RELAX NG Specification

Committee Specification 3 December 2001

This version:
Committee Specification: 3 December 2001

Previous versions:
Committee Specification: 11 August 2001

Editors:
James Clark <jjc@jclark.com>, MURATA Makoto <EB2M-MRT@asahi-net.or.jp>

Abstract

This is the definitive specification of RELAX NG, a simple schema language for XML, based on [RELAX] and [TREX]. A RELAX NG schema specifies a pattern for the structure and content of an XML document. A RELAX NG schema is itself an XML document.

Status of this Document

This Committee Specification was approved for publication by the OASIS RELAX NG technical committee. It is a stable document which represents the consensus of the committee. Comments on this document may be sent to relax-ng-comment@lists.oasis-open.org.

A list of known errors in this document is available at http://www.oasis-open.org/committees/relax-ng/spec-20011203-errata.html.

Table of Contents

Appendixes

---

1. Introduction

This document specifies

- when an XML document is a correct RELAX NG schema
- when an XML document is valid with respect to a correct RELAX NG schema

An XML document that is being validated with respect to a RELAX NG schema is referred to as an instance.

The structure of this document is as follows. Section 2 describes the data model, which is the abstraction of an XML document used throughout the rest of the document. Section 3 describes the syntax of a RELAX NG schema; any correct RELAX NG schema must conform to this syntax. Section 4 describes a sequence of transformations that are applied to simplify a RELAX NG schema; applying the transformations also involves checking certain restrictions that must be satisfied by a correct RELAX NG schema. Section 5 describes the syntax that results from applying the transformations; this simple syntax is a subset of the full syntax. Section 6 describes the semantics of a correct RELAX NG schema that uses the simple syntax; the semantics specify when an element is valid with respect to a RELAX NG schema. Section 7 describes restrictions in terms of the simple syntax; a correct RELAX NG schema must be such that, after transformation into the simple form, it satisfies these restrictions. Finally, Section 8 describes conformance requirements for RELAX NG validators.

A tutorial is available separately (see [Tutorial]).

2. Data model

RELAX NG deals with XML documents representing both schemas and instances through an abstract data model. XML documents representing schemas and instances must be well-formed in conformance with [XML 1.0] and must conform to the constraints of [XML Namespaces].

An XML document is represented by an element. An element consists of

- a name
- a context
- a set of attributes
- an ordered sequence of zero or more children; each child is either an element or a non-empty string; the sequence never contains two consecutive strings

A name consists of

- a string representing the namespace URI; the empty string has special significance, representing the absence of any namespace
- a string representing the local name; this string matches the NCName production of [XML Namespaces]

A context consists of

- a base URI
- a namespace map; this maps prefixes to namespace URIs, and also may specify a default namespace URI (as declared by the `xmlns` attribute)

An attribute consists of

- a name
- a string representing the value

A string consists of a sequence of zero or more characters, where a character is as defined in [XML 1.0].

The element for an XML document is constructed from an instance of the [XML Infoset] as follows. We use the notation [*x*] to refer to the value of the *x* property of an information item. An element is constructed from a document information item by constructing an element from the [document element]. An element is constructed from an element information item by constructing the name from the [namespace name] and [local name], the context from the [base URI] and [in-scope namespaces], the attributes from the [attributes], and the children from the [children]. The attributes of an element are constructed from the unordered set of attribute information items by constructing an attribute for each attribute information item. The children of an element are constructed from the list of child information items first by removing information items other than element information items and character information items, and then by constructing an element for each element information item in the list and a string for each maximal sequence of character information items. An attribute is constructed from an attribute information item by constructing the name from the [namespace name] and [local name], and the value from the [normalized value]. When constructing the name of an element or attribute from the [namespace name] and [local name], if the [namespace name] property is not present, then the name is constructed from an empty string and the [local name]. A string is constructed from a sequence of character information items by constructing a character from the [character code] of each character information item.

It is possible for there to be multiple distinct infosets for a single XML document. This is because XML parsers are not required to process all DTD declarations or expand all external parsed general entities. Amongst these multiple infosets, there is exactly one infoset for which [all declarations processed] is true and which does not contain any unexpanded entity reference information items. This is the infoset that is the basis for defining the RELAX NG data model.

2.1. Example

Suppose the document `http://www.example.com/doc.xml` is as follows:

```
<?xml version="1.0"?>
<foo><pre1:bar1 xmlns:pre1="http://www.example.com/n1"/><pre2:bar2
  xmlns:pre2="http://www.example.com/n2"/></foo>
```

The element representing this document has

- a name which has
  - the empty string as the namespace URI, representing the absence of any namespace
  - `foo` as the local name
- a context which has
  - `http://www.example.com/doc.xml` as the base URI
  - a namespace map which

        - maps the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace` (the `xml` prefix is implicitly declared by every XML document)
        - specifies the empty string as the default namespace URI
- an empty set of attributes
- a sequence of children consisting of an element which has
    - a name which has
        - `http://www.example.com/n1` as the namespace URI
        - `bar1` as the local name
    - a context which has
        - `http://www.example.com/doc.xml` as the base URI
        - a namespace map which
            - maps the prefix `pre1` to the namespace URI `http://www.example.com/n1`
            - maps the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`
            - specifies the empty string as the default namespace URI
    - an empty set of attributes
    - an empty sequence of children

  followed by an element which has

    - a name which has
        - `http://www.example.com/n2` as the namespace URI
        - `bar2` as the local name
    - a context which has
        - `http://www.example.com/doc.xml` as the base URI
        - a namespace map which
            - maps the prefix `pre2` to the namespace URI `http://www.example.com/n2`
            - maps the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`
            - specifies the empty string as the default namespace URI
    - an empty set of attributes
    - an empty sequence of children

3. Full syntax

The following grammar summarizes the syntax of RELAX NG. Although we use a notation based on the XML representation of an RELAX NG schema as a sequence of characters, the grammar must be understood as operating at the data model level. For example, although the syntax uses `<text/>`, an instance or schema can use `<text></text>` instead, because they both represent the same element at the data model level. All elements shown in the grammar are qualified with the namespace URI:

`http://relaxng.org/ns/structure/1.0`

The symbols QName and NCName are defined in [XML Namespaces]. The anyURI symbol has the same meaning as the anyURI datatype of [W3C XML Schema Datatypes]: it indicates a string that, after escaping of disallowed values as described in Section 5.4 of [XLink], is a URI reference as defined in [RFC 2396] (as modified by [RFC 2732]). The symbol string matches any string.

In addition to the attributes shown explicitly, any element can have an `ns` attribute and any element can have a `datatypeLibrary` attribute. The `ns` attribute can have any value. The value of the `datatypeLibrary` attribute must match the anyURI symbol as described in the previous paragraph; in addition, it must not use the relative form of URI reference and must not have a fragment identifier; as an exception to this, the value may be the empty string.

Any element can also have foreign attributes in addition to the attributes shown in the grammar. A foreign attribute is an attribute with a name whose namespace URI is neither the empty string nor the RELAX NG namespace URI. Any element that cannot have string children (that is, any element other than `value`, `param` and `name`) may have foreign child elements in addition to the child elements shown in the grammar. A foreign element is an element with a name whose namespace URI is not the RELAX NG namespace URI. There are no constraints on the relative position of foreign child elements with respect to other child elements.

Any element can also have as children strings that consist entirely of whitespace characters, where a whitespace character is one of #x20, #x9, #xD or #xA. There are no constraints on the relative position of whitespace string children with respect to child elements.

