings is **normalized**. **Whitespace normalization strips leading and trailing whitespace characters**, and collapses sequences of one or more whitespace characters to a single space character. This corresponds to the behaviour of an XML parser for an attribute that is declared as other than CDATA. Thus the above pattern will match any of:

```
<card name="John Smith" email="js@example.com" preferredFormat="html"/>
<card name="John Smith" email="js@example.com" preferredFormat="  html  "/>
```

The way that the **value** pattern compares the pattern string with the document string **can be controlled by specifying a type attribute and optionally a datatypeLibrary attribute**, which identify a **datatype** in the same way as for the **data** pattern.

The pattern **string** matches the document **string** if they both represent the same value of the specified **datatype**. Thus, whereas the **data** pattern matches an arbitrary **value** of a datatype, the **value** pattern matches a specific **value** of a datatype.

If there is no ancestor element with a **datatypeLibrary** element, the datatype library defaults to a built-in RELAX NG datatype library. This provides two datatypes, string and token. The built-in datatype token corresponds to the default comparison behavior of the value pattern. The built-in datatype string compares strings **without any whitespace normalization** (other than the end-of-line and attribute value normalization automatically performed by XML). For example,

```
<element name="card">
  <attribute name="name"/>
  <attribute name="email"/>
  <attribute name="preferredFormat">
    <choice>
      <value type="string">html</value>
      <value type="string">text</value>
    </choice>
  </attribute>
</element>
```

will not match

```
<card name="John Smith" email="js@example.com" preferredFormat="  html  "/>
```

Lists

The **list** pattern matches a **whitespace-separated sequence of tokens**; it contains a pattern that the sequence of individual tokens must match. The list pattern splits a string into a list of strings, and then matches the resulting list of strings against the pattern inside the list pattern.

For example, suppose we want to have a vector element that contains two floating point numbers separated by whitespace. We could use list as follows:

```
<element name="vector">
  <list>
    <data type="float"/>
    <data type="float"/>
  </list>
</element>
```

Or suppose we want the vector element to contain a **list** of one or more floating point numbers separated by whitespace:

```
<element name="vector">
  <list>
    <oneOrMore>
      <data type="double"/>
    </oneOrMore>
  </list>
</element>
```

Or suppose we want a path element containing an **even** number of floating point numbers:

```
<element name="path">
  <list>
    <oneOrMore>
      <data type="double"/>
      <data type="double"/>
    </oneOrMore>
  </list>
</element>
```

Interleaving

The **interleave** pattern allows child elements to **occur in any order**. For example, the following would allow the card element to contain the name and email elements in any order:

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <interleave>
        <element name="name">
          <text/>
        </element>
        <element name="email">
          <text/>
        </element>
      </interleave>
    </element>
 </zeroOrMore>
</element>
```

The pattern is called **interleave** because of how it works with patterns that match more than one element.

Suppose we want to write a pattern for the HTML head element which requires exactly one **title** element, at most one **base** element and zero or more **style**, **script**, **link** and **meta** elements and suppose we are writing a **grammar** pattern that has one definition for each element. Then we could define the pattern for head as follows:

```
<define name="head">
  <element name="head">
    <interleave>
      <ref name="title"/>
      <optional>
        <ref name="base"/>
      </optional>
      <zeroOrMore>
        <ref name="style"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="script"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="link"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="meta"/>
      </zeroOrMore>
    </interleave>
  </element>
</define>
```

Suppose we had a **head** element that contained a **meta** element, followed by a **title** element, followed by a **meta** element. This would match the pattern because it is an interleaving of a sequence of two **meta** elements, which match the child pattern

```
   <zeroOrMore>
     <ref name="meta"/>
   </zeroOrMore>
```

and a sequence of one title element, which matches the child pattern

```
   <ref name=„title"/>
```

The semantics of the **interleave** pattern are that a sequence of elements matches an **interleave** pattern if it is an interleaving of sequences that match the child patterns of the **interleave** pattern. Note that this is different from the & connector in SGML: A* & B matches the sequence of elements A A B or the sequence of elements B A A but not the sequence of elements A B A.

One special case of **interleave** is very common: interleaving *<text/>* with a pattern **p** represents a pattern that matches what **p** matches but also **allows characters to occur as children**. The mixed element is a shorthand for this.

