HOW TO RELAX

MURATA Makoto

$Id: howToRELAX.sdoc 1.9 2000/08/26 03:12:38 murata Exp $

Contents

Part I

Part 1: RELAX Core

Chapter 1

STEP 0: Introduction

$Id: step0.sdoc 1.10 2000/08/06 08:45:41 murata Exp $

STEP 0 gives a brief overview of RELAX and shows how to read this tutorial.

1.1 RELAX, brief overview

RELAX is a specification for describing XML-based languages. XHTML 1.0, for example, can be described in RELAX. A description written in RELAX is called a *RELAX grammar*.

1.1.1 Comparison with DTD

Compared with a traditional DTD (Document Type Definition), RELAX has new features as below:

- A RELAX grammar can be written as an XML document.
- RELAX borrows rich datatypes from XML Schema Part 2.
- RELAX is namespace-aware.

1.1.2 The RELAX processor

The RELAX processor verifies XML documents against RELAX grammars. The input to the RELAX processors is an XML document and a RELAX grammar. To be precise, the RELAX processor does not directly handle XML documents and RELAX grammars, but rather receives the output of the XML processor which handles them.

The RELAX processor reports if the XML document is legitimate against the RELAX grammar. It may also report some other messages. The RELAX processor has no other outputs.

1.1.3 The organization of RELAX

RELAX consists of RELAX Core and RELAX Namespace. RELAX Core handles elements in a single namespace and their attributes. RELAX Core borrows datatypes from XML Schema Part 2[1]. RELAX Namespace combines multiple modules so as to handle multiple namespaces. At present, this tutorial covers RELAX Core only.

1.1.4 About this tutorial

This tutorial is intended to be very easy to understand. All STEPs except this one use plenty of examples and provide concrete explanations.

STEPs 1 thru 10 are concerned with RELAX Core. You may stop at the end of any step and still have a reasonable understanding of RELAX. You can start to use RELAX immediately after reading STEP 1. If you read through STEP 6, you will know all the features shared by RELAX and XML DTDs. All RELAX processors are required to support these features. STEPs 7 thru 10 explain new features of RELAX. RELAX processors are not required to support these new features.

A little inaccuracy saves tons of explanation. STEPs 1 and 2 are sometimes inaccurate. Accurate explanations are provided by STEPs 7 and 8.

1.2 Summary

RELAX is very simple. If you are familiar with DTDs, you can be fluent in RELAX almost immediately. Even if you are not, you can easily master RELAX. Enjoy and RELAX!

[1] http://www.w3.org/TR/xmlschema-2/

Chapter 2

STEP 1: Migration from XML DTD (without parameter entities)

$Id: step1.sdoc 1.13 2000/08/06 08:47:44 murata Exp $

STEP 1 covers basic features, which allow easy migration from DTDs. The DTD2RELAX converter uses these features only.

2.1 An example module

To provide an idea of RELAX, we will recapture a DTD as a RELAX module.

A DTD is shown below. The `number` attribute of `title` elements should be integers, but DTDs cannot represent this constraint.

```
<!ELEMENT doc   (title, para*)>

<!ELEMENT para  (#PCDATA | em)*>

<!ELEMENT title (#PCDATA | em)*>

<!ELEMENT em    (#PCDATA)>

<!ATTLIST para
  class   NMTOKEN #IMPLIED
>

<!ATTLIST title
  class   NMTOKEN #IMPLIED
  number  CDATA   #REQUIRED
>
```

Next, we show a RELAX module. The `number` attribute is specified as an integer.

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc">
    <sequence>
      <ref label="title"/>
      <ref label="para" occurs="*"/>
    </sequence>
  </elementRule>

  <elementRule role="para">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <elementRule role="title">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <elementRule role="em" type="string"/>

  <tag name="doc"/>

  <tag name="para">
    <attribute name="class" type="NMTOKEN"/>
  </tag>

  <tag name="title">
    <attribute name="class" type="NMTOKEN"/>
    <attribute name="number" required="true" type="integer"/>
  </tag>

  <tag name="em"/>

</module>
```

Subsequent sections explain syntactical constructs appeared in this example.

2.2 The `module` element

A RELAX grammar is a combination of modules. If the number of namespaces is one and the grammar is not too large, a module provides a RELAX grammar. A module is represented by a `module` element.

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">
  ...
</module>
```

The `moduleVersion` attribute shows the version of this module. In this example, it is `"1.2"`.

The `relaxCoreVersion` attribute shows the version of RELAX Core. At present, it is always `"1.0"`.

The `targetNamespace` attribute shows the namespace which this module is concerned with. In this example, it is `""`.

The namespace name for RELAX Core is `"http://www.xml.gr.jp/xmlns/relaxCore"`.

2.3 The `interface` element

A `module` element begins with an `interface` element. A module has one `interface` element.

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    ...
  </interface>
  ...
</module>
```

2.3.1 The `export` element

An `interface` element contains `export` element(s).

```
<export label="foo"/>
```

The `label` attribute of `export` elements specifies an element type that may become the root. More than one `export` may appear in an `interface` element.

The following example allows element type `foo` and `bar` as the root.

```
<interface>
  <export label="foo"/>
  <export label="bar"/>
</interface>
```

2.4 Element type declarations

Element type declarations (<!ELEMENT ...>) of XML are represented by `elementRule` elements. The `role` attribute of `elementRule` specifies an element type name. More than one `elementRule` may follow the `interface` element.

```
<elementRule role="element-type-name">
  ...hedge model...
</elementRule>
```

An `elementRule` element has a **hedge model**. A **hedge** is a sequence of elements (and their descendants) as well as character data. A hedge model is a constraint on permissible hedges.

A hedge model is either an element hedge model, datatype reference, or mixed hedge model.

2.4.1 Element hedge model

Element hedge modelsare represented by `empty, none, ref, choice, sequence` elements and the `occurs` attribute. An element hedge model represents permissible sequences of child elements, which are possibly intervened by whitespace characters.

2.4.2 The `empty` element

`empty`represents the empty sequence.

Consider an `elementRule` as below:

```
<elementRule role="foo">
  <empty/>
</elementRule>
```

This `elementRule` implies that the content of a `foo` element is the empty sequence. A `foo` element can be a start tag followed by an end tag, or an empty-element tag.

```
<foo/>
```

```
<foo></foo>
```

Unlike `EMPTY` of XML, whitespace characters **may** intervene between start and end tags.

```
<foo>  </foo>
```

`empty`can be used within `choice` and `sequence`. The motivation behind this extension will become clear in STEP 2[1]. If you need exactly the same feature as `EMPTY` of XML, use the `emptyString` datatype (shown in STEP 3[2]).

From now on, we assume that `foo, foo1,` and `foo2` are declared by `elementRule`s whose hedge models are `empty`.

2.4.3 The `ref` element

`ref`references to an element type. For example, <`ref label="foo"/`> references to an element type `foo`.

Consider an `elementRule` as below:

```
<elementRule role="bar">
  <ref label="foo"/>
</elementRule>
```

[1] ./step2e.html
[2] ./step3e.html

This `elementRule` implies that the content of a `bar` element is a `foo` element. For example, the next `bar` element is legitimate against this `elementRule`.

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

Whitespace may appear before and after the `foo` element.

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

`ref`can have the `occurs` attribute. Permissible values are "*", "+", and "?" , which indicate "zero or more", "one or more", and "zero or one time", respectively.

An example of "?" as the `occurs` attribute is as below:

```
<elementRule role="bar">
  <ref label="foo" occurs="?"/>
</elementRule>
```

This `elementRule` implies that the content of a `bar` element is either a `foo` or empty.

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

```
<bar></bar>
```

Whitespace characters may appear before and after the `foo` element. Even when this `bar` is empty, it may have whitespace characters.

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

```
<bar>
</bar>
```

2.4.4 The `choice` element

`choice`indicates a choice of the specified hedge models ("|" of XML 1.0). Subordinate elements of `choice` elements are element hedge models. `choice` can also have the `occurs` attribute.

An example of `elementRule` containing `choice` is shown below:

```
<elementRule role="bar">
  <choice occurs="+">
    <ref label="foo1"/>
    <ref label="foo2"/>
  </choice>
</elementRule>
```

This `elementRule` indicates that the content of a `bar` element is one or more occurrences of either `foo1` or `foo2` elements.