Leading and trailing whitespace is allowed for value of each `name`, `type` and `combine` attribute and for the content of each `name` element.

```
pattern ::= <element name="QName"> pattern+ </element>
          | <element> nameClass pattern+ </element>
          | <attribute name="QName"> [pattern] </attribute>
          | <attribute> nameClass [pattern] </attribute>
          | <group> pattern+ </group>
          | <interleave> pattern+ </interleave>
          | <choice> pattern+ </choice>
          | <optional> pattern+ </optional>
          | <zeroOrMore> pattern+ </zeroOrMore>
          | <oneOrMore> pattern+ </oneOrMore>
          | <list> pattern+ </list>
          | <mixed> pattern+ </mixed>
          | <ref name="NCName"/>
          | <parentRef name="NCName"/>
          | <empty/>
          | <text/>
          | <value [type="NCName"]> string </value>
          | <data type="NCName"> param* [exceptPattern] </data>
          | <notAllowed/>
          | <externalRef href="anyURI"/>
          | <grammar> grammarContent* </grammar>
param   ::= <param name="NCName"> string </param>
```

exceptPattern	::=	`<except>` *pattern*+ `</except>`
grammarContent	::=	*start* | *define* | `<div>` *grammarContent** `</div>` | `<include href="`*anyURI*`">` *includeContent** `</include>`
includeContent	::=	*start* | *define* | `<div>` *includeContent** `</div>`
start	::=	`<start [combine="`*method*`"]>` *pattern* `</start>`
define	::=	`<define name="`*NCName*`" [combine="`*method*`"]>` *pattern*+ `</define>`
method	::=	`choice` | `interleave`
nameClass	::=	`<name>` *QName* `</name>` | `<anyName>` [*exceptNameClass*] `</anyName>` | `<nsName>` [*exceptNameClass*] `</nsName>` | `<choice>` *nameClass*+ `</choice>`
exceptNameClass	::=	`<except>` *nameClass*+ `</except>`

3.1. Example

Here is an example of a schema in the full syntax for the document in Section 2.1.

```
<?xml version="1.0"?>
<element name="foo"
         xmlns="http://relaxng.org/ns/structure/1.0"
         xmlns:a="http://relaxng.org/ns/annotation/1.0"
         xmlns:ex1="http://www.example.com/n1"
         xmlns:ex2="http://www.example.com/n2">
  <a:documentation>A foo element.</a:document>
  <element name="ex1:bar1">
    <empty/>
  </element>
  <element name="ex2:bar2">
    <empty/>
  </element>
</element>
```

4. Simplification

The full syntax given in the previous section is transformed into a simpler syntax by applying the following transformation rules in order. The effect must be as if each rule was applied to all elements in the schema before the next rule is applied. A transformation rule may also specify constraints that must be satisfied by a correct schema. The transformation rules are applied at the data model level. Before the transformations are applied, the schema is parsed into an instance of the data model.

4.1. Annotations

Foreign attributes and elements are removed.

Note

It is safe to remove `xml:base` attributes at this stage because `xml:base` attributes are used in determining the [base URI] of an element information item, which is in turn used to construct the base URI of the context of an element. Thus, after a document has been parsed into an instance of the data model, `xml:base` attributes can be discarded.

4.2. Whitespace

For each element other than `value` and `param`, each child that is a string containing only whitespace characters is removed.

Leading and trailing whitespace characters are removed from the value of each `name`, `type` and `combine` attribute and from the content of each `name` element.

4.3. `datatypeLibrary` attribute

The value of each `datatypeLibary` attribute is transformed by escaping disallowed characters as specified in Section 5.4 of [XLink].

For any `data` or `value` element that does not have a `datatypeLibrary` attribute, a `datatypeLibrary` attribute is added. The value of the added `datatypeLibrary` attribute is the value of the `datatypeLibrary` attribute of the nearest ancestor element that has a `datatypeLibrary` attribute, or the empty string if there is no such ancestor. Then, any `datatypeLibrary` attribute that is on an element other than `data` or `value` is removed.

4.4. `type` attribute of `value` element

For any `value` element that does not have a `type` attribute, a `type` attribute is added with value `token` and the value of the `datatypeLibrary` attribute is changed to the empty string.

4.5. `href` attribute

The value of the `href` attribute on an `externalRef` or `include` element is first transformed by escaping disallowed characters as specified in Section 5.4 of [XLink]. The URI reference is then resolved into an absolute form as described in section 5.2 of [RFC 2396] using the base URI from the context of the element that bears the `href` attribute.

The value of the `href` attribute will be used to construct an element (as specified in Section 2). This must be done as follows. The URI reference consists of the URI itself and an optional fragment identifier. The resource identified by the URI is retrieved. The result is a MIME entity: a sequence of bytes labeled with a MIME media type. The media type determines how an element is constructed from the MIME entity and optional fragment identifier. When the media type is `application/xml` or `text/xml`, the MIME entity must be parsed as an XML document in accordance with the applicable RFC (at the term of writing [RFC 3023]) and an element constructed from the result of the parse as specified in Section 2. In particular, the `charset` parameter must be handled as specified by the RFC. This specification does not define

the handling of media types other than `application/xml` and `text/xml`. The `href` attribute must not include a fragment identifier unless the registration of the media type of the resource identified by the attribute defines the interpretation of fragment identifiers for that media type.

> **Note**
>
> [RFC 3023] does not define the interpretation of fragment identifiers for `application/xml` or `text/xml`.

4.6. `externalRef` element

An `externalRef` element is transformed as follows. An element is constructed using the URI reference that is the value of `href` attribute as specified in Section 4.5. This element must match the syntax for pattern. The element is transformed by recursively applying the rules from this subsection and from previous subsections of this section. This must not result in a loop. In other words, the transformation of the referenced element must not require the dereferencing of an `externalRef` attribute with an `href` attribute with the same value.

Any `ns` attribute on the `externalRef` element is transferred to the referenced element if the referenced element does not already have an `ns` attribute. The `externalRef` element is then replaced by the referenced element.

4.7. `include` element

An `include` element is transformed as follows. An element is constructed using the URI reference that is the value of `href` attribute as specified in Section 4.5. This element must be a `grammar` element, matching the syntax for grammar.

This `grammar` element is transformed by recursively applying the rules from this subsection and from previous subsections of this section. This must not result in a loop. In other words, the transformation of the `grammar` element must not require the dereferencing of an `include` attribute with an `href` attribute with the same value.

Define the *components* of an element to be the children of the element together with the components of any `div` child elements. If the `include` element has a `start` component, then the `grammar` element must have a `start` component. If the `include` element has a `start` component, then all `start` components are removed from the `grammar` element. If the `include` element has a `define` component, then the `grammar` element must have a `define` component with the same name. For every `define` component of the `include` element, all `define` components with the same name are removed from the `grammar` element.

The `include` element is transformed into a `div` element. The attributes of the `div` element are the attributes of the `include` element other than the `href` attribute. The children of the `div` element are the `grammar` element (after the removal of the `start` and `define` components described by the preceding paragraph) followed by the children of the `include` element. The `grammar` element is then renamed to `div`.

4.8. `name` attribute of `element` and `attribute` elements

The name attribute on an element or attribute element is transformed into a name child element.

If an attribute element has a name attribute but no ns attribute, then an ns="" attribute is added to the name child element.

4.9. ns attribute

For any name, nsName or value element that does not have an ns attribute, an ns attribute is added. The value of the added ns attribute is the value of the ns attribute of the nearest ancestor element that has an ns attribute, or the empty string if there is no such ancestor. Then, any ns attribute that is on an element other than name, nsName or value is removed.

Note

The value of the ns attribute is *not* transformed either by escaping disallowed characters, or in any other way, because the value of the ns attribute is compared against namespace URIs in the instance, which are not subject to any transformation.

Note

Since include and externalRef elements are resolved after datatypeLibrary attributes are added but before ns attributes are added, ns attributes are inherited into external schemas but datatypeLibrary attributes are not.

4.10. QNames

For any name element containing a prefix, the prefix is removed and an ns attribute is added replacing any existing ns attribute. The value of the added ns attribute is the value to which the namespace map of the context of the name element maps the prefix. The context must have a mapping for the prefix.

4.11. div element

Each div element is replaced by its children.