```
<mixed> p </mixed>
```

is short for

```
<interleave> <text/> p </interleave>
```

Modularity

Referencing external patterns

The *externalRef* pattern can be used to reference a pattern defined in a separate file. The *externalRef* element has a required *href* attribute that specifies the URL of a file containing the pattern. The externalRef matches if the pattern contained in the specified URL matches.
Suppose for example, you have a RELAX NG pattern that matches HTML inline content stored in *inline.rng*:

```
<grammar>
  <start>
    <ref name="inline"/>
  </start>

  <define name="inline">
    <zeroOrMore>
      <choice>
        <text/>
        <element name="code">
          <ref name="inline"/>
        </element>
        <element name="em">
          <ref name="inline"/>
        </element>
        <!-- etc -->
      </choice>
    </zeroOrMore>
  </define>
</grammar>
```

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
      <optional>
     <element name="note">
       <externalRef href="inline.rng"/>
     </element>
      </optional>
    </element>
  </zeroOrMore>
</element>
```

For another example, suppose you have two RELAX NG patterns stored in files *pattern1.rng* and *pattern2.rng*. Then the following is a pattern that matches anything matched by either of those patterns:

```
<choice>
  <externalRef href="pattern1.rng"/>
  <externalRef href="pattern2.rng"/>
</choice>
```

Modularity

Combining definitions

If a grammar contains multiple definitions with the same name, then the definitions must specify how they are to be combined into a single definition by using the combine attribute. The combine attribute may have the value choice or interleave. For example:

```
<define name="inline.class"
combine="choice">
  <element name="bold">
    <ref name="inline"/>
  </element>
</define>

<define name="inline.class"
combine="choice">
  <element name="italic">
    <ref name="inline"/>
  </element>
</define>
```

is equivalent to

```
<define name="inline.class">
  <choice>
    <element name="bold">
      <ref name="inline"/>
    </element>
    <element name="italic">
      <ref name="inline"/>
    </element>
  </choice>
</define>
```

When combining attributes, *combine="interleave"* is typically used. For example,

```
<grammar>
  <start>
    <element name="addressBook">
      <zeroOrMore>
     <element name="card">
       <ref name="card.attlist"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name=„card.attlist" combine="interleave">
    <attribute name="name">
      <text/>
    </attribute>
  </define>

  <define name="card.attlist" combine="interleave">
    <attribute name="email">
      <text/>
    </attribute>
  </define>
</grammar>
```

is equivalent to

```
<grammar>

  <start>
    <element name="addressBook">
      <zeroOrMore>
     <element name="card">
       <ref name="card.attlist"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="card.attlist">
    <interleave>
      <attribute name="name">
     <text/>
      </attribute>
      <attribute name="email">
     <text/>
      </attribute>
    </interleave>
  </define>

</grammar>
```

which is equivalent to

```
<grammar>

  <start>
    <element name="addressBook">
      <zeroOrMore>
     <element name="card">
       <ref name="card.attlist"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="card.attlist">
    <group>
      <attribute name="name">
     <text/>
      </attribute>
      <attribute name="email">
     <text/>
      </attribute>
    </group>
  </define>

</grammar>
```

since combining attributes with *interleave* has the same effect as combining them with *group*.

It is an error for two definitions of the same name to specify different values for *combine*. Note that the order of definitions within a grammar is not significant.

Multiple *start* elements can be combined in the same way as multiple definitions.

Modularity

Merging grammars

The *include* element allows grammars to be merged together. A *grammar* pattern may have *include* elements as children. An *include* element has a required *href* attribute that specifies the URL of a file containing a grammar pattern. The definitions in the referenced grammar pattern will be included in grammar pattern containing the *include* element.