```
<bar><foo2/></bar>
```

```
<bar>
  <foo2/>
</bar>
```

```
<bar>
  <foo1/>
  <foo2/>
  <foo1/>
</bar>
```

2.4.5 The `sequence` element

`sequence`is a sequence of the specified hedge models. (",") of XML 1.0). Subordinate elements of `sequence` are element hedge models. `sequence` can also have the `occurs` attribute.

An example of `elementRule` containing `sequence` is shown below:

```
<elementRule role="bar">
  <sequence occurs="?">
    <ref label="foo1"/>
    <ref label="foo2"/>
  </sequence>
</elementRule>
```

This `elementRule` implies that the content of a `bar` element is either a sequence of a `foo1` element and a `foo2` element, or empty.

```
<bar><foo1/><foo2/></bar>
```

```
<bar>
  <foo1/>
  <foo2/></bar>
```

```
<bar/>
```

```
<bar></bar>
```

```
<bar>
  </bar>
```

2.4.6 The `none` element

`none`is an element hedge model, which does not match anything. `none` is unique to RELAX.

```
<elementRule role="bar">
  <none/>
</elementRule>
```

This `elementRule` implies that nothing is permitted as the content of `bar` elements. The motivation behind `none` will become clear in STEP 2[3].

[3] ./step2e.html

2.4.7 Datatype reference

The `type` attribute of `elementRule` allows a content model that references to a datatype. Character strings in a document are compared with the specified datatype. Permissible datatypes are built-in datatypes of XML Schema Part 2, or datatypes unique to RELAX. Details of datatypes will be covered by STEP 3[4].

An example of `elementRule` containing `type` is shown below:

```
<elementRule role="bar" type="integer"/>
```

This `elementRule` indicates that the content of a `bar` element is a character string representing an integer.

```
<bar>10</bar>
```

Whitespace characters may not occur before or after the integer. For example, the following is not permitted.

```
<bar>
  10
</bar>
```

2.4.8 Mixed hedge model

`mixed`significantly extends mixed content models (#PCDATA|a|b|...|z)* of XML.

A `mixed` element wraps an element hedge model. Recall that an element hedge model allows whitespace characters to intervene between elements. By wrapping it with `mixed`, **any** character is allowed to intervene.

`(#PCDATA | foo1| foo2)*`of XML can be captured as below:

```
<elementRule role="bar">
  <mixed>
    <choice occurs="*">
      <ref label="foo1"/>
      <ref label="foo2"/>
    </choice>
  </mixed>
</elementRule>
```

[4] ./step2e.html

The `choice` element in this `mixed` element matches zero or more occurrences of `foo1` or `foo2` elements. The `mixed` allows any character to intervene between these elements. Thus, this hedge model is equivalent to a `(#PCDATA | foo1| foo2)*` mixed content model of XML 1.0.

There are two ways to capture a `(#PCDATA)` content model. One is to reference to the datatype `string` by the `type` attribute. The other is to make an element hedge model that matches the empty sequence only and wrap it with `mixed`.

```
<elementRule role="bar" type="string"/>
```

```
<elementRule role="bar">
  <mixed>
    <empty/>
  </mixed>
</elementRule>
```

As a more advanced example, consider `elementRule` as below:

```
<elementRule role="bar">
  <mixed>
    <sequence>
      <ref label="foo1"/>
      <ref label="foo2"/>
    </sequence>
  </mixed>
</elementRule>
```

A sequence of <`foo/`> and <`foo2/`> matches `ref` in the `mixed` element. Thus, the following example is permitted by this `elementRule`.

```
<bar>Murata<foo1/>Makoto<foo2/>IUJ</bar>
```

As shown in the following example, CDATA sections and character references may appear.

```
<bar><![CDATA[Murata]]><foo1/>Mako&#x74;&#x6F;<foo2/>IUJ</bar>
```

2.5 Attribute-list declarations

Attribute-list declarations (<!ATTLIST ...>) of XML are captured by `tag` elements.

```
<tag name="element-type-name">
  ...list of attribute declarations...
</tag>
```

`tag`can have `attribute` elements as subordinates.

```
<tag name="element-type-name">
  <attribute ... />
  <attribute ... />
</tag>
```

`attribute`declares an attribute. An example of `attribute` is shown below:

```
<attribute name="age" required="true" type="integer"/>
```

The value of the `name` attribute is the name of the declared attribute. In this example, it is `age`.

If the value of the `required` attribute is `true`, the attribute being declared is mandatory. If `required` is not specified, it is optional. Since `required` is specified in this example, the `age` attribute is mandatory.

The `type` attribute specifies a datatype name. If `type` is not specified, a datatype `string` (which allows any string) is assumed.

Consider an example of `tag` which contains this `attribute` element only.

```
<tag name="bar">
  <attribute name="age" required="true" type="integer"/>
</tag>
```

The following start tag is permitted by this `tag`.

```
<bar age="39">
```

The following two start tags are not permitted. In the first example, the `age` attribute is omitted. In the second example, the value of `age` is not an integer.

```
<bar>
```

```
<bar age="bu huo">
<!-- "bu huo" means forty years in Chinese.  In Japan,
     it is pronounced as "FUWAKU". -->
```

In DTD, you do not have to write an attribute-list declaration if an element type does not have any attributes. In RELAX, you **must** write an empty `tag` element even if there are no attributes. For example, if an element type `bar` does not have any attributes, you have to write a `tag` element as below:

```
<tag name="bar"/>
```

2.6 Summary

If you have finished reading this STEP, you can immediately start to use RELAX. If you do not need further features, you do not have to read other STEPs. Enjoy and RELAX!

Chapter 3

STEP 2: Migration from XML DTD (with parameter entities)

$Id: step2.sdoc 1.10 2000/11/01 13:41:12 murata Exp $

Often, you have to write the same thing many times. Features in STEP 2 allow you to create a description once and reference to it repeatedly. These features mimic parameter entities of XML.

3.1 Parameter entities used in content models

`hedgeRule`allows you to write a hedge model once, name it, and reference to it repeatedly. In other words, `hedgeRule` mimics parameter entities referenced from content models in DTD.

3.1.1 Overview

The syntax of `hedgeRule` is shown below. `foo` is a name assigned to the hedge model of this `hedgeRule`.

```
<hedgeRule label="foo">
  ...element content model...
</hedgeRule>
```

To reference to such a `hedgeRule`, we write <`hedgeRef label="foo"`/>. This `hedgeRef` is replaced with the element hedge model specified in the `hedgeRule`.

In the following example, the hedge model of the `elementRule` for the element type `doc` references to a `hedgeRule`. This `elementRule` is borrowed from

the module in the beginning of STEP 1[1], and the hedge model minus `title` is rewritten by a `hedgeRule`.

```
<hedgeRule label="doc.body">
  <ref label="para" occurs="*"/>
</hedgeRule>

<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="doc.body"/>
  </sequence>
</elementRule>
```

The reference to `doc.body` is expanded as below:

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <ref label="para" occurs="*"/>
  </sequence>
</elementRule>
```

In this example, a `hedgeRule` is referenced from an `elementRule`. But a `hedgeRule` may reference to another `hedgeRule`.