4.12. Number of child elements

A define, oneOrMore, zeroOrMore, optional, list or mixed element is transformed so that it has exactly one child element. If it has more than one child element, then its child elements are wrapped in a group element. Similarly, an element element is transformed so that it has exactly two child elements, the first being a name class and the second being a pattern. If it has more than two child elements, then the child elements other than the first are wrapped in a group element.

A except element is transformed so that it has exactly one child element. If it has more than one child element, then its child elements are wrapped in a choice element.

If an `attribute` element has only one child element (a name class), then a `text` element is added.

A `choice`, `group` or `interleave` element is transformed so that it has exactly two child elements. If it has one child element, then it is replaced by its child element. If it has more than two child elements, then the first two child elements are combined into a new element with the same name as the parent element and with the first two child elements as its children. For example,

```
<choice> p1 p2 p3 </choice>
```

is transformed to

```
<choice> <choice> p1 p2 </choice> p3 </choice>
```

This reduces the number of child elements by one. The transformation is applied repeatedly until there are exactly two child elements.

4.13. `mixed` element

A `mixed` element is transformed into an interleaving with a `text` element:

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

is transformed into

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

4.14. `optional` element

An `optional` element is transformed into a choice with `empty`:

```
<optional> p </optional>
```

is transformed into

```
<choice> p <empty/> </choice>
```

4.15. `zeroOrMore` element

A `zeroOrMore` element is transformed into a choice between `oneOrMore` and `empty`:

```
<zeroOrMore> p </zeroOrMore>
```

is transformed into

```
<choice> <oneOrMore> p </oneOrMore> <empty/> </choice>
```

4.16. Constraints

In this rule, no transformation is performed, but various constraints are checked.

Note

The constraints in this section, unlike the constraints specified in Section 7, can be checked without resolving any `ref` elements, and are accordingly applied even to patterns that will disappear during later stages of simplification because they are not reachable (see Section 4.19) or because of `notAllowed` (see Section 4.20).

An `except` element that is a child of an `anyName` element must not have any `anyName` descendant elements. An `except` element that is a child of an `nsName` element must not have any `nsName` or `anyName` descendant elements.

A `name` element that occurs as the first child of an `attribute` element or as the descendant of the first child of an `attribute` element and that has an `ns` attribute with value equal to the empty string must not have content equal to `xmlns`.

A `name` or `nsName` element that occurs as the first child of an `attribute` element or as the descendant of the first child of an `attribute` element must not have an `ns` attribute with value `http://www.w3.org/2000/xmlns`.

Note

The [XML Infoset] defines the namespace URI of namespace declaration attributes to be `http://www.w3.org/2000/xmlns`.

A `data` or `value` element must be correct in its use of datatypes. Specifically, the `type` attribute must identify a datatype within the datatype library identified by the value of the `datatypeLibrary` attribute. For a `data` element, the parameter list must be one that is allowed by the datatype (see Section 6.2.8).

4.17. `combine` attribute

For each `grammar` element, all `define` elements with the same name are combined together. For any name, there must not be more than one `define` element with that name that does not have a `combine` attribute. For any name, if there is a `define` element with that name that has a `combine` attribute with the value `choice`, then there must not also be a `define` element with that name that has a `combine` attribute with the value `interleave`. Thus, for any name, if there is more than one `define` element with that name, then there is a unique value for the `combine` attribute for that name. After determining this unique value, the `combine` attributes are removed. A pair of definitions

```
<define name="n">
  p1
</define>
<define name="n">
  p2
</define>
```

is combined into

```
<define name="n">
  <c>
    p1
    p2
  </c>
</define>
```

where *c* is the value of the `combine` attribute. Pairs of definitions are combined until there is exactly one `define` element for each name.

Similarly, for each `grammar` element all `start` elements are combined together. There must not be more than one `start` element that does not have a `combine` attribute. If there is a `start` element that has a `combine` attribute with the value `choice`, there must not also be a `start` element that has a `combine` attribute with the value `interleave`.

4.18. `grammar` element

In this rule, the schema is transformed so that its top-level element is `grammar` and so that it has no other `grammar` elements.

Define the *in-scope grammar* for an element to be the nearest ancestor `grammar` element. A `ref` element *refers to* a `define` element if the value of their `name` attributes is the same and their in-scope grammars are the same. A `parentRef` element *refers to* a `define` element if the value of their `name` attributes is the same and the in-scope grammar of the in-scope grammar of the `parentRef` element is the same as the in-scope grammar of the `define` element. Every `ref` or `parentRef` element must refer to a `define` element. A `grammar` must have a `start` child element.

First, transform the top-level pattern *p* into `<grammar><start>`*p*`</start></grammar>`. Next, rename `define` elements so that no two `define` elements anywhere in the schema have the same name. To rename a `define` element, change the value of its `name` attribute and change the value of the `name` attribute of all `ref` and `parentRef` elements that refer to that `define` element. Next, move all `define` elements to be children of the top-level `grammar` element, replace each nested `grammar` element by the child of its `start` element and rename each `parentRef` element to `ref`.

4.19. `define` and `ref` elements

In this rule, the grammar is transformed so that every `element` element is the child of a `define` element, and the child of every `define` element is an `element` element.

First, remove any `define` element that is not *reachable*. A `define` element is reachable if there is reachable `ref` element referring to it. A `ref` element is reachable if it is the descendant of the `start` element or of a reachable `define` element. Now, for each `element` element that is not the child of a `define` element, add a `define` element to the `grammar` element, and replace the `element` element by a `ref` element referring to the added `define` element. The value of the `name` attribute of the added `define` element must be different from value of the `name` attribute of all other `define` elements. The child of the added `define` element is the `element` element.

Define a `ref` element to be *expandable* if it refers to a `define` element whose child is not an `element` element. For each `ref` element that is expandable and is a descendant of a `start` element or an `element` element, expand it by replacing the `ref` element by the child of the

`define` element to which it refers and then recursively expanding any expandable `ref` elements in this replacement. This must not result in a loop. In other words expanding the replacement of a `ref` element having a `name` with value *n* must not require the expansion of `ref` element also having a `name` with value *n*. Finally, remove any `define` element whose child is not an `element` element.

4.20. `notAllowed` element

In this rule, the grammar is transformed so that a `notAllowed` element occurs only as the child of a `start` or `element` element. An `attribute`, `list`, `group`, `interleave`, or `oneOrMore` element that has a `notAllowed` child element is transformed into a `notAllowed` element. A `choice` element that has two `notAllowed` child elements is transformed into a `notAllowed` element. A `choice` element that has one `notAllowed` child element is transformed into its other child element. An `except` element that has a `notAllowed` child element is removed. The preceding transformations are applied repeatedly until none of them is applicable any more. Any `define` element that is no longer reachable is removed.

4.21. `empty` element

In this rule, the grammar is transformed so that an `empty` element does not occur as a child of a `group`, `interleave`, or `oneOrMore` element or as the second child of a `choice` element. A `group`, `interleave` or `choice` element that has two `empty` child elements is transformed into an `empty` element. A `group` or `interleave` element that has one `empty` child element is transformed into its other child element. A `choice` element whose second child element is an `empty` element is transformed by interchanging its two child elements. A `oneOrMore` element that has an `empty` child element is transformed into an `empty` element. The preceding transformations are applied repeatedly until none of them is applicable any more.