The *combine* attribute is particularly useful in conjunction with *include*. For example, suppose a RELAX NG pattern *inline.rng* provides a pattern for *inline* content, which allows *bold* and *italic* elements arbitrarily nested:

```
<grammar>
  <define name="inline">
    <zeroOrMore>
      <ref name="inline.class"/>
    </zeroOrMore>
  </define>

  <define name="inline.class">
    <choice>
      <text/>
      <element name="bold">
     <ref name="inline"/>
      </element>
      <element name="italic">
     <ref name="inline"/>
      </element>
    </choice>
  </define>
</grammar>
```

Another RELAX NG pattern could use inline.rng and add code and em to the set of inline elements as follows:

```
<grammar>
  <include href="inline.rng"/>

  <start>
    <element name="doc">
      <zeroOrMore>
     <element name="p">
       <ref name="inline"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="inline.class" combine="choice">
    <choice>
      <element name="code">
     <ref name="inline">
      </element>
      <element name="em">
     <ref name="inline">
      </element>
    </choice>
  </define>
</grammar>
```

This would be equivalent to

```
<grammar>
  <define name="inline">
    <zeroOrMore>
      <ref name="inline.class"/>
    </zeroOrMore>
  </define>

  <define name="inline.class">
    <choice>
      <text/>
      <element name="bold">
    <ref name="inline"/>
      </element>
      <element name="italic">
    <ref name="inline"/>
      </element>
    </choice>
  </define>


  ….
```

This would be equivalent to

```
<grammar>
  ….

  <start>
    <element name="doc">
      <zeroOrMore>
     <element name="p">
       <ref name="inline"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="inline.class" combine="choice">
    <choice>
      <element name="code">
     <ref name="inline">
      </element>
      <element name="em">
     <ref name="inline">
      </element>
    </choice>
  </define>

</grammar>
```

which is equivalent to

```
<grammar>
  <define name="inline">
    <zeroOrMore>
      <ref name="inline.class"/>
    </zeroOrMore>
  </define>

  <define name="inline.class">
    <choice>
      <text/>
      <element name="bold">
     <ref name="inline"/>
      </element>
      <element name="italic">
     <ref name="inline"/>
      </element>
      <element name="code">
     <ref name="inline">
      </element>
      <element name="em">
     <ref name="inline">
      </element>
    </choice>
  </define>
  …
```

which is equivalent to

```
<grammar>
  …
  <start>
    <element name="doc">
      <zeroOrMore>
   <element name="p">
     <ref name="inline"/>
   </element>
      </zeroOrMore>
    </element>
  </start>

</grammar>
```

Note that it is allowed for one of the definitions of a name to omit the combine attribute. However, it is an error if there is more than one definition that does so.

The *notAllowed* pattern is useful when merging grammars. The *notAllowed* pattern never matches anything. Just as adding *empty* to a *group* makes no difference, so adding *notAllowed* to a *choice* makes no difference. It is typically used to allow an including pattern to specify additional choices with *combine="choice"*. For example, if *inline.rng* were written like

```
<grammar>

  <define name="inline">
    <zeroOrMore>
      <choice>
    <text/>
    <element name="bold">
      <ref name="inline"/>
    </element>
    <element name="italic">
      <ref name="inline"/>
    </element>
    <ref name="inline.extra"/>
      </choice>
    </zeroOrMore>
  </define>

  <define name="inline.extra">
    <notAllowed/>
  </define>

</grammar>
```

then it could be customized to allow inline code and em elements as follows:

```
<grammar>
  <include href="inline.rng"/>

  <start>
    <element name="doc">
      <zeroOrMore>
     <element name="p">
       <ref name="inline"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="inline.extra" combine="choice">
    <choice>
      <element name="code">
     <ref name="inline">
      </element>
      <element name="em">
     <ref name="inline">
      </element>
    </choice>
  </define>

</grammar>
```

