4 SCHEMA LANGUAGES

Objectives

In this chapter, you will learn:

- The purpose of using schemas
- The schema languages DTD, XML Schema, DSD2, and RELAX NG
- Regular expressions – a commonly used formalism in schema languages

4.1 XML Languages and Validation

As we have seen in the previous chapters, XML is by itself merely a standard notation for markup languages. Every XML document uses this notation and must be well-formed to be meaningful. An XML language, for example RecipeML, is a particular family of XML documents. Such a language is defined by (1) its *syntax*, which describes the vocabulary of markup tags that this language uses and constraints on their use, and (2) its *semantics*, which gives some sort of meaning to the constituents of each syntactically correct document. In this chapter, we focus on syntax.

Valid: syntactically correct with respect to a given schema.

A *schema* is a formal definition of the syntax of an XML-based language, that is, it defines a family of XML documents. A *schema language* is a formal language for expressing schemas. There exists a variety of schema languages, as we shall see later. Each schema language is implemented in the form of *schema processors*: these are tools that as input take an XML document X, called the *instance document*, and a schema S, which is written in that particular schema language, and then checks whether or not X is syntactically correct according to S. If this is the case, then we say that X is *valid* with respect to S. If X is invalid, then most schema processors produce useful diagnostic error messages.

As a side-effect, if the schema processor determines that X does in fact conform to S, then it may *normalize* X according to rules specified in S, as indicated in Figure 4.1. Depending

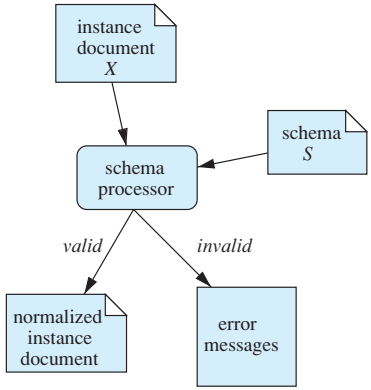

Figure 4.1 Schema processing.

on the choice of schema language, normalization may involve insertion of default attributes and elements, removal of insignificant whitespace, and addition of parsing information.

Schema: a formal definition of the syntax of an XML-based language.
XML Schema: a particular language for expressing schemas.

Schemas are in many ways similar to grammars for programming languages. For example, the syntax of Java is formally described by a context-free grammar using a so-called BNF notation. In order for a text to be a meaningful Java program, it must at the very least be syntactically correct with respect to this grammar. The BNF notation has for many years been successful for describing the syntax for programming languages. However, it turns out that BNF, or similar variants of context-free grammars, are not expressive enough for XML, which has led to the development of special grammar formalisms for XML. The term 'schema' comes from the database community. As we shall see in Chapter 6, there is a close connection between schemas in XML and schemas in relational databases. For readers that are familiar with grammars in programming languages, it is appropriate to think of schemas as grammars for XML languages. One final note about the term 'schema': in this book, we use 'schema' generally to denote a document from some schema language, not necessarily from the particular schema language named XML Schema. It has caused quite a lot of confusion that W3C chose the name 'XML Schema' for their particular XML schema language proposal.

In Section 2.7, we informally described the syntax of RecipeML. Why should one bother formalizing the syntax with a schema? There are a number of good reasons for this:

1. A formal definition can provide a precise but human-readable reference, which is a significant advantage when others are to read and write documents in our language. In contrast, informal definitions tend to be ambiguous, incomplete, or too verbose.
2. Each schema language comes with existing, often Open Source, implementations of schema processors. This means that once we have written a schema for our XML language, for example RecipeML, we can easily check validity whenever we write new RecipeML instance documents or edit existing ones.
3. Often, new XML languages are accompanied by specialized tools; for example, we could envision a tailor-made recipe editor with a nice graphical user interface for

RecipeML. Such tools always need to parse their input, and to be meaningful, the input must be valid. If the tool is presented with invalid data (for example, a required attribute is missing) it is not satisfactory that it produces null pointer exceptions or exhibits some arbitrary behavior. Instead, the tool should in this case abort with a useful error message. Rather than writing, by hand, some code in your favorite programming language checking validity of the input, a straightforward solution is to pipe the input through a preexisting schema processor such that syntax errors are caught before the data enters our tool. As an additional benefit, the schema processor will take care of normalizing the input for us.

A schema language must satisfy three criteria to be useful. First, it must provide sufficient *expressiveness* such that most syntactic requirements can be formalized in the language. Second, it must be possible to implement *efficient* schema processors. The time and space requirements for checking validity of a document should ideally be linear in the size of the document. Third, the language must be *comprehensible* by non-experts. It is fair enough that the schema authors need to master the schema language being used, but if others should be able to write valid instance documents, then they need to be able to understand the schema.

In the remainder of this chapter, we focus our attention on the two schema languages DTD and XML Schema, both originating from the W3C. However, we shall also investigate other proposals, which have emerged due to a number of problematic issues with the W3C proposals, especially pertaining to expressiveness and comprehensibility.

4.2 Regular Expressions

The notion of *regular expressions* is an important formalism that is used – in variants – in most schema languages to describe sequences of elements or characters. For example, one may wish to constrain certain attributes named `date` such that their values are dates of the form *dd-mm-yyyy*, that is, two digits for the day of month, followed by the number of the month and then the year, all separated by '-'. Or, one may wish to specify that the content of a `number` element must be an integer. As yet another example, in XHTML, the contents of a `table` element must be a sequence consisting of optionally one `caption` element followed by either a number of `col` or `colgroup` elements, which are optionally followed by a `thead` and then a `tfoot` element, and finally at least one either `tbody` or `tr` element.