5. Simple syntax

After applying all the rules in Section 4, the schema will match the following grammar:

grammar	::=	`<grammar> <start>` *top* `</start>` *define** `</grammar>`
define	::=	`<define name="`*NCName*`"> <element>` *nameClass top* `</element> </define>`
top	::=	`<notAllowed/>` | *pattern*
pattern	::=	`<empty/>` | *nonEmptyPattern*
nonEmptyPattern	::=	`<text/>` | `<data type="`*NCName*`" datatypeLibrary="`*anyURI*`">` *param** [*exceptPattern*] `</data>` | `<value datatypeLibrary="`*anyURI*`" type="`*NCName*`" ns="`*string*`">` *string* `</value>` | `<list>` *pattern* `</list>` | `<attribute>` *nameClass pattern* `</attribute>` | `<ref name="`*NCName*`"/>`

		| `<oneOrMore>` *nonEmptyPattern* `</oneOrMore>`
		| `<choice>` *pattern nonEmptyPattern* `</choice>`
		| `<group>` *nonEmptyPattern nonEmptyPattern* `</group>`
		| `<interleave>` *nonEmptyPattern nonEmptyPattern* `</interleave>`
param	::=	`<param name="`*NCName*`">` *string* `</param>`
exceptPattern	::=	`<except>` *pattern* `</except>`
nameClass	::=	`<anyName>` [*exceptNameClass*] `</anyName>`
		| `<nsName ns="`*string*`">` [*exceptNameClass*] `</nsName>`
		| `<name ns="`*string*`">` *NCName* `</name>`
		| `<choice>` *nameClass nameClass* `</choice>`
exceptNameClass	::=	`<except>` *nameClass* `</except>`

With this grammar, no elements or attributes are allowed other than those explicitly shown.

5.1. Example

The following is an example of how the schema in Section 3.1 can be transformed into the simple syntax:

```
<?xml version="1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="foo.element"/>
  </start>

  <define name="foo.element">
    <element>
      <name ns="">foo</name>
      <group>
        <ref name="bar1.element"/>
        <ref name="bar2.element"/>
      </group>
    </element>
  </define>

  <define name="bar1.element">
    <element>
      <name ns="http://www.example.com/n1">bar1</name>
      <empty/>
    </element>
  </define>

  <define name="bar2.element">
    <element>
      <name ns="http://www.example.com/n2">bar2</name>
      <empty/>
    </element>
  </define>
</grammar>
```

Note

Strictly speaking, the result of simplification is an instance of the data model rather than an XML document. For convenience, we use an XML document to represent an instance of the data model.

6. Semantics

In this section, we define the semantics of a correct RELAX NG schema that has been transformed into the simple syntax. The semantics of a RELAX NG schema consist of a specification of what XML documents are valid with respect to that schema. The semantics are described formally. The formalism uses axioms and inference rules. Axioms are propositions that are provable unconditionally. An inference rule consists of one or more antecedents and exactly one consequent. An antecedent is either positive or negative. If all the positive antecedents of an inference rule are provable and none of the negative antecedents are provable, then the consequent of the inference rule is provable. An XML document is valid with respect to a RELAX NG schema if and only if the proposition that it is valid is provable in the formalism specified in this section.

Note

This kind of formalism is similar to a proof system. However, a traditional proof system only has positive antecedents.

The notation for inference rules separates the antecedents from the consequent by a horizontal line: the antecedents are above the line; the consequent is below the line. If an antecedent is of the form not(p), then it is a negative antecedent; otherwise, it is a positive antecedent. Both axioms and inferences rules may use variables. A variable has a name and optionally a subscript. The name of a variable is italicized. Each variable has a range that is determined by its name. Axioms and inference rules are implicitly universally quantified over the variables they contain. We explain this further below.

The possibility that an inference rule or axiom may contain more than one occurrence of a particular variable requires that an identity relation be defined on each kind of object over which a variable can range. The identity relation for all kinds of object is value-based. Two objects of a particular kind are identical if the constituents of the objects are identical. For example, two attributes are considered the same if they have the same name and the same value. Two characters are identical if their Unicode character codes are the same.

6.1. Name classes

The main semantic concept for name classes is that of a name belonging to a name class. A name class is an element that matches the production nameClass. A name is as defined in Section 2: it consists of a namespace URI and a local name.

We use the following notation:

n

is a variable that ranges over names

nc
: ranges over name classes

n in *nc*
: asserts that name *n* is a member of name class *nc*

We are now ready for our first axiom, which is called "anyName 1":

(anyName 1) *n* in `<anyName/>`

This says for any name *n*, *n* belongs to the name class `<anyName/>`, in other words `<anyName/>` matches any name. Note the effect of the implicit universal quantification over the variables in the axiom: this is what makes the axiom apply for any name *n*.

Our first inference rule is almost as simple:

(anyName 2) $$\frac{\text{not}(n \text{ in } nc)}{n \text{ in } \texttt{<anyName> <except>}\ nc\ \texttt{</except> </anyName>}}$$

This says that for any name *n* and for any name class *nc*, if *n* does not belong to *nc*, then *n* belongs to `<anyName> <except>` *nc* `</except> </anyName>`. In other words, `<anyName> <except>` *nc* `</except> </anyName>` matches any name that does not match *nc*.

We now need the following additional notation:

ln
: ranges over local names; a local name is a string that matches the NCName production of [XML Namespaces], that is, a name with no colons

u
: ranges over URIs

name(*u*, *ln*)
: constructs a name with URI *u* and local name *ln*

The remaining axioms and inference rules for name classes are as follows:

(nsName 1) name(*u*, *ln*) in `<nsName ns="`*u*`"/>`

(nsName 2) $$\frac{\text{not}(\text{name}(\ u, ln\ ) \text{ in } nc)}{\text{name}(\ u, ln\ ) \text{ in } \texttt{<nsName ns="}u\texttt{"> <except>}\ nc\ \texttt{</except> </nsName>}}$$

(name) name(*u*, *ln*) in `<name ns="`*u*`">` *ln* `</name>`

(name choice 1) $$\frac{n \text{ in } nc_1}{n \text{ in } \texttt{<choice>}\ nc_1\ nc_2\ \texttt{</choice>}}$$

(name choice 2) $$\frac{n \text{ in } nc_2}{n \text{ in } \texttt{<choice>}\ nc_1\ nc_2\ \texttt{</choice>}}$$

6.2. Patterns

The axioms and inference rules for patterns use the following notation:

cx
: ranges over contexts (as defined in Section 2)

a
: ranges over sets of attributes; a set with a single member is considered the same as that member

m
: ranges over sequences of elements and strings; a sequence with a single member is considered the same as that member; the sequences ranged over by *m* may contain consecutive strings and may contain strings that are empty; thus, there are sequences ranged over by *m* that cannot occur as the children of an element

p
: ranges over patterns (elements matching the pattern production)

cx |- *a*; *m* =~ *p*
: asserts that with respect to context *cx*, the attributes *a* and the sequence of elements and strings *m* matches the pattern *p*

6.2.1. `choice` pattern

The semantics of the `choice` pattern are as follows:

(choice 1) $$\frac{cx \text{ |- } a;\ m \text{ =\~{} } p_1}{cx \text{ |- } a;\ m \text{ =\~{} } \texttt{<choice>}\ p_1\ p_2\ \texttt{</choice>}}$$

(choice 2) $$\frac{cx \text{ |- } a;\ m \text{ =\~{} } p_2}{cx \text{ |- } a;\ m \text{ =\~{} } \texttt{<choice>}\ p_1\ p_2\ \texttt{</choice>}}$$

6.2.2. `group` pattern

We use the following additional notation:

m_1, m_2

represents the concatenation of the sequences m_1 and m_2

$a_1 + a_2$
: represents the union of a_1 and a_2

The semantics of the `group` pattern are as follows:

(group) $$\frac{cx \text{ |- } a_1;\ m_1 \text{ =\~ } p_1 \qquad cx \text{ |- } a_2;\ m_2 \text{ =\~ } p_2}{cx \text{ |- } a_1 + a_2;\ m_1, m_2 \text{ =\~ } \texttt{<group>}\ p_1\ p_2\ \texttt{</group>}}$$

> **Note**
>
> The restriction in Section 7.3 ensures that the set of attributes constructed in the consequent will not have multiple attributes with the same name.