Modularity

Replacing definitions

RELAX NG allows define elements to be put inside the include element to indicate that they are to replace definitions in the included grammar pattern. Suppose the file addressBook.rng contains:

```
<grammar>
  <start>
    <element name="addressBook">
      <zeroOrMore>
     <element name="card">
       <ref name="cardContent"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="cardContent">
    <element name="name">
      <text/>
    </element>
    <element name="email">
      <text/>
    </element>
  </define>
</grammar>
```

Suppose we wish to modify this pattern so that the card element contains an emailAddress element instead of an email element. Then we could replace the definition of cardContent as follows:

```
<grammar>

  <include href="addressBook.rng">

    <define name="cardContent">
      <element name="name">
    <text/>
      </element>
      <element name="emailAddress">
    <text/>
      </element>
    </define>

  </include>

</grammar>
```

This would be equivalent to

```
<grammar>

  <start>
    <element name="addressBook">
      <zeroOrMore>
     <element name="card">
       <ref name="cardContent"/>
     </element>
      </zeroOrMore>
    </element>
  </start>

  <define name="cardContent">
    <element name="name">
      <text/>
    </element>
    <element name="emailAddress">
      <text/>
    </element>
  </define>

</grammar>
```

An include element can also contain a start element, which replaces the start in the included grammar pattern.

Namespaces

RELAX NG is namespace-aware. Thus, it considers an element or attribute to have both a local name and a namespace URI which together constitute the name of that element or attribute.

Using the *ns* attribute

The element pattern uses an ns attribute to specify the namespace URI of the elements that it matches. For example,

```
<element name="foo" ns="http://www.example.com">
  <empty/>
</element>
```

would match any of:

```
<foo xmlns="http://www.example.com"/>
<e:foo xmlns:e="http://www.example.com"/>
<example:foo xmlns:example="http://www.example.com"/>
```

but not any of:

```
<foo/>

<e:foo xmlns:e="http://WWW.EXAMPLE.COM"/>

<example:foo xmlns:example="http://www.example.net"/>
```

A value of an empty string for the ns attribute indicates a null or absent namespace URI (just as with the xmlns attribute). Thus, the pattern

```
<element name="foo" ns="">
  <empty/>
</element>
```

matches any of:

```
<foo xmlns=""/>
<foo/>
```

but not any of:

```
<foo xmlns="http://www.example.com"/>
<e:foo xmlns:e="http://www.example.com"/>
```

It is tedious and error-prone to specify the ns attribute on every element, so RELAX NG allows it to be defaulted. If an element pattern does not specify an *ns* attribute, then it defaults to the value of the *ns* attribute of the nearest ancestor that has an *ns* attribute, or the empty string if there is no such ancestor. Thus,

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

is equivalent to

```
<element name="addressBook" ns="">
  <zeroOrMore>
    <element name="card" ns="">
      <element name="name" ns="">
        <text/>
      </element>
      <element name="email" ns="">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

next example:

```
<element name="addressBook" ns="http://www.example.com">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

is equivalent to

```
<element name="addressBook" ns="http://www.example.com">
  <zeroOrMore>
    <element name="card" ns="http://www.example.com">
      <element name="name" ns="http://www.example.com">
        <text/>
      </element>
      <element name="email" ns="http://www.example.com">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

The attribute pattern also takes an ns attribute. However, there is a difference in how it defaults. This is because of the fact that the XML Namespaces Recommendation does not apply the default namespace to attributes. If an ns attribute is not specified on the attribute pattern, then it defaults to the empty string. Thus,

```
<element name="addressBook" ns="http://www.example.com">
  <zeroOrMore>
    <element name="card">
      <attribute name="name"/>
      <attribute name="email"/>
    </element>
  </zeroOrMore>
</element>
```

is equivalent to