3.1.2 Permissible hedge models

`hedgeRule`can have element hedge models only. Datatype references or mixed hedge models are not permitted. For example, the following rules are not permitted.

```
<hedgeRule label="mixed.param">
  <mixed>
    <choice occurs="*">
      <ref label="em"/>
      <ref label="strong"/>
    <choice>
  </mixed>
</hedgeRule>

<hedgeRule label="string.param" type="string"/>
```

[1] ./step1.html

If you want to use `hedgeRef` in conjunction with a mixed hedge model, you have to surround the `hedgeRef` with `mixed` in an `elementRule` element, rather than using the `mixed` element inside a `hedgeRule` element. An example is shown below. The mixed hedge model references to `phrase`, and `phrase` is described by a `hedgeRule`.

```
<hedgeRule label="phrase">
  <choice>
    <ref label="em"/>
    <ref label="strong"/>
  <choice>
</hedgeRule>

<elementRule role="p">
  <mixed>
    <hedgeRef label="phrase" occurs="*"/>
  </mixed>
</elementRule>
```

3.1.3 The `occurs` attribute

`hedgeRef`that references to a parameter entity can have `occurs`, and an element hedge model specified in `hedgeRule` can also have `occurs`. In the following example, both have `occurs`.

```
<hedgeRule label="bar">
  <sequence occurs="+" >
    <ref label="foo1"/>
    <ref label="foo2"/>
  </sequence>
</hedgeRule>

<elementRule role="foo">
  <hedgeRef label="bar" occurs="*"/>
</elementRule>
```

If this example is recaptured in DTD, expansion of the parameter entity `bar` is obvious.

```
<!ENTITY % bar "(foo1, foo2)+">
<!-- original --> <!ELEMENT foo (%bar;)*>
<!-- expanded --> <!ELEMENT foo ((foo1, foo2)+)*>
```

The following shows expansion of the above example. Observe that a `choice` element containing a single child is introduced during expansion. This `choice` element inherits `occurs="*"` from the `ref`.

```
<elementRule role="foo">
  <choice occurs="*">
    <sequence occurs="+" >
      <ref label="foo1"/>
      <ref label="foo2"/>
    </sequence>
  </choice>
</elementRule>
```

3.1.4 Occurrence order of `hedgeRef` and `hedgeRule`

Unlike parameter entities of DTD, `hedgeRule` does not have to precede `ref` that reference to it. For example, the following is not an error.

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="doc.body"/>
  </sequence>
</elementRule>

<hedgeRule label="doc.body">
  <ref label="para" occurs="*"/>
</hedgeRule>
```

3.1.5 Illegal reference to itself

`hedgeRule`may not reference to itself directly or indirectly. The follow is an error since the hedge model for `bar` references to `bar` itself.

```
<hedgeRule label="bar">
  <choice>
    <ref label="title"/>
    <hedgeRef label="bar" occurs="*"/>
  </choice>
</hedgeRule>
```

In the following example, the hedge model for `bar1` references to `bar2` and the hedge model for `bar2` references to `bar1`. Thus, there is an error.

```
<hedgeRule label="bar1">
  <hedgeRef label="bar2" occurs="*"/>
</hedgeRule>

<hedgeRule label="bar2">
  <choice>
    <ref label="title"/>
    <hedgeRef label="bar1"/>
  </choice>
</hedgeRule>
```

3.1.6 Use of `empty`

`empty`, shown in STEP 1[2], is typically used in `hedgeRule`. An example is as below:

```
<hedgeRule label="frontMatter">
  <empty/>
</hedgeRule>

<elementRule role="section">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="frontMatter"/>
    <ref label="para" occurs="*"/>
  </sequence>
</elementRule>
```

Users of this module can change the structure of `section` by customizing the description of `frontMatter`.

3.1.7 Use of `none`

`none`, shown in STEP 1[3], is also used in `hedgeRule`. An example is as below:

[2] ./step1.html
[3] ./step1.html

```
<hedgeRule label="local-block-class">
  <none/>
</hedgeRule>

<hedgeRule label="block-class">
  <choice>
    <ref label="para"/>
    <ref label="fig"/>
    <hedgeRef label="local-black-class"/>
  </choice>
</hedgeRule>
```

Users of this module can change the structure of `block-class` by customizing the description of `local-block-class`.

3.2 Parameter entities used in attribute-list declarations

`attPool`allows you to declare attributes once and reference to the declarations repeatedly. In other words, `attPool` mimics parameter entities referenced from attribute-list declarations.

3.2.1 Overview

The syntax of `attPool` is shown below. `foo` is a name of a parameter entity.

```
<attPool role="foo">
  ...attribute definitions...
</attPool>
```

To reference to such an `attPool`, we write <`ref role="foo"/`> before attribute declarations. This `ref` is replaced with attribute declarations specified in the `attPool`.

In the following example, a `tag` for the element type `title` references to `attPool`. This `tag` is borrowed from the module in the beginning of STEP 1[4] and rewritten. The `role` attribute, which is common to many element types, is described by `attPool` named `common.att`.

```
<attPool role="common.att">
  <attribute name="class" type="NMTOKEN"/>
</attPool>
```

[4] ./step1.html

```
<tag name="title">
  <ref role="common.att"/>
  <attribute name="number" required="true" type="integer"/>
</tag>
```

This `ref` is expanded as below:

```
<tag name="title">
  <attribute name="class" type="NMTOKEN"/>
  <attribute name="number" required="true" type="integer"/>
</tag>
```

In this example, `attPool` is referenced from `tag`, but it can also be referenced from `attPool`.

3.2.2 Occurrence order of `ref` and `attPool`

Unlike parameter entities of DTD, `attPool` does not have to precede `ref` that reference to it. For example, the following is not an error.

```
<tag name="title">
  <ref role="common.att"/>
  <attribute name="number" required="true" type="integer"/>
</tag>

<attPool role="common.att">
  <attribute name="role" type="NMTOKEN"/>
</attPool>
```

3.2.3 Multiple `ref` elements

A single `tag` or `attPool` may contain more than one `ref` element. In the following example, an `attPool` element references to more than one `ref` element. Required attributes are grouped as `common-req.att` and optional attributes are grouped as `common-opt.att`. These two are referenced from the `attPool` element for `common.att`.

```
<attPool role="common.att">
  <ref role="common-req.att"/>
  <ref role="common-opt.att"/>
</attPool>

<attPool role="common-req.att">
```

```
  <attribute name="role" type="NMTOKEN" required="true"/>
</attPool>

<attPool role="common-opt.att">
  <attribute name="id" type="NMTOKEN"/>
</attPool>
```

3.2.4 Illegal reference to itself

As in the case of `hedgeRule`, a direct or indirect reference to itself is an error. For example, the following is an error.

```
<attPool role="bar1">
  <ref role="bar2"/>
  <attribute name="id" type="NMTOKEN"/>
</attPool>

<attPool role="bar2">
  <ref role="bar1"/>
</attPool>
```

3.3 Summary

STEP 2 covers almost all features of XML DTD. Enjoy and RELAX!

Chapter 4

STEP 3: Datatypes

$Id: step3.sdoc 1.10 2000/08/26 03:14:38 murata Exp $

STEP 3 introduces datatypes.

4.1 Datatypes of XML Schema Part 2

XML Schema Part 2[1]introduces many built-in datatypes. They are designed so that other specifications can utilize them. RELAX borrows all these built-in datatypes.

Some of the built-in datatypes of XML Schema Part 2 are borrowed from XML DTD; the others are newly introduced. Those borrowed from XML DTD are shown as below:

- NMTOKEN
- NMTOKENS
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NOTATION

Next, built-in datatypes newly introduced by XML Schema Part 2 are shown below:

- string

[1]http://www.w3.org/TR/xmlschema-2/

- boolean
- float
- double
- decimal
- timeDuration
- recurringDuration
- binary
- uriReference
- QName
- language
- Name
- NCName
- integer
- nonPositiveInteger
- negativeInteger
- long
- int
- short
- byte
- nonNegativeInteger
- unsignedLong
- unsignedInt
- unsignedShort
- unsignedByte
- positiveInteger
- timeInstant
- time
- timePeriod

- date
- month
- year
- century
- recurringDate
- recurringDay

In XML Schema Part 2, when users reference to these built-in datatypes, users can further specify constraints such as value ranges. The same thing applies to RELAX. However, unlike XML Schema Part 2, RELAX does not allow users to define their own datatypes.

4.2 Datatypes unique to RELAX

Datatypes unique to RELAX are `none` and `emptyString`.

4.2.1 `none`

`none`is an empty datatype. No character strings belong to this datatype. RELAX uses `none` so as to prohibit attributes. In the following example, the `class` attribute is prohibited.

```
<tag name="p">
  <attribute name="class" type="none"/>
</tag>
```

Thus, the following start tag is not permitted.

```
  <p class="foo">
```

4.2.2 `emptyString`

`emptyString`is a datatype that allows the empty string only. This datatype is compatible with `EMPTY` of DTD.

```
<elementRule role="em" type="emptyString"/>
```

This `elementRule` allows the following two elements only. Whitespace characters may not occur between and .