6.2.3. `empty` pattern

We use the following additional notation:

( )
: represents an empty sequence

{ }
: represents an empty set

The semantics of the `empty` pattern are as follows:

(empty) $cx \text{ |- } \{\ \};\ (\ ) \text{ =\~ } \texttt{<empty/>}$

6.2.4. `text` pattern

We use the following additional notation:

s
: ranges over strings

The semantics of the `text` pattern are as follows:

(text 1) $cx \text{ |- } \{\ \};\ (\ ) \text{ =\~ } \texttt{<text/>}$

(text 2) $$\frac{cx \text{ |- } \{\ \};\ m \text{ =\~ } \texttt{<text/>}}{cx \text{ |- } \{\ \};\ m, s \text{ =\~ } \texttt{<text/>}}$$

The effect of the above rule is that a `text` element matches zero or more strings.

6.2.5. `oneOrMore` pattern

We use the following additional notation:

disjoint(a_1, a_2)
: asserts that there is no name that is the name of both an attribute in a_1 and of an attribute in a_2

The semantics of the `oneOrMore` pattern are as follows:

(oneOrMore 1) $$\frac{cx \text{ |- } a;\ m \text{ =\~{} } p}{cx \text{ |- } a;\ m \text{ =\~{} } \texttt{<oneOrMore>}\ p\ \texttt{</oneOrMore>}}$$

(oneOrMore 2) $$\frac{cx \text{ |- } a_1;\ m_1 \text{ =\~{} } p \qquad cx \text{ |- } a_2;\ m_2 \text{ =\~{} } \texttt{<oneOrMore>}\ p\ \texttt{</oneOrMore>} \qquad \text{disjoint}(a_1, a_2)}{cx \text{ |- } a_1 + a_2;\ m_1, m_2 \text{ =\~{} } \texttt{<oneOrMore>}\ p\ \texttt{</oneOrMore>}}$$

6.2.6. `interleave` pattern

We use the following additional notation:

m_1 interleaves m_2; m_3
: asserts that m_1 is an interleaving of m_2 and m_3

The semantics of interleaving are defined by the following rules.

(interleaves 1) $$(\ ) \text{ interleaves } (\ );\ (\ )$$

(interleaves 2) $$\frac{m_1 \text{ interleaves } m_2;\ m_3}{m_4,\ m_1 \text{ interleaves } m_4,\ m_2;\ m_3}$$

(interleaves 3) $$\frac{m_1 \text{ interleaves } m_2;\ m_3}{m_4,\ m_1 \text{ interleaves } m_2;\ m_4,\ m_3}$$

For example, the interleavings of `<a/><a/>` and `<b/>` are `<a/><a/><b/>`, `<a/><b/><a/>`, and `<b/><a/><a/>`.

The semantics of the `interleave` pattern are as follows:

(interleave) $$\frac{cx \text{ |- } a_1; m_1 \sim p_1 \quad cx \text{ |- } a_2; m_2 \sim p_2 \quad m_3 \text{ interleaves } m_1; m_2}{cx \text{ |- } a_1 + a_2; m_3 \sim \texttt{<interleave>}\ p_1\ p_2\ \texttt{</interleave>}}$$

Note

The restriction in Section 7.3 ensures that the set of attributes constructed in the consequent will not have multiple attributes with the same name.

6.2.7. `element` and `attribute` pattern

The value of an attribute is always a single string, which may be empty. Thus, the empty sequence is not a possible attribute value. On the hand, the children of an element can be an empty sequence and cannot consist of an empty string. In order to ensure that validation handles attributes and elements consistently, we introduce a variant of matching called *weak matching*. Weak matching is used when matching the pattern for the value of an attribute or for the attributes and children of an element. We use the following notation to define weak matching.

`""`
: represents an empty string

ws
: ranges over the empty sequence and strings that consist entirely of whitespace

cx |- a; $m \sim_{\text{weak}} p$
: asserts that with respect to context cx, the attributes a and the sequence of elements and strings m weakly matches the pattern p

The semantics of weak matching are as follows:

(weak match 1) $$\frac{cx \text{ |- } a; m \sim p}{cx \text{ |- } a; m \sim_{\text{weak}} p}$$

(weak match 2) $$\frac{cx \text{ |- } a; (\,) \sim p}{cx \text{ |- } a; ws \sim_{\text{weak}} p}$$

(weak match 3) $$\frac{cx \text{ |- } a; \texttt{""} \sim p}{cx \text{ |- } a; (\,) \sim_{\text{weak}} p}$$

We use the following additional notation:

attribute(n, s)

constructs an attribute with name *n* and value *s*

element(*n*, *cx*, *a*, *m*)
: constructs an element with name *n*, context *cx*, attributes *a* and mixed sequence *m* as children

okAsChildren(*m*)
: asserts that the mixed sequence *m* can occur as the children of an element: it does not contain any member that is an empty string, nor does it contain two consecutive members that are both strings

deref(*ln*) = `<element>` *nc p* `</element>`
: asserts that the grammar contains `<define name="`*ln*`">` `<element>` *nc p* `</element>` `</define>`

The semantics of the `attribute` pattern are as follows:

$$\text{(attribute)}\quad \frac{cx \text{ |- } \{\ \};\ s =\sim_{\text{weak}} p \qquad n \text{ in } nc}{cx \text{ |- attribute(} n, s \text{); ( ) } =\sim \texttt{<attribute>}\ nc\ p\ \texttt{</attribute>}}$$

The semantics of the `element` pattern are as follows:

$$\text{(element)}\quad \frac{cx_1 \text{ |- } a;\ m \quad =\sim_{\text{weak}} p \qquad n \text{ in } nc \qquad \text{okAsChildren}(m) \qquad \text{deref}(ln) = \texttt{<element>}\ nc\ p\ \texttt{</element>}}{cx_2 \text{ |- } \{\ \};\ ws_1, \text{ element(} n, cx_1, a, m \text{), } ws_2 =\sim \texttt{<ref name="}ln\texttt{"/>}}$$

6.2.8. `data` and `value` pattern

RELAX NG relies on datatype libraries to perform datatyping. A datatype library is identified by a URI. A datatype within a datatype library is identified by an NCName. A datatype library provides two services.

- It can determine whether a string is a legal representation of a datatype. This service accepts a list of zero or more parameters. For example, a string datatype might have a parameter specifying the length of a string. The datatype library determines what parameters are applicable for each datatype.
- It can determine whether two strings represent the same value of a datatype. This service does not have any parameters.

Both services may make use of the context of a string. For example, a datatype representing a QName would use the namespace map.

We use the following additional notation:

datatypeAllows(*u*, *ln*, *params*, *s*, *cx*)
: asserts that in the datatype library identified by URI *u*, the string *s* interpreted with context *cx* is a legal value of datatype *ln* with parameters *params*

datatypeEqual(u, ln, s_1, cx_1, s_2, cx_2)

asserts that in the datatype library identified by URI u, string s_1 interpreted with context cx_1 represents the same value of the datatype ln as the string s_2 interpreted in the context of cx_2

params

ranges over sequences of parameters

[*cx*]

within the start-tag of a pattern refers to the context of the pattern element

context(u, cx)

constructs a context which is the same as cx except that the default namespace is u; if u is the empty string, then there is no default namespace in the constructed context

The datatypeEqual function must be reflexive, transitive and symmetric, that is, the following inference rules must hold:

(datatypeEqual reflexive) $$\frac{\text{datatypeAllows}(u, ln, params, s, cx)}{\text{datatypeEqual}(u, ln, s, cx, s, cx)}$$

(datatypeEqual transitive) $$\frac{\text{datatypeEqual}(u, ln, s_1, cx_1, s_2, cx_2) \qquad \text{datatypeEqual}(u, ln, s_2, cx_3, s_3, cx_3)}{\text{datatypeEqual}(u, ln, s_1, cx_1, s_3, cx_3)}$$

(datatypeEqual symmetric) $$\frac{\text{datatypeEqual}(u, ln, s_1, cx_1, s_2, cx_2)}{\text{datatypeEqual}(u, ln, s_2, cx_2, s_1, cx_1)}$$