```
<element name="addressBook" ns="http://www.example.com">
  <zeroOrMore>
    <element name="card" ns="http://www.example.com">
      <attribute name="name" ns=""/>
      <attribute name="email" ns=""/>
    </element>
  </zeroOrMore>
</element>
```

and so will match

```
<addressBook xmlns="http://www.example.com">
  <card name="John Smith" email="js@example.com"/>
</addressBook>
```

or

```
<example:addressBook xmlns:example="http://www.example.com">
  <example:card name="John Smith" email="js@example.com"/>
</example:addressBook>
```

but not

```
<example:addressBook xmlns:example="http://www.example.com">
  <example:card example:name="John Smith" example:email="js@example.com"/>
</example:addressBook>
```

Qualified names

When a pattern matches elements and attributes from multiple namespaces, using the ns attribute would require repeating namespace URIs in different places in the pattern. This is error-prone and hard to maintain, so RELAX NG also allows the element and attribute patterns to use a prefix in the value of the name attribute to specify the namespace URI. In this case, the prefix specifies the namespace URI to which that prefix is bound by the namespace declarations in scope on the element or attribute pattern. Thus,

```
<element name="ab:addressBook" xmlns:ab="http://www.example.com/addressBook"
                               xmlns:a="http://www.example.com/address">
  <zeroOrMore>
    <element name="ab:card">
      <element name="a:name">
        <text/>
      </element>
      <element name="a:email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

is equivalent to

```
<element name="addressBook" ns="http://www.example.com/addressBook">
  <zeroOrMore>
    <element name="card" ns="http://www.example.com/addressBook">
      <element name="name" ns="http://www.example.com/address">
        <text/>
      </element>
      <element name="email" ns="http://www.example.com/address">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

If a prefix is specified in the value of the name attribute of an element or attribute pattern, then that prefix determines the namespace URI of the elements or attributes that will be matched by that pattern, regardless of the value of any ns attribute.

Note that the XML default namespace (as specified by the xmlns attribute) is not used in determining the namespace URI of elements and attributes that element and attribute patterns match.

Name classes

Normally, the name of the element to be matched by an element element is specified by a name attribute. An element element can instead start with an element specifying a name-class. In this case, the element pattern will only match an element if the name of the element is a member of the name-class. The simplest name-class is anyName, which any name at all is a member of, regardless of its local name and its namespace URI. For example, the following pattern matches any well-formed XML document:

```
<grammar>
 <start>
  <ref name="anyElement"/>
 </start>

 <define name="anyElement">
  <element>
   <anyName/>
   <zeroOrMore>
        <choice>
         <attribute>
          <anyName/>
         </attribute>
         <text/>
         <ref name="anyElement"/>
        </choice>
   </zeroOrMore>
  </element>
 </define>

</grammar>
```

The nsName name-class contains any name with the namespace URI specified by the ns attribute, which defaults in the same way as the ns attribute on the element pattern.

The choice name-class matches any name that is a member of any of its child name-classes.

The anyName and nsName name-classes can contain an except clause. For example,

```
<element name="card" ns="http://www.example.com">
  <zeroOrMore>
    <attribute>
      <anyName>
        <except>
          <nsName/>
          <nsName ns=""/>
        </except>
      </anyName>
    </attribute>
  </zeroOrMore>
  <text/>
</element>
```

would allow the card element to have any number of namespace-qualified attributes provided that they were qualified with namespace other than that of the card element.

Note that an attribute pattern matches a single attribute even if it has a name-class that contains multiple names. To match zero or more attributes, the zeroOrMore element must be used.

The name name-class contains a single name. The content of the name element specifies the name in the same way as the name attribute of the element pattern. The ns attribute specifies the namespace URI in the same way as the element pattern.

Some schema languages have a concept of lax validation, where an element or attribute is validated against a definition only if there is one. We can implement this concept in RELAX NG with name classes that uses except and name. Suppose, for example, we wanted to allow an element to have any attribute with a qualified name, but we still wanted to ensure that if there was an xml:space attribute, it had the value default or preserve. It wouldn't work to use