```
<em/>
```

```
<em></em>
```

4.3 Additional constraints

Like XML Schema Part 2, RELAX allows users to specify additional constraints on datatypes. For example, users can specify `integer` and further specify a constraint "18 thru 65". The syntax for such additional constraints is the same as in XML Schema Part 2.

4.3.1 `elementRule`

To impose constraints on a datatype specified by `elementRule`, attach child elements to the `elementRule`.

In the following example, the hedge model for the element type `age` is a reference to `integer`. `minInclusive` and `maxInclusive` represent constraints on minimum and maximum values, respectively. Thus, permissible contents of `age` elements are character strings representing integers from 18 to 65.

```
<elementRule role="age" type="integer">
  <minInclusive value="18"/>
  <maxInclusive value="65"/>
</elementRule>
```

A `age` element can contain string "20" as its content.

```
<age>20</age>
```

But string "11" is not allowed.

```
<age>11</age>
```

4.3.2 attribute

To impose constraints on a datatype specified by `attribute`, attach child elements to `attribute`.

In the following example, the `sex` attribute of `employee` is constrained to be either `man` or `woman`. Here, `enumeration` is a constraint which specifies a permissible value.

```
<tag name="employee">
  <attribute name="sex" type="NMTOKEN">
    <enumeration value="man"/>
    <enumeration value="woman"/>
  </attribute>
</tag>
```

The `sex` attribute can have the string `"man"`.

```
<employee sex="man"/>
```

But it cannot contain the string `"foo"`.

```
<employee sex="foo"/>
```

4.4 Summary

STEP 3 provides more than enough features to play with. Enjoy and RELAX!

Chapter 5

STEP 4: Annotation

$Id: step4.sdoc 1.12 2000/11/01 13:43:29 murata Exp $

DTD documentation is highly important. Since DTDs merely define syntactical constructs, plenty of annotations in natural languages are required so as to explain the intended semantics of these constructs. Although XML comments are always available, such comments will be ignored by browsers which parse and then display RELAX modules.

STEP 4 provides features for annotating RELAX modules. Since they are represented by elements and attributes, browsers which parse RELAX modules can show annotations in a user-friendly manner.

5.1 The `annotation` element

The top-level element for annotations is the `annotation` element. `annotation` may occur in the following places.

- at most once before the `interface` element
- at most once in an `export` element
- at most once as the eldest child of each `elementRule`
- at most once as the eldest child of each `hedgeRule`
- at most once as the eldest child of each `tag`
- at most once as the eldest child of each `attPool`
- at most once as the eldest child of each `attribute`
- at most once as the child of each `include`
- at most once as the eldest child of each `element`
- at most once as the eldest child of each `div`

The `elementRule` element shown below has an annotation as its eldest child. The content of this annotation is omitted.

```
<elementRule role="para">
  <annotation> ... </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

Child elements of an `annotation` element are `documentation` elements and `appinfo` elements.

5.1.1 The `documentation` element

`documentation`is an element for representing explanations in natural languages. Since RELAX Namespace is not available yet, `documentation` may contain text data only.

The following shows a `documentation` element added to the above example.

```
<elementRule role="para">
  <annotation>
    <documentation>This is a paragraph.</documentation>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

If a `documentation` element has the `source` attribute, the attribute value is a URI that references to an explanation. In this case, the content of `documentation` is not used. Browsers for modules typically use this URI to provide a link.

```
<elementRule role="para">
  <annotation>
    <documentation source="http://www.xml.gr.jp/relax/"/>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

If a `documentation` element has the `xml:lang` attribute, the attribute value announces the natural language in which the content of the `documentation` is written.

In the next example, "en" is specified as the value of `xml:lang`.

```
<elementRule role="para">
  <annotation>
    <documentation xml:lang="en">This is a paragraph.</documentation>
  </annotation>
  <mixed>
    <ref label="fnote" occurs="*"/>
  </mixed>
</elementRule>
```

5.1.2 The `appinfo` element

Other than verifiers, which examine documents against RELAX modules, many programs might handle RELAX modules. For example, some programs may create a database schema from a module. `appinfo` provides hidden information for such programs. Since RELAX Namespace is not available yet, `appinfo` may contain text data only.

```
<elementRule role="foo" type="integer">
  <annotation><appinfo>default:1</appinfo></annotation>
</elementRule>
```

If an `appinfo` element has the `source` attribute, the attribute value is a URI that references to hidden information. In this case, the content of `appinfo` is not used.

5.2 The `div` element

We often would like to annotate a collection of `elementRule`s, `hedgeRule`s, `tag`s, and `attPool`s. The `div` element allows such an annotated group.

`div`elements may occur in `module` elements as siblings of `elementRule`s, `hedgeRule`s, `tag`s, and `attPool`s. `div` elements may further contain `div` elements. A `div` element may contain `elementRule`s, `hedgeRule`s, `tag`s, `attPool`s, and `div`s.

The following is a module shown in STEP 1. It is annotated after introducing `div` elements.

```
<module
      moduleVersion="1.2"
      relaxCoreVersion="1.0"
      targetNamespace=""
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
```

```
  </interface>

  <div>
    <annotation>
      <documentation>The root node</documentation>
    </annotation>

    <elementRule role="doc">
      <sequence>
        <ref label="title"/>
        <ref label="para" occurs="*"/>
      </sequence>
    </elementRule>

    <tag name="doc"/>

  </div>

  <div>
    <annotation>
      <documentation>Paragraphs</documentation>
    </annotation>

    <elementRule role="para">
      <mixed>
        <ref label="em" occurs="*"/>
      </mixed>
    </elementRule>

    <tag name="para">
      <attribute name="class" type="NMTOKEN"/>
    </tag>

  </div>

  <elementRule role="title">
    <mixed>
      <ref label="em" occurs="*"/>
    </mixed>
  </elementRule>

  <tag name="title">
    <attribute name="class" type="NMTOKEN"/>
    <attribute name="number" required="true" type="integer"/>
  </tag>

  <elementRule role="em" type="string"/>
  <tag name="em"/>

</module>
```

5.3 Summary

STEP 4 makes it easy to document your module. Enjoy and RELAX!

Chapter 6

STEP 5: Dividing large modules

$Id: step5.sdoc 1.7 2000/04/14 12:40:02 murata Exp $

Large modules are hard to maintain. STEP 5 introduces a mechanism for dividing a module into several pieces, which can be maintained easier.

6.1 Why divide modules?

Suppose that we rewrite a DTD containing 200 element types in RELAX. This size is fairly large, but is not uncommon. For each element type, RELAX needs an `elementRule` and a `tag`. If each `elementRule` and `tag` requires three lines, the total is 1200 lines. If we write extensive documentation, the total may become 3000 lines or even more. This size is too large to put in a single file.

Even DTD provides external parameter entities so as to divide large DTDs into modules and maintain each module independently. RELAX strongly requires some mechanism for dividing large modules.

6.2 The `include` element

In RELAX, a module can reference to another module by the `include` element. The `include` element is replaced with the content of the referenced module.

Let us examine an example of `include`. First, a module to be included is as below:

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">
```

```
  <interface/>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

This module contains an `elementRule` and `tag` for the element type `bar`. The `interface` element is empty. Suppose that this module is stored in `bar.rlx`.