The semantics of the `data` and `value` patterns are as follows:

(value) $$\frac{\text{datatypeEqual}(u_1, ln, s_1, cx_1, s_2, \text{context}(\ u_2, cx_2\ ))}{cx_1 \text{ |- \{ \}; } s_1 \text{ =\sim } \texttt{<value datatypeLibrary="}u_1\texttt{" type="}ln\texttt{" ns="}u_2\texttt{" [}cx_2\texttt{]>}\ s_2\ \texttt{</value>}}$$

(data 1) $$\frac{\text{datatypeAllows}(u, ln, params, s, cx)}{cx \text{ |- \{ \}; } s \text{ =\sim } \texttt{<data datatypeLibrary="}u\texttt{" type="}ln\texttt{">}\ params\ \texttt{</data>}}$$

(data 2) $$\frac{\text{datatypeAllows}(u, ln, params, s, cx) \qquad \text{not}(cx \text{ |- } a; s \text{ =\sim } p)}{cx \text{ |- \{ \}; } s \text{ =\sim } \texttt{<data datatypeLibrary="}u\texttt{" type="}ln\texttt{">}\ params\ \texttt{<except>}\ p\ \texttt{</except> </data>}}$$

6.2.9. Built-in datatype library

The empty URI identifies a special built-in datatype library. This provides two datatypes, `string` and `token`. No parameters are allowed for either of these datatypes.

$s_1 = s_2$
: asserts that s_1 and s_2 are identical

normalizeWhiteSpace(s)
: returns the string s, with leading and trailing whitespace characters removed, and with each other maximal sequence of whitespace characters replaced by a single space character

The semantics of the two built-in datatypes are as follows:

(string allows) datatypeAllows("", "string", ( ), s, cx)

(string equal) datatypeEqual("", "string", s, cx_1, s, cx_2)

(token allows) datatypeAllows("", "token", ( ), s, cx)

(token equal) $$\frac{\text{normalizeWhiteSpace}(\ s_1\ ) = \text{normalizeWhiteSpace}(\ s_2\ )}{\text{datatypeEqual}(\text{"", "token"}, s_1, cx_1, s_2, cx_2)}$$

6.2.10. `list` pattern

We use the following additional notation:

split(s)
: returns a sequence of strings one for each whitespace delimited token of s; each string in the returned sequence will be non-empty and will not contain any whitespace

The semantics of the `list` pattern are as follows:

(list) $$\frac{cx \vdash \{\ \};\ \text{split}(\ s\ ) =\sim p}{cx \vdash \{\ \};\ s =\sim \texttt{<list>}\ p\ \texttt{</list>}}$$

Note

It is crucial in the above inference rule that the sequence that is matched against a pattern can contain consecutive strings.

6.3. Validity

Now we can define when an element is valid with respect to a schema. We use the following additional notation:

e
: ranges over elements

valid(*e*)
: asserts that the element *e* is valid with respect to the grammar

start() = *p*
: asserts that the grammar contains `<start>` *p* `</start>`

An element is valid if together with an empty set of attributes it matches the `start` pattern of the grammar.

$$\text{(valid)} \quad \frac{\text{start}() = p \quad cx \vdash \{\}; e =\sim p}{\text{valid}(e)}$$

6.4. Example

Let e_0 be

element(name(`""`, `"foo"`), cx_0, { }, m)

where m is

e_1, e_2

and e_1 is

element(name(`"http://www.example.com/n1"`, `"bar1"`), cx_1, { }, ( ))

and e_2 is

element(name(`"http://www.example.com/n2"`, `"bar2"`), cx_2, { }, ( ))

Assuming appropriate definitions of cx_0, cx_1 and cx_2, this represents the document in Section 2.1.

We now show how e_0 can be shown to be valid with respect to the schema in Section 5.1. The schema is equivalent to the following propositions:

start() = `<ref name="foo"/>`
deref(`"foo.element"`) = `<element> <name ns=""> "foo" </name> <group> <ref name="bar1"/> <ref name="bar2"/> </group> </element>`
deref(`"bar1.element"`) = `<element> <name ns="http://www.example.com/n1"> "bar1" </name> <empty/> </element>`

deref("bar2.element") = <element> <name ns="http://www.example.com/n2">
"bar2" </name> <empty/> </element>

Let name class nc_1 be

<name ns="http://www.example.com/n1"> "bar1" </name>

and let nc_2 be

<name ns="http://www.example.com/n2"> "bar2" </name>

Then, by the inference rule (name) in Section 6.1, we have

name("http://www.example.com/n1", "bar1") in nc_1

and

name("http://www.example.com/n2", "bar2") in nc_2

By the inference rule (empty) in Section 6.2.3, we have

cx_1 |- { }; ( ) =~ <empty/>

and

cx_2 |- { }; ( ) =~ <empty/>

Thus by the inference rule (element) in Section 6.2.7, we have

cx_0 |- { }; e_1 =~ <ref name="bar1"/>

Note that we have chosen cx_0, since any context is allowed.

Likewise, we have

cx_0 |- { }; e_2 =~ <ref name="bar2"/>

By the inference rule (group) in Section 6.2.1, we have

cx_0 |- { }; e_1, e_2 =~ <group> <ref name="bar1"/> <ref name="bar2"/> </group>

By the inference rule (element) in Section 6.2.7, we have

cx_3 |- { }; element(name("", "foo"), cx_0, { }, m) =~ <ref name="foo"/>

Here cx_3 is an arbitrary context.

Thus we can apply the inference rule (valid) in Section 6.3 and obtain

valid(e_0)

7. Restrictions

The following constraints are all checked after the grammar has been transformed to the simple form described in Section 5. The purpose of these restrictions is to catch user errors and to facilitate implementation.

7.1. Contextual restrictions

In this section we describe restrictions on where elements are allowed in the schema based on the names of the ancestor elements. We use the concept of a *prohibited path* to describe these restrictions. A path is a sequence of NCNames separated by `/` or `//`.

- An element matches a path *x*, where *x* is an NCName, if and only if the local name of the element is *x*
- An element matches a path *x*`/`*p*, where *x* is an NCName and *p* is a path, if and only if the local name of the element is *x* and the element has a child that matches *p*
- An element matches a path *x*`//`*p*, where *x* is an NCName and *p* is a path, if and only if the local name of the element is *x* and the element has a descendant that matches *p*

For example, the element

```
<foo>
  <bar>
    <baz/>
  </bar>
</foo>
```

matches the paths `foo`, `foo/bar`, `foo//bar`, `foo//baz`, `foo/bar/baz`, `foo/bar//baz` and `foo//bar/baz`, but not `foo/baz` or `foobar`.

A correct RELAX NG schema must be such that, after transformation to the simple form, it does not contain any element that matches a prohibited path.

7.1.1. `attribute` pattern

The following paths are prohibited:

- `attribute//ref`
- `attribute//attribute`

7.1.2. `oneOrMore` pattern

The following paths are prohibited:

- `oneOrMore//group//attribute`
- `oneOrMore//interleave//attribute`

7.1.3. `list` pattern

The following paths are prohibited:

- `list//list`
- `list//ref`
- `list//attribute`
- `list//text`
- `list//interleave`

7.1.4. `except` in `data` pattern

The following paths are prohibited:

- `data/except//attribute`
- `data/except//ref`
- `data/except//text`
- `data/except//list`
- `data/except//group`
- `data/except//interleave`
- `data/except//oneOrMore`
- `data/except//empty`

> **Note**
>
> This implies that an `except` element with a `data` parent can contain only `data`, `value` and `choice` elements.