```
<element name="example">
  <zeroOrMore>
    <attribute>
      <anyName/>
    </attribute>
  </zeroOrMore>
  <optional>
    <attribute name="xml:space">
      <choice>
        <value>default</value>
        <value>preserve</value>
      </choice>
    </attribute>
  </optional>
</element>
```

because an xml:space attribute with a value other than default or preserve would match

```
<attribute>
  <anyName/>
</attribute>
```

even though it did not match

```
<attribute name="xml:space">
  <choice>
    <value>default</value>
    <value>preserve</value>
  </choice>
</attribute>
```

The solution is to use name together with except:

```
<element name="example">
  <zeroOrMore>
    <attribute>
      <anyName>
        <except>
          <name>xml:space</name>
        </except>
      </anyName>
    </attribute>
  </zeroOrMore>
  <optional>
    <attribute name="xml:space">
      <choice>
        <value>default</value>
        <value>preserve</value>
      </choice>
    </attribute>
  </optional>
</element>
```

Note that the define element cannot contain a name-class; it can only contain a pattern.

Annotations

If a RELAX NG element has an attribute or child element with a namespace URI other than the RELAX NG namespace, then that attribute or element is ignored. Thus, you can add annotations to RELAX NG patterns simply by using an attribute or element in a separate namespace:

```
<element name="addressBook" xmlns="http://relaxng.org/ns/structure/1.0" xmlns:a="http://www.example.com/annotation">
  <zeroOrMore>
    <element name="card">
      <a:documentation>Information about a single email address.</a:documentation>
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

RELAX NG also provides a div element which allows an annotation to be applied to a group of definitions in a grammar. For example, you might want to divide up the definitions of the grammar into modules:

```
<grammar xmlns:m="http://www.example.com/module">

  <div m:name="inline">

    <define name="code"> pattern </define>
    <define name="em"> pattern </define>
    <define name="var"> pattern </define>

  </div>

  <div m:name="block">

    <define name="p"> pattern </define>
    <define name="ul"> pattern </define>
    <define name="ol"> pattern </define>

  </div>
</grammar>
```

This would allow you easily to generate variants of the grammar based on a selection of modules.

A companion specification, RELAX NG DTD Compatibility [Compatibility], defines annotations to implement some features of XML DTDs.

Nested grammars

There is no prohibition against nesting grammar patterns. A ref pattern refers to a definition from nearest grammar ancestor. There is also a parentRef element that escapes out of the current grammar and references a definition from the parent of the current grammar.

Imagine the problem of writing a pattern for tables. The pattern for tables only cares about the structure of tables; it doesn't care about what goes inside a table cell. First, we create a RELAX NG pattern table.rng as follows:

```
<grammar>

<define name="cell.content">
  <notAllowed/>
</define>

<start>
  <element name="table">
    <oneOrMore>
      <element name="tr">
        <oneOrMore>
         <element name="td">
           <ref name="cell.content"/>
         </element>
        </oneOrMore>
      </element>
    </oneOrMore>
  </element>
</start>

</grammar>
```

Patterns that include table.rng must redefine cell.content. By using a nested grammar pattern containing a parentRef pattern, the including pattern can redefine cell.content to be a pattern defined in the including pattern's grammar, thus effectively importing a pattern from the parent grammar into the child grammar:

```
<grammar>
<start>
  <element name="doc">
    <zeroOrMore>
      <choice>
     <element name="p">
       <ref name="inline"/>
     </element>
     <grammar>
       <include href="table.rng">
         <define name="cell.content">
           <parentRef name="inline"/>
         </define>
          </include>
     </grammar>
      </choice>
    </zeroOrMore>
  </element>
</start>

<define name="inline">
  <zeroOrMore>
    <choice>
      <text/>
      <element name="em">
        <ref name="inline"/>
      </element>
    </choice>
  </zeroOrMore>
```