Next, a module which references to and includes this module is as below:

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="foo"/>
  </interface>

  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <include moduleLocation="bar.rlx" />

</module>
```

This module contains an `elementRule` and `tag` for the element type `foo`. The `include` at the end of this this module references to `bar.rlx` via the `moduleLocation` attribute.

The `include` element is replaced by the body of the referenced module, which the content of the `module` element except the `interface` element. In this example, replacement is done as below:

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">
```

```
  <interface>
    <export label="foo"/>
  </interface>

  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

6.3 Non-empty `interface` elements

In the above example, the `interface` element of the referenced module is empty. Suppose that an `export` element is supplied in the `interface` element.

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="bar"/>
  </interface>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

In this case, the children of the `interface` element in the referenced module are attached to the `interface` element in the referencing module. In this example, the result of replacement is as below:

```
<module
    moduleVersion="1.2"
    relaxCoreVersion="1.0"
    targetNamespace=""
```

```
    xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="foo"/>
    <export label="bar"/>
  </interface>

  <elementRule role="foo">
    <ref label="bar"/>
  </elementRule>

  <tag name="foo"/>

  <elementRule role="bar" type="emptyString"/>

  <tag name="bar"/>

</module>
```

6.4 Summary

STEP 5 makes it easy to maintain large modules. Enjoy and RELAX!

Chapter 7

STEP 6: Default values, entities, and notations

$Id: step6.sdoc 1.9 2000/11/01 13:45:32 murata Exp $

Among the features of DTD, we have not covered default values, entities, and notations. STEP 6 is concerned with them.

7.1 Reasons that RELAX does not handle them

RELAX does not handle default values, entities, and notations. They are intentionally omitted from RELAX so that we can continue to use existing XML processors.

Suppose that RELAX introduced constructs for these features. For example, assume that RELAX had the `default` attribute which provides the default value of an attribute. Existing XML processors will *not* examine RELAX modules when they parse XML documents. Thus, they will not use `default`. The same thing applies to entities and notations: even if RELAX had constructs for declaring entities and notations, existing XML processors would not use them.

If we would like to introduce such features to RELAX, the only solution is to create RELAX-specific XML parsers. Those users who create and verify XML documents against RELAX grammars would certainly have to use such RELAX-specific XML parsers. Furthermore, those users who *receive* such XML documents would have to switch to RELAX-specific XML parsers. In our opinion, this is not very realistic.

7.2 Using DTD and RELAX together

Then, are we unable to use default values, entities, and notations? No, we can use these features if we use DTD and RELAX together.

The following is an XML document containing a DTD.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE person [
<!ATTLIST person
        bloodType CDATA "A">
]>
<person/>
```

This document is verified against a RELAX module as below:

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="person"/>
  </interface>

  <elementRule role="person">
    <empty/>
  </elementRule>

  <tag name="person">
    <attribute name="bloodType">
      <enumeration value="O"/>
      <enumeration value="A"/>
      <enumeration value="B"/>
      <enumeration value="AB"/>
    </attribute>
  </tag>
</module>
```

In this example, the DTD specifies the default value "A". XML processors do use this default. We can verify this XML document against the RELAX module without any problems. Verification is done as if "A" was specified as the attribute value.

Similarly, entities and notations can be described in DTD. First, we show an example of parsed entities.

```
<?xml version="1.0"?>
<!DOCTYPE doc [
<!ENTITY foo "This is a pen">
]>
<doc>
  <para>&foo;</para>
</doc>
```

This document is legitimate against the RELAX module as below:

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc">
    <ref label="para" occurs="*"/>
  </elementRule>

  <elementRule role="para" type="string"/>

  <tag name="doc"/>

  <tag name="para"/>

</module>
```

Next, we show an example of unparsed entities and notations.

```
<?xml version="1.0"?>
<!DOCTYPE doc [

<!NOTATION eps          PUBLIC
"-//ISBN 0-7923-9432-1::Graphic Notation//NOTATION Adobe Systems
Encapsulated Postscript//EN">

<!ENTITY logo_eps SYSTEM "logo.eps" NDATA eps>

<!ELEMENT doc EMPTY>

<!ATTLIST doc logo ENTITY #IMPLIED>
]>
<doc logo="logo_eps"/>
```

This document is legitimate against the following RELAX module.

```
<module
      moduleVersion="1.0"
      relaxCoreVersion="1.0"
      xmlns="http://www.xml.gr.jp/xmlns/relaxCore">

  <interface>
    <export label="doc"/>
  </interface>

  <elementRule role="doc" type="emptyString"/>

  <tag name="doc">
    <attribute name="logo" type="ENTITY"/>
  </tag>

</module>
```

7.3 Better leave them out

As we have seen in the previous section, we *can* use default values, entities, and notations by using DTD and RELAX together. Their use is *not* recommended, however.

Default values can be mimicked by application programs. We only have to hardcode "default values" in application programs and use them when attributes are absent. We can also write XSLT scripts so as to embed "default values" when attributes are absent.

Use links (especially, XLink) rather than external parsed entities or external unparsed entities. Links are much more appropriate for the WWW.

Internal parsed entities can be used without any problems, however. Some text data such as "<" can be best represented by internal parsed entities (e.g., <).

Unfortunately, default values, entities, and notations in DTD are not always processed as expected by casual users. This is because some XML processors do not fetch external DTD subsets or external parameter entities. However, all examples in this STEP use *internal* DTD subsets and thus are free from such unexpected results.

7.4 Summary

STEPs 1 thru 6 provide more than enough features for the migration from DTD to RELAX. As long as we use these features only, we can convert RELAX to DTD and vice versa without loss of information except for datatypes and facets. In the future, conversion between XML Schema should be possible. Enjoy and RELAX!

Chapter 8

STEP 7: `elementRule` and `hedgeRule`, revisited

$Id: step7.sdoc 1.13 2000/08/26 08:37:11 murata Exp $

Until this step, we have covered those features of `elementRule` and `hedgeRule` which can be easily understood by DTD authors. Actually, RELAX has a much more generalized framework.

8.1 `elementRule` and labels

An `elementRule` element can have the `label` **attribute**. We first consider underlying requirements and then introduce this attribute.

8.1.1 Context-sensitive content models

We often would like to attach different content models to the same tag name, depending on the context. As an example, consider paragraphs in sections and those in tables. Permissible subordinates of these two types of paragraphs are slightly different; we might want to allow paragraphs in sections to contain footnotes, but might want to allow those in tables to contain text only.

```
<!-- This example is legal. -->
<section>
  <para>This paragraph can contain footnotes <footnote>This
        is a footnote</footnote>.</para>
</section>
```

```
<!-- This example is illegal. -->
<table>
  ...
  <para>This paragraph cannot contain a footnote <footnote>This
        is an illegal footnote</footnote>.</para>
  ...
</table>
```

Thus, we would like to switch content models (shown below) depending on whether paragraphs occur in sections or tables.

```
<!-- Case 1: subordinate to <section> elements. -->
<!ELEMENT para (#PCDATA|footnote)*>

<!-- Case 2: subordinate to <table> elements. -->
<!ELEMENT para (#PCDATA)>
```

A good motivation for context-sensitive content models can be found in HTML. In HTML, an `a` element may not occur as a direct or indirect subordinate of another `a` element. The same situation is true of the `form` element, as well; a `form` element may not occur inside another `form` element.

```
<!-- This example is illegal. -->
<a href="foo"><span><a href="bar">dmy</a></span></a>
```

```
<!-- This example is also illegal. -->
<form>
  ...
  <div>
    <form>
      ...
    </form>
  </div>
  ...
</form>
```

In HTML, `a` elements can contain `span` elements. Since we would like to prohibit even indirect nesting, we have to allow those `span` elements outside of `a` elements to contain `a` elements and do not allow those in `a` elements to contain `span` elements. The same thing applies to `form`; we have to allow those `div` elements outside of `form` elements to contain `form` elements and do not allow those in `form` elements to contain `form` elements.

```
<!-- Case 1: subordinate to <a> elements. -->
<!ELEMENT span (#PCDATA|a)*>

<!-- Case 2: not subordinate to <a> elements. -->
<!ELEMENT span (#PCDATA)>
```

However, the DTD formalism allows only one content model per tag name. Thus, we cannot use different content models for paragraphs in different contexts. `span` may have only one content model; we cannot switch content models depending on whether the `span` element appears in some `a` element. The same thing applies to `div`.

Historically, two approaches have been used to overcome this problem. One is to introduce different tag names for different contexts. The following example illustrates this approach. Paragraphs in sections have the tag name `paraInSection`, and those in tables have the tag name `paraInTable`.

```
<!ELEMENT paraInSection (#PCDATA|footnote)*>

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

```
<!-- This example is legal. -->
<section>
  <paraInSection>This paragraph can contain footnotes <footnote>This
        is a footnote</footnote>.</paraInSection>
</section>
```

```
<table>
  ...
  <paraInTable>This paragraph cannot contain a footnote.</paraInTable>
  ...
</table>
```

This approach causes a flood of similar tag names: we have to duplicate tag name sets for common constructs such as paragraphs, footnotes, itemized lists, etc.

Instead of introducing more than one tag name for each construct, another approach creates a loose content model by merging different content models for different contexts. The following example illustrates this approach. Not only paragraphs in sections but those in tables are allowed to contain subordinate footnotes.