7.1.5. `start` element

The following paths are prohibited:

- `start//attribute`
- `start//data`
- `start//value`
- `start//text`
- `start//list`
- `start//group`
- `start//interleave`
- `start//oneOrMore`
- `start//empty`

7.2. String sequences

RELAX NG does not allow a pattern such as:

```
<element name="foo">
  <group>
    <data type="int"/>
    <element name="bar">
      <empty/>
```

```
    </element>
  </group>
</element>
```

Nor does it allow a pattern such as:

```
<element name="foo">
  <group>
    <data type="int"/>
    <text/>
  </group>
</element>
```

More generally, if the pattern for the content of an element or attribute contains

- a pattern that can match a child (that is, an `element`, `data`, `value`, `list` or `text` pattern), and
- a pattern that matches a single string (that is, a `data`, `value` or `list` pattern),

then the two patterns must be alternatives to each other.

This rule does not apply to patterns occurring within a `list` pattern.

To formalize this, we use the concept of a content-type. A pattern that is allowable as the content of an element has one of three content-types: empty, complex and simple. We use the following notation.

empty( )
: returns the empty content-type

complex( )
: returns the complex content-type

simple( )
: returns the simple content-type

ct
: ranges over content-types

groupable(ct_1, ct_2)
: asserts that the content-types ct_1 and ct_2 are groupable

The empty content-type is groupable with anything. In addition, the complex content-type is groupable with the complex content-type. The following rules formalize this.

(group empty 1) groupable(empty( ), *ct*)

(group empty 2) groupable(*ct*, empty( ))

(group complex) groupable(complex( ), complex( ))

Some patterns have a content-type. We use the following additional notation.

p :$_c$ ct
: asserts that pattern p has content-type ct

max(ct_1, ct_2)
: returns the maximum of ct_1 and ct_2 where the content-types in increasing order are empty( ), complex( ), simple( )

The following rules define when a pattern has a content-type and, if so, what it is.

(value) `<value datatypeLibrary="`u_1`" type="`ln`" ns="`u_2`">` s `</value>` :$_c$ simple( )

(data 1) `<data datatypeLibrary="`u`" type="`ln`">` *params* `</data>` :$_c$ simple( )

(data 2)
$$\frac{p :_c ct}{\texttt{<data datatypeLibrary="}u\texttt{" type="}ln\texttt{">}\ params\ \texttt{<except>}\ p\ \texttt{</except>}\ \texttt{</data>} :_c \text{simple( )}}$$

(list) `<list>` p `</list>` :$_c$ simple( )

(text) `<text/>` :$_c$ complex( )

(ref) `<ref name="`ln`"/>` :$_c$ complex( )

(empty) `<empty/>` :$_c$ empty( )

(attribute)
$$\frac{p :_c ct}{\texttt{<attribute>}\ nc\ p\ \texttt{</attribute>} :_c \text{empty( )}}$$

(group)
$$\frac{p_1 :_c ct_1 \quad p_2 :_c ct_2 \quad \text{groupable}(ct_1, ct_2)}{\texttt{<group>}\ p_1\ p_2\ \texttt{</group>} :_c \max(\ ct_1, ct_2\ )}$$

(interleave)
$$\frac{p_1 :_c ct_1 \qquad p_2 :_c ct_2 \qquad \text{groupable}(ct_1, ct_2)}{\texttt{<interleave>}\ p_1\ p_2\ \texttt{</interleave>} :_c \max(\ ct_1, ct_2\ )}$$

(oneOrMore) $$\frac{p :_c ct \qquad \text{groupable}(ct, ct)}{\texttt{<oneOrMore>}\ p\ \texttt{</oneOrMore>} :_c ct}$$

(choice) $$\frac{p_1 :_c ct_1 \qquad p_2 :_c ct_2}{\texttt{<choice>}\ p_1\ p_2\ \texttt{</choice>} :_c \max(\ ct_1, ct_2\ )}$$

> **Note**
>
> The antecedent in the (data 2) rule above is in fact redundant because of the prohibited paths in Section 7.1.4.

Now we can describe the restriction. We use the following notation.

incorrectSchema()
: asserts that the schema is incorrect

All patterns occurring as the content of an element pattern must have a content-type.

(element) $$\frac{\text{deref}(ln) = \texttt{<element>}\ nc\ p\ \texttt{</element>} \quad \text{not}(p :_c ct)}{\text{incorrectSchema}()}$$

7.3. Restrictions on attributes

Duplicate attributes are not allowed. More precisely, for a pattern `<group> p1 p2 </group>` or `<interleave> p1 p2 </interleave>`, there must not be a name that belongs to both the name class of an `attribute` pattern occurring in *p1* and the name class of an `attribute` pattern occurring in *p2*. A pattern *p1* is defined to *occur in* a pattern *p2* if

- *p1* is *p2*, or
- *p2* is a `choice`, `interleave`, `group` or `oneOrMore` element and *p1* occurs in one or more children of *p2*.

Attributes using infinite name classes must be repeated. More precisely, an `attribute` element that has an `anyName` or `nsName` descendant element must have a `oneOrMore` ancestor element.

> **Note**
>
> This restriction is necessary for closure under negation.

7.4. Restrictions on `interleave`

For a pattern `<interleave> p1 p2< /interleave>`,

- there must not be a name that belongs to both the name class of an `element` pattern referenced by a `ref` pattern occurring in *`p1`* and the name class of an `element` pattern referenced by a `ref` pattern occurring in *`p2`*, and
- a `text` pattern must not occur in both *`p1`* and *`p2`*.

Section 7.3 defines when one pattern is considered to occur in another pattern.

8. Conformance

A conforming RELAX NG validator must be able to determine for any XML document whether it is a correct RELAX NG schema. A conforming RELAX NG validator must be able to determine for any XML document and for any correct RELAX NG schema whether the document is valid with respect to the schema.

However, the requirements in the preceding paragraph do not apply if the schema uses a datatype library that the validator does not support. A conforming RELAX NG validator is only required to support the built-in datatype library described in Section 6.2.9. A validator that claims conformance to RELAX NG should document which datatype libraries it supports. The requirements in the preceding paragraph also do not apply if the schema includes `externalRef` or `include` elements and the validator is unable to retrieve the resource identified by the URI or is unable to construct an element from the retrieved resource. A validator that claims conformance to RELAX NG should document its capabilities for handling URI references.