Let Σ be an *alphabet* consisting of some set of atoms, which are typically Unicode characters or element names. A *regular expression* over Σ is an expression built from the following rules:

- each atom in Σ is by itself a regular expression; and
- if α and β are regular expressions, then the following are also regular expressions: α`?`, α`*`, α`+`, $\alpha\ \beta$, $\alpha \mid \beta$, and `(`α`)`.

The operators `?`, `*`, and `+` have higher precedence than concatenation (the empty operator, or juxtapositioning), which in turn has higher precedence than `|`. The parentheses can be used to override precedence by grouping subexpressions explicitly. For example, the expression `ab*|c`, where the alphabet contains `a`, `b`, and `c`, is interpreted as `(a(b*))|c` and *not* as `a(b*|c)` *nor* as `(ab)*|c`.

A finite string of atoms from Σ may or may not *match* a given regular expression α:

- an atom σ from Σ matches just the single atom σ;
- α`?` matches α optionally, that is, either the empty string or whatever α matches;
- α`*` matches zero or more repetitions of what α matches;
- α`+` matches one or more repetitions of what α matches;
- $\alpha\ \beta$ matches what α matches followed by what β matches;
- α`|`β matches the union of what α and β match; and
- `(`α`)` matches the same as α.

The regular expression `(a(b*))|c` thus matches all strings which either consist of one `a` followed by zero or more `b`'s or consist of a single `c`. A *regular language* is a set of strings that are matched by some regular expression. Besides their use in XML, regular languages are utilized in many other areas of computer science, ranging from text processing to formal verification of hardware and natural language processing.

As an example, we may define a regular expression named *`d`* (for 'digit') by

```
0|1|2|3|4|5|6|7|8|9
```

where our alphabet, for example, consists of all Unicode characters. From this definition, we can choose to define another regular expression *`date`* by the expression *`dd-dd-dddd`*. Of course, this expression matches strings, such as `87-13-2005`, that do not describe real dates, but with a slightly more complicated expression, it is, in fact, possible to capture the desired set of strings. (Even if we handle leap years properly this set is a regular language, but then the corresponding regular expression becomes quite complicated!)

As another example, the following regular expression describes integers (using an alphabet that contains `0`, `1`, . . . , `9` and `-`):

```
0|-?(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

According to this definition, `-42`, `0`, and `117` are permissible, but `000`, `-0`, and `3.14` are not.

To capture the constraint on the contents of `table` elements described above, let Σ denote the alphabet of element names in XHTML. The constraint now corresponds to the regular expression

```
caption? ( col | colgroup )* thead? tfoot? ( tbody | tr )+
```

Regular expressions are useful for describing valid attributes, character data, and element contents.

There exist many variants of regular expressions, most of which simply add syntactic sugar on top of the basic notation presented here. One common extension is *character ranges*: for example, the character range `[0-9]` is a convenient abbreviation for the basic regular expression `0|1|2|...|9`. Similarly, α{`n,m`}, where n and m are non-negative integers, often denotes from n to m repetitions of α. In the following, we will explain such variations whenever they arise.

4.3 DTD – Document Type Definition

XML has since the first working draft contained a built-in schema language: *Document Type Definition (DTD)*. Just as the XML notation itself is designed as a subset of SGML, the DTD part of XML is designed as a subset of the DTD part of SGML. When we refer to 'DTD', it will be the XML variant. Also, to avoid confusion, we will use the term 'DTD schema' for referring to a particular schema written in the DTD language. (Elsewhere, 'DTD' is often used as a noun referring to a particular DTD schema.)

DTD is a reasonably simple schema language with a rather restricted expressive power. From a language design point of view, it is not particularly elegant, but there exist thousands of DTD schemas so it is useful to know the DTD language. Also, it has provided the starting point for the development of newer and more expressive schema languages. In the following, note that DTD is *not* itself written in the XML notation (which is somewhat peculiar, since XML was designed to be suitable for all kinds of structured information). Another thing to note is that DTD does not support namespaces – this is not surprising since the DTD language was introduced before the namespace mechanism.

4.3.1 Document Type Declarations

An XML document may contain a *document type declaration*, which is essentially a reference to a DTD schema. By inserting such a declaration, the author states that the XML document is intended to be valid relative to that schema. In this way, documents become *self-describing*, which makes it easy for tools to determine what kind of input they receive. (Be aware of the difference between a document type *declaration* and document type *definition*.)

A document type declaration typically has the form

```
<!DOCTYPE root SYSTEM "URI">
```

where `root` is an element name, and `URI`, called the *system identifier*, is a URI of the DTD schema. The document type declaration appears between the XML declaration (`<?xml ...?>`), if present, and the root element in the instance document. The instance document is *valid* if the name of its root element is `root` and the document satisfies all constraints specified in the DTD schema as described below.

Assume that we have written a DTD schema for RecipeML and made it available at the URL `http://www.brics.dk/ixwt/recipes.dtd`. Our recipe collection could then look as follows (where the content of the root element is shown as '. . .'):

The **document type declaration** associates a DTD schema with the instance document.

```
<?xml version="1.1"?>
<!DOCTYPE collection
     SYSTEM "http