```
<!ELEMENT para (#PCDATA|footnote)*>
```

This approach causes loose validation. The following example validates against the above example.

```
<!-- This example is illegal. -->
<table>
  ...
  <para>This paragraph cannot contain a footnote <footnote>This
        is an illegal footnote</footnote>.</para>
  ...
</table>
```

8.1.2 The `label` attribute of `elementRule` elements

For a single tag name to have different content models depending on contexts, RELAX introduces **labels**. A single tag name associated with different labels can have different hedge models.

An `elementRule` can have the `label` attribute. A form of `elementRule` is as below:

```
<elementRule role="name" label="label">
  ...content model...
</elementRule>
```

If the `label` attribute is omitted, the value of the `role` attribute is used. Thus, the following `elementRule`s are equivalent.

```
<elementRule role="foo">
  ...content model...
</elementRule>
```

```
<elementRule role="foo" label="foo">
  ...content model...
</elementRule>
```

For paragraphs containing footnotes and paragraphs not containing footnotes, the following example uses different labels and thus different content models.

```
<elementRule role="para" label="paraWithFNotes">
  <mixed>
    <ref label="footnote" occurs="*"/>
  </mixed>
</elementRule>

<elementRule role="para" label="paraWithoutFNotes">
  <mixed>
    <empty/>
  <mixed/>
</elementRule>

<tag name="para"/>
```

The first `elementRule` show that paragraphs of the `paraWithFNotes` label contain text and footnotes. The second `elementRule` show that paragraphs of the `paraWithoutFNotes` label contain text only.

In most cases, there is one to one correspondence between labels and tag names. In fact, in all examples until this STEP, a tag name has only one associated label. To address issues presented in the previous subsection, we have to associate more than one label with a single tag name.

8.1.3 The `label` attribute of `ref` elements

Next, we revisit the `label` attribute of `ref` elements. Values of this attribute are always labels. STEP 1 explained that values are element type names, but that explanation is a white lie. RELAX does not have element types. (To tell the truth, XML 1.0 does not define element types. Element type declarations are defined, but element types are never defined.)

Since `paraWithFNotes` and `paraWithoutFNotes` in the last example in the previous section are labels, they can be referenced by `ref` elements. Content models for sections reference to `paraWithFNotes`, while those for tables (to be precise, table cells) reference to `paraWithoutFNotes`.

```
<elementRule role="section">
  <ref label="paraWithFNotes" occurs="*"/>
</elementRule>

<elementRule role="cell">
  <ref label="paraWithoutFNotes" occurs="*"/>
</elementRule>
```

8.2 Sharing labels

8.2.1 Multiple `hedgeRule` elements sharing the same label

More than one `hedgeRule` can specify the same label for the `label` attribute. In the following example, there are two `hedgeRule`s for the `blockElem` label.

```
<hedgeRule label="blockElem">
  <ref label="para"/>
</hedgeRule>

<hedgeRule label="blockElem">
  <ref label="itemizedList"/>
</hedgeRule>
```

The following `elementRule` references to this `blockElem`.

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <hedgeRef label="blockElem" occurs="*"/>
  </sequence>
</elementRule>
```

On validation against RELAX grammars, `hedgeRef` are first expanded. We use this `hedgeRef` as an example to demonstrate such expansion.

Both of the `hedgeRule`s describing the `blockElem` label have `ref` elements as hedge models. By grouping them with a `choice` element, we have the following.

```
<choice>
  <ref label="para"/>
  <ref label="itemizedList"/>
</choice>
```

The `hedgeRef` we intend to expand specifies `*` as the `occurs` attribute. We copy this attribute to the `choice` element.

```
<choice occurs="*">
  <ref label="para"/>
  <ref label="itemizedList"/>
</choice>
```

Finally, we replace the `hedgeRef` with this `choice` element.

```
<elementRule role="doc">
  <sequence>
    <ref label="title"/>
    <choice occurs="*">
      <ref label="para"/>
      <ref label="itemizedList"/>
    </choice>
  </sequence>
</elementRule>
```

Let us summarize the procedure for expanding a `hedgeRef` element referencing to some label.

1. Locate all `hedgeRule`s for this label.
2. Group hedge models of these `hedgeRule`s with a `choice` element.
3. Copy the `occurs` attribute of the `hedgeRef` to this `choice` element.
4. Replace the `hedgeRef` with this `choice` element.

Since multiple `hedgeRule`s are allowed to share a label, we are not forced to write a single `hedgeRule`. For example, if we would like to add `numberedItemizedList` as another sort of `blockElem`, we only have to add the following `hedgeRule`; we do not have to modify other `hedgeRule`s.

```
<hedgeRule label="blockElem">
  <ref label="numberedItemizedList"/>
</hedgeRule>
```

8.2.2 Prohibition of label sharing by `hedgeRule` and `elementRule`

`hedgeRule`and `elementRule` are prohibited from sharing a label. The following example is a syntax error.

```
<hedgeRule label="foo">
  <ref label="bar"/>
</hedgeRule>

<elementRule role="foo" label="foo">
  <empty/>
</elementRule>
```

8.2.3 Multiple `elementRule` elements sharing the same label

Multiple `elementRule`s can specify the same label for the `label` attribute. Moreover, the attribute `role` of multiple `elementRule`s may be identical.

In the following example, two `elementRule`s specify `section` as the value of the `role` attribute. Since neither specify the `label` attribute, `section` is assumed as the value of this attribute.

```
<tag name="section"/>

<elementRule role="section">
  <ref label="para" occurs="*"/>
</elementRule>

<elementRule role="section">
  <choice occurs="*">
    <ref label="para"/>
    <ref label="fig"/>
  </choice>
</elementRule>
```

In the case that multiple `elementRule`s exist for a single label, at least one of them are required to hold.

Consider the following `section` element. The first `elementRule` and the second `elementRule` holds for this element. Thus, we can attach the `section` label.

```
<section><para/></section>
```

The following `section` element contains a `fig` element, and thus only the second `elementRule` holds. Since one holding `elementRule` is sufficient, we can again attach the `section` label to this element.

```
<section><para/><fig/><para/></section>
```

Let us consider advantages of allowing more than one `elementRule` for a single label. Suppose that we already have a module and that we are going to modify this module so that more documents become legitimate.

In the traditional approach, we have to modify an existing `elementRule`. We cannot guarantee that what was legitimate is still legitimate after such modification.

In RELAX, we do not have to revise existing `elementRule`s, but we only have to add more `elementRule`s. In this approach, what was legitimate is guaranteed to be legitimate.

In the previous example, the initial plan was to allow only `para`s as contents of `section`. The first `elementRule` was written for this purpose. Later, the second `elementRule` was added so as to allow `fig` as contents of `section`. Since the first `elementRule` is still active, none of the then-legitimate documents has become illegitimate.

8.3 Summary

If you have struggled to create large DTDs, STEP 7 would probably look attractive. Weak points of DTD can be easily addressed in RELAX. Enjoy and RELAX!

Chapter 9

STEP 8: `tag` and `attPool`, revisited

$Id: step8.sdoc 1.15 2000/08/26 14:03:04 murata Exp murata $

In STEP 2, `tag` was compared to an attribute-list declaration and `attPool` was compared to parameter entities describing attributes. Actually, RELAX has a much more generalized framework.

9.1 The `role` attribute of `tag` elements

On top of the `name` attribute, `tag` elements can have the `role` **attribute**. In this section, we first consider motivations for this extension, and then introduce this attribute.

9.1.1 Switching content models depending on attribute values

Often, we would like to attach different content models to the same tag name, depending on attribute values. For example, we might want to switch content models of `val` element, depending on the `type` attribute. If the attribute value is `integer`, the content model is a reference to the datatype `integer`. If it is `string`, the content model is a reference to the datatype `string`.

```
<!-- This is legal. -->
<val type="integer">10</val>

<!-- This is also legal. -->
<val type="string">foo bar</val>

<!-- This is illegal. -->
<val type="integer">foo bar</val>
```

Thus, we would like to switch content models (shown below) depending on whether the attribute value is `integer` or `string`.

```
<!-- Case 1: type="integer" -->
<elementRule role="val" type="integer"/>

<!-- Case 2: type="string" -->
<elementRule role="val" type="string"/>
```

However, as long as we use features covered in STEPs 0 thru 7, we have to attach content models to tag names. Attribute values are not taken into consideration. Thus, no matter what the value of the `type` attribute is, the same `elementRule` is used.

9.1.2 Constraints represented by `tag` elements

On top of the `name` attribute, `tag` elements can have the `role` **attribute**. `tag` elements take the following form. While the `name` attribute specifies tag names, the `role` attribute specifies **roles**.

```
<tag name="tag-name" role="role-name">
  ...
</tag>
```

A `tag` element attaches a role to a collection of constraints on tag names and attributes. When a start tag (or empty-element tag) satisfies these constraints, this tag plays the specified role.

For example, consider a `tag` element as below:

```
<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>
```

This `tag` element specifies that **the tag name be `val` and the `type` attribute have the value `integer`.** If a start tag (or empty-element tag) satisfies this constraint, this tag plays the `val-integer` role.

```
<val type="integer">
```

In the following `tag` element, the constraint on the `type` attribute is that **the attribute value be** `string` and the role name is `val-string`.

```
<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>
```

The following start tag does not play the `val-integer` role, but plays the `val-string` role.

```
<val type="string">
```

Attributes may occur even if they are not specified by `tag` elements. For example, the following start tag has an attribute `unknown`, which is not specified by the previous `tag` element. This start tag still plays the role `val-string`, but warning message will be issued.

```
<val type="string" unknown="">
```

How should we interpret those `tag` elements without the `role` attribute such as those in STEPs 1 thru STEP 7? When the `role` attribute is omitted, it is assumed to have the value of the `name` attribute. Thus, the following two `tag` elements are semantically identical.

```
<tag name="foo">
  <attribute name="bar" type="int"/>
</tag>

<tag name="foo" role="foo">
  <attribute name="bar" type="int"/>
</tag>
```

9.1.3 The `role` attribute of `elementRule` elements

The `role` attribute of `elementRule` elements do **not** specify tag names, but rather specifies roles. Thus, we can switch hedge models for the same tag name, depending on attribute values.

If we use roles `val-string` and `val-integer` shown in the previous example, we can have two `elementRule`s for start tags of the tag name `val`. An `elementRule` that references to the `val-string` role is concerned with start tags whose `type` attribute has the value `string`. An `elementRule` that references to the `val-integer` role is concerned with start tags whose `type` attribute has the value `integer`.

```
<!-- Case 1: type="integer" -->

<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>

<elementRule role="val-integer" label="val" type="integer"/>

<!-- Case 2: type="string" -->

<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>

<elementRule role="val-string" label="val" type="string"/>
```

Note that two `tag` elements specify the tag name `val` and the `type` attribute. In RELAX, `tag` elements are not declarations, which may appear once and only once, but rather constraints, which may appear more than once.

9.1.4 Prohibition of references by `ref` elements

Roles referenced by `ref` elements may not be described by `tag` elements. If they are described, they must be described by `attPool` elements.

In the next example, a `ref` element references to the `foo` role, which is described by a `tag` element. This example is thus a syntax error.

```
<tag name="foo"/>

<attPool role="bar">
  <ref role="foo"/>
</attPool>
```

9.1.5 The `none` datatype, revisited

STEP 3 introduced the `none` datatype. `none` is useful for switching content models depending on the presence or absence of an attribute.

For example, suppose that <`div class="sec"`> and <`div`> require different content models. A role for the former, say `divSec`, can be described as below:

```
<tag name="div" role="divSec">
  <attribute name="class" type="string">
    <enumeration value="sec"/>
  </attribute>
</tag>
```

How do we describe a role for <`div`>, say `divWithoutClass`? One might think that the following example would work.

```
<tag name="div" role="divWithoutClass"/>
```

However, this description allows `divWithoutClass` even for <`div class="sec"`>. Although the "undeclared attribute" message is issued, this start tag is assumed to play both roles. [1]

To explicitly disallow the `class` attribute, we have to use the `none` datatype and write as below:

```
<tag name="div" role="divWithoutClass">
  <attribute name="class" type="none"/>
</tag>
```

Since no character strings are permitted by the `none` datatype, any value specified for the `class` attribute will prevent the `divWithoutClass` role.

9.2 `attPool` elements

Unlike parameter entitles of DTDs, `attPool` elements are **not** expanded. `tag` elements and `attPool` elements are very similar and equally important in RELAX.

[1] RELAX allows tags to play roles even if they have undeclared attributes. There are two reasons for this design. First, traditional XML processors continue validation even if they encounter undeclared attributes. Second, HTML allows undeclared attributes.

9.2.1 Constraints represented by `attPool`

We have observed that a `tag` element attaches a role to a collection of constraints on tag names and attributes. The only difference between `attPool` and `tag` is that `attPool` elements do not contain constraints on tag names. In other words, an `attPool` element attaches a role to a collection of constraints on attributes.

Consider the following `attPool`.

```
<attPool role="info">
  <attribute name="class" required="true">
    <enumeration value="informative"/>
  </attribute>
</attPool>
```

This `attPool` element specifies that **the `class` attribute is specified, and its value is `"informative"`** and attaches the `info` role to this constraint. There are no constraints on tag names. Because of this `attPool`, the following empty-element tag plays the `info` role.

```
<some class="informative"/>
```

Just like `tag`, attributes **not** specified by `attPool` may occur. For example, the following start tag plays the `info` role.

```
<some class="informative" unknown=""/>
```

9.2.2 Prohibition of references by `elementRule` elements

Roles referenced by the `role` attribute of `elementRule` elements may not be described by `attPool` elements. If they are described, they must be described by `tag` elements.

The following `elementRule` describes the `info` role, which is described by an `attPool` element. Thus, this example is a syntax error.

```
<attPool role="info"/>
<elementRule role="info" label="informative" type="emptyString"/>
```

9.3 Prohibition of role sharing by multiple `tag` or `attPool` elements

Multiple `tag` elements cannot share a single role.

In the following example, two `tag` elements share the `bar` role. Thus, this example is a syntax error.

```
<tag name="foo1" role="bar">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo2" role="bar">
  <attribute name="b" type="string"/>
  ...
</tag>
```

In the next example, a role and tag name are both shared by two `tag` elements. This example is also a syntax error.

```
<tag name="foo" role="foo">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo" role="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

Even when the `role` attribute is omitted and the value of the `name` attribute is used, role sharing is prohibited. The two `tag` elements in the next example are identical to the two `tag` elements shown above. Thus, this example is also a syntax error.

```
<tag name="foo">
  <attribute name="a" type="string"/>
  ...
</tag>

<tag name="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

In the following example, two `attPool` elements share the `bar` role. Thus, this example is a syntax error.

```
<attPool role="bar">
  <attribute name="a" type="string"/>
  ...
</attPool>

<attPool role="bar">
  <attribute name="b" type="string"/>
  ...
</attPool>
```

In this last example, a `tag` element and an `attPool` element share the `bar` role. Thus, this example is also a syntax error.

```
<attPool role="bar">
  <attribute name="a" type="string"/>
  ...
</attPool>

<tag role="bar" name="foo">
  <attribute name="b" type="string"/>
  ...
</tag>
```

9.4 Summary

In STEPs 0 thru 7, we have assumed that a `tag` element **declares** a tag name and attributes. Actually, a `tag` element **attaches** a role to a collection of constraints on tag names and attributes. In examples in STEPs 1 thru 7, roles and tag names coincide, but they are not always identical. In most cases, there are one-to-one correspondences among labels, roles, and tag names. But this is not always the case.

The following table summarizes syntactical constructs that describe or reference to tag names, labels, or roles.

The following table summarizes whether tag names, labels, and roles occur in XML documents.

In traditional DTDs, it has been impossible to switch content models depending on attribute values, but RELAX has made it possible. The only required extension is the `role` attribute. This demonstrates simplicity and descriptive power of RELAX. Enjoy and RELAX!

Syntactical constructs	tag names, labels, or roles
The `role` attribute of `elementRule`	references to roles described by `tag`
The `label` attribute of `elementRule`	description of labels
The `label` attribute of `hedgeRule`	description of labels
The `label` attribute of `ref`	reference to labels described by `elementRule`
The `label` attribute of `hedgeRef`	reference to labels described by `hedgeRule`
The `name` attribute of `tag`	description of tag names
The `role` attribute of `tag`	description of roles
The `role` attribute of `attPool`	description of roles
The `role` attribute of `ref`	reference to roles described by `attPool`

Types of names	In XML instances	In RELAX modules
tag names	occur	occur as part of clauses
roles	do not occur	occur in clauses
labels	do not occur	occur in production rules

Chapter 10

STEP 9: Hedge content model `element`

RELAX allows `element` elements as permissible hedge models. They are mere syntax sugar, and are expanded as `ref`, `elementRule`, and `tag` elements. In this section, we show motivation behind `element` elements and then present the mechanism.

10.1 Simulating programming languages and database languages

RELAX is an extension of DTDs, and is based on a grammatical data model. This model is very different from data models of programming languages and database systems. On the other hand, RELAX should be able to mimic declarations in programming languages and schemata in database languages.

In programming languages, we declare variables and attach datatypes to them. In the next example, variables `x` and `y` are declared and a datatype `int` is attached to them.

```
public class Point {
    int x;
    int y;
}
```

When a variable `x` is declared in another class, it may have a different type. In the next example, a datatype `float` is attached to `x` of the class `Foo`.

```
public class Foo {
    float x;
}
```

10.2 The `element` element

The `element` element is an element hedge model that specifies both a variable name and type name. An `element` element always has the `name` **attribute** and `type` **attribute**. Furthermore, it may have the `occurs` attribute.

```
<element name="tag-name" type="datatype-name"/>
```

```
<element name="tag-name" type="datatype-name" occurs="*"/>
```

Use of `element` elements allows `tag` and `elementRule` elements such as below:

```
<tag name="Point"/>

<elementRule role="Point">
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

A `Point` such that `x=100` and `y=200` can be represented by an XML document as below:

```
<Point>
  <x>100</x>
  <y>200</y>
</Point>
```

10.3 Expansion to `ref`, `elementRule`, and `tag` elements

The `element` element is merely syntax sugar. Each `element` element in a hedge model is replaced by a `ref` element, while an `elementRule` element and `tag` element are generated.

The `elementRule` in the previous subsection is duplicated below. Two `element` elements in this example have the `type` attribute. Let us consider how these `element` elements are expanded.

```
<elementRule label="Point">
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

Each of the `element` elements is replaced by a `ref` element. Furthermore, an `elementRule` element and `tag` element are generated for each `element` element. As a hedge model, each `elementRule` has a reference to the datatype specified by the `type` attribute of the original `element` element.

```
<elementRule label="Point">
  <sequence>
    <ref label="Point$1"/>
    <ref label="Point$2"/>
  </sequence>
</elementRule>

<elementRule role="Point$1" label="Point$1" type="integer"/>
<tag role="Point$1" name="x"/>

<elementRule role="Point$2" label="Point$2" type="integer"/>
<tag role="Point$2" name="y"/>
```

When an `element` element has the `occurs` attribute, it is copied to the generated `ref` element. For example, suppose that the `element`s in the first `elementRule` specifies `occurs="?"` (see below).

```
<elementRule label="Point">
  <sequence>
    <element name="x" type="integer" occurs="?"/>
    <element name="y" type="integer" occurs="?"/>
  </sequence>
</elementRule>
```

The result of expansion is as below:

```
<elementRule label="Point">
  <sequence>
    <ref label="Point$1" occurs="?"/>
    <ref label="Point$2" occurs="?"/>
  </sequence>
</elementRule>

<elementRule role="Point$1" label="Point$1" type="integer"/>
<tag role="Point$1" name="x"/>

<elementRule role="Point$2" label="Point$2" type="integer"/>
<tag role="Point$2" name="y"/>
```

10.4 Expansion procedure

In this section, we summarize expansion of `element` elements.

10.4.1 Generating `ref` elements

A `ref` element is generated. As the value of its `label` attribute, we generate a label that does not conflict with any other label. If the `element` has the `occurs` attribute, it is copied to the generated `ref` element.

10.4.2 Generating `elementRule` elements

An `elementRule` element is generated. As the value of its `role` attribute, we generate a role that does not conflict with any other role. The value of the `label` attribute is the label generated together with the `ref` element. As the hedge model of this `elementRule`, the `type` attribute of the `element` element is copied.

10.4.3 Generating `tag` elements

A `tag` element is generated. Its `role` attribute specifies the role automatically generated together with the `elementRule`. The `name` attribute of the generated `tag` specifies the value of the `name` attribute of the original `element` element.

10.5 Summary

For users of programming languages and database languages, description by `element` elements probably look very natural and easy to understand. Enjoy and RELAX!

Chapter 11

STEP 10: `tag` embedded in `elementRule`

$Id: step10.sdoc 1.8 2000/11/01 13:46:38 murata Exp $

In this section, we consider embedding of `tag` elements in `elementRule` elements.

11.1 Describing attributes and hedge models together

In STEPs 0 thru 9, attributes and tag names are separated from hedge models. Attributes and tag names are described by `tag` and `attPool` elements, while hedge models are described by `elementRule` and `hedgeRule` elements. An `elementRule` references to a `tag` via a role, and the `tag` may in turn reference to `attPool` elements.

When an `elementRule` and a `tag` is so closely related, it may be convenient to merge them into a single element rather than separating them.

As an example of `elementRule`-`tag` separation, we duplicate an example in STEP 8 below.

```
<!-- Case 1: type="integer" -->

<tag name="val" role="val-integer">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>

<elementRule role="val-integer" label="val" type="integer"/>

<!-- Case 2: type="string" -->
```

```
<tag name="val" role="val-string">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>

<elementRule role="val-string" label="val" type="string"/>
```

Suppose that roles `val-integer` and `val-string` are referenced from these two `elementRule` elements only. Rather than introducing two names `val-integer` and `val-string` for referencing, authors might want to directly embed `tag` elements within `elementRule` elements.

```
<!-- Case 1: type="integer" -->

<elementRule label="val" type="integer">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="integer"/>
    </attribute>
  </tag>
</elementRule>

<!-- Case 2: type="string" -->

<elementRule label="val" type="string">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="string"/>
    </attribute>
  </tag>
</elementRule>
```

An advantage of this style is that roles do not need names. Before this rewrite, we needed names which are different from the tag names or labels. Omission of these names enhance readability.

Some people find it attractive to describe attributes and hedge models together. For example, points with the x-coordinate and y-coordinate can be represented in two alternative manners. The first example uses attributes, while the second uses elements. Their differences are minor and can be easily rewritten from each other.

```
<elementRule label="point" type="emptyString">
  <tag>
    <attribute name="x" type="integer"/>
```

```
    <attribute name="y" type="integer"/>
  </tag>
</elementRule>
```

```
<elementRule label="point">
  <tag/>
  <sequence>
    <element name="x" type="integer"/>
    <element name="y" type="integer"/>
  </sequence>
</elementRule>
```

An `elementRule` containing a `tag` may not have the `role` attribute. The `label` attribute is mandatory, instead.

An embedded `tag` may not have the `role` attribute. The `name` attribute is permitted, but it is not present in this example.

11.2 Handling of embedded `tag` elements

An embedded `tag` element is moved from the `elementRule` and placed as a sibling element. We show how the first example in this STEP is handled.

```
<elementRule label="val" type="integer">
  <tag>
    <attribute name="type" type="NMTOKEN" required="true">
      <enumeration value="integer"/>
    </attribute>
  </tag>
</elementRule>
```

First, we generate a role that does not conflict with any other role. In this example, we generate role `val$1`.

Next, we move the embedded `tag` element from the `elementRule` and place as a sibling element. We then add the `role` attribute and specify the generated role as the attribute value.

Only when this `tag` element does not have the `name` attribute, we introduce this attribute. As the attribute value, we use the value of the `label` attribute of the `elementRule` element. In this example, we specify `"val"` as the value of the `name` attribute.

Finally, we add the `role` attribute to the `elementRule` and specify the role generated above.

```
<elementRule label="val" type="integer" role="val$1">
</elementRule>

<tag name="val" role="val$1">
  <attribute name="type" type="NMTOKEN" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>
```

11.3 Summary

To describe elements and attributes together, embedded `tag` elements provides concise and comprehensible description. Enjoy and RELAX!