A. RELAX NG schema for RELAX NG

```
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
  ns="http://relaxng.org/ns/structure/1.0"
  xmlns="http://relaxng.org/ns/structure/1.0">

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

  <define name="pattern">
    <choice>
      <element name="element">
        <choice>
          <attribute name="name">
            <data type="QName"/>
          </attribute>
          <ref name="open-name-class"/>
        </choice>
        <ref name="common-atts"/>
        <ref name="open-patterns"/>
      </element>
      <element name="attribute">
        <ref name="common-atts"/>
        <choice>
          <attribute name="name">
            <data type="QName"/>
          </attribute>
          <ref name="open-name-class"/>
```

```
  </choice>
  <interleave>
    <ref name="other"/>
    <optional>
      <ref name="pattern"/>
    </optional>
  </interleave>
</element>
<element name="group">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="interleave">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="choice">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="optional">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="zeroOrMore">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="oneOrMore">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="list">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="mixed">
  <ref name="common-atts"/>
  <ref name="open-patterns"/>
</element>
<element name="ref">
  <attribute name="name">
    <data type="NCName"/>
  </attribute>
  <ref name="common-atts"/>
</element>
<element name="parentRef">
  <attribute name="name">
    <data type="NCName"/>
  </attribute>
  <ref name="common-atts"/>
</element>
<element name="empty">
  <ref name="common-atts"/>
  <ref name="other"/>
</element>
<element name="text">
  <ref name="common-atts"/>
```

```
      <ref name="other"/>
    </element>
    <element name="value">
      <optional>
        <attribute name="type">
          <data type="NCName"/>
        </attribute>
      </optional>
      <ref name="common-atts"/>
      <text/>
    </element>
    <element name="data">
      <attribute name="type">
        <data type="NCName"/>
      </attribute>
      <ref name="common-atts"/>
      <interleave>
        <ref name="other"/>
        <group>
          <zeroOrMore>
            <element name="param">
              <attribute name="name">
                <data type="NCName"/>
              </attribute>
              <text/>
            </element>
          </zeroOrMore>
          <optional>
            <element name="except">
              <ref name="common-atts"/>
              <ref name="open-patterns"/>
            </element>
          </optional>
        </group>
      </interleave>
    </element>
    <element name="notAllowed">
      <ref name="common-atts"/>
      <ref name="other"/>
    </element>
    <element name="externalRef">
      <attribute name="href">
        <data type="anyURI"/>
      </attribute>
      <ref name="common-atts"/>
      <ref name="other"/>
    </element>
    <element name="grammar">
      <ref name="common-atts"/>
      <ref name="grammar-content"/>
    </element>
  </choice>
</define>

<define name="grammar-content">
  <interleave>
    <ref name="other"/>
    <zeroOrMore>
```

```
      <choice>
        <ref name="start-element"/>
        <ref name="define-element"/>
        <element name="div">
          <ref name="common-atts"/>
          <ref name="grammar-content"/>
        </element>
        <element name="include">
          <attribute name="href">
            <data type="anyURI"/>
          </attribute>
          <ref name="common-atts"/>
          <ref name="include-content"/>
        </element>
      </choice>
    </zeroOrMore>
  </interleave>
</define>

<define name="include-content">
  <interleave>
    <ref name="other"/>
    <zeroOrMore>
      <choice>
        <ref name="start-element"/>
        <ref name="define-element"/>
        <element name="div">
          <ref name="common-atts"/>
          <ref name="include-content"/>
        </element>
      </choice>
    </zeroOrMore>
  </interleave>
</define>

<define name="start-element">
  <element name="start">
    <ref name="combine-att"/>
    <ref name="common-atts"/>
    <ref name="open-pattern"/>
  </element>
</define>

<define name="define-element">
  <element name="define">
    <attribute name="name">
      <data type="NCName"/>
    </attribute>
    <ref name="combine-att"/>
    <ref name="common-atts"/>
    <ref name="open-patterns"/>
  </element>
</define>

<define name="combine-att">
  <optional>
    <attribute name="combine">
      <choice>
```

```
        <value>choice</value>
        <value>interleave</value>
      </choice>
    </attribute>
  </optional>
</define>

<define name="open-patterns">
  <interleave>
    <ref name="other"/>
    <oneOrMore>
      <ref name="pattern"/>
    </oneOrMore>
  </interleave>
</define>

<define name="open-pattern">
  <interleave>
    <ref name="other"/>
    <ref name="pattern"/>
  </interleave>
</define>

<define name="name-class">
  <choice>
    <element name="name">
      <ref name="common-atts"/>
      <data type="QName"/>
    </element>
    <element name="anyName">
      <ref name="common-atts"/>
      <ref name="except-name-class"/>
    </element>
    <element name="nsName">
      <ref name="common-atts"/>
      <ref name="except-name-class"/>
    </element>
    <element name="choice">
      <ref name="common-atts"/>
      <ref name="open-name-classes"/>
    </element>
  </choice>
</define>

<define name="except-name-class">
  <interleave>
    <ref name="other"/>
    <optional>
      <element name="except">
        <ref name="open-name-classes"/>
      </element>
    </optional>
  </interleave>
</define>

<define name="open-name-classes">
  <interleave>
    <ref name="other"/>
```

```
    <oneOrMore>
      <ref name="name-class"/>
    </oneOrMore>
  </interleave>
</define>

<define name="open-name-class">
  <interleave>
    <ref name="other"/>
    <ref name="name-class"/>
  </interleave>
</define>

<define name="common-atts">
  <optional>
    <attribute name="ns"/>
  </optional>
  <optional>
    <attribute name="datatypeLibrary">
      <data type="anyURI"/>
    </attribute>
  </optional>
  <zeroOrMore>
    <attribute>
      <anyName>
        <except>
          <nsName/>
          <nsName ns=""/>
        </except>
      </anyName>
    </attribute>
  </zeroOrMore>
</define>

<define name="other">
  <zeroOrMore>
    <element>
      <anyName>
        <except>
          <nsName/>
        </except>
      </anyName>
      <zeroOrMore>
        <choice>
          <attribute>
            <anyName/>
          </attribute>
          <text/>
          <ref name="any"/>
        </choice>
      </zeroOrMore>
    </element>
  </zeroOrMore>
</define>

<define name="any">
  <element>
    <anyName/>
```

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

</grammar>
```

B. Changes since version 0.9

The changes in this version relative to version 0.9 are as follows:

- in the namespace URI, `0.9` has been changed to `1.0`
- `data/except//empty` has been added as a prohibited path (see Section 7.1.4)
- `start//empty` has been added as a prohibited path (see Section 7.1.5)
- Section 4.12 now specifies how a `list` element with more than one child element is transformed
- Section 4.20 now specifies how a `notAllowed` element occurring in an `except` element is transformed
- although a relative URI is not allowed as the value of the `ns` and `datatypeLibrary` attributes, an empty string is allowed (see Section 3)
- the removal of unreachable definitions in Section 4.19 is now correctly specified
- Section 4.20 now specifies that `define` elements that are no longer reachable are removed
- Section 4.16 has been added; the restrictions on the contents of `except` in name classes that are now specified in the newly added section were previously specified in a subsection of Section 7.1, which has been removed
- the treatment of element and attribute values that consist only of whitespace has been refined (see Section 6.2.7 and Section 6.2.8)
- attributes with infinite name classes are now required to be repeated (see Section 7.3)
- restrictions have been imposed on `interleave` (see Section 7.4); `list//interleave` has been added as a prohibited path (see Section 7.1.3)
- some of the prohibited paths in Section 7.1 have been corrected to use `ref` rather than `element`
- an error in the inference rule (text 1) in Section 6.2.4 has been corrected
- the value of the `ns` attribute is now unconstrained (see Section 3)

C. RELAX NG TC (Non-Normative)

This specification was prepared and approved for publication by the RELAX NG TC. The current members of the TC are:

- Fabio Arciniegas
- James Clark
- Mike Fitzgerald

- KAWAGUCHI Kohsuke
- Josh Lubell
- MURATA Makoto
- Norman Walsh
- David Webber

References

Normative

RFC 2396
: T. Berners-Lee, R. Fielding, L. Masinter. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*. IETF (Internet Engineering Task Force). 1998.

RFC 2732
: R. Hinden, B. Carpenter, L. Masinter. *RFC 2732: Format for Literal IPv6 Addresses in URL's*. IETF (Internet Engineering Task Force), 1999.

RFC 3023
: M. Murata, S. St.Laurent, D. Kohn. *RFC 3023: XML Media Types*. IETF (Internet Engineering Task Force), 2001.

XLink
: Steve DeRose, Eve Maler and David Orchard, editors. *XML Linking Language (XLink) Version 1.0*. W3C (World Wide Web Consortium), 2001.

XML 1.0
: Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, Eve Maler, editors. *Extensible Markup Language (XML) 1.0 Second Edition*. W3C (World Wide Web Consortium), 2000.

XML Infoset
: John Cowan, Richard Tobin, editors. *XML Information Set*. W3C (World Wide Web Consortium), 2001.

XML Namespaces
: Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. W3C (World Wide Web Consortium), 1999.

Non-Normative

RELAX
: MURATA Makoto. *RELAX (Regular Language description for XML)*. INSTAC (Information Technology Research and Standardization Center), 2001.

TREX
: James Clark. *TREX - Tree Regular Expressions for XML*. Thai Open Source Software Center, 2001.

Tutorial
: James Clark, Makoto MURATA, editors. *RELAX NG Tutorial*. OASIS, 2001.

W3C XML Schema Datatypes
: Paul V. Biron, Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. W3C (World Wide Web Consortium), 2001.

XML Schema Formal

Allen Brown, Matthew Fuchs, Jonathan Robie, Philip Wadler, editors. *XML Schema: Formal Description*. W3C (World Wide Web Consortium), 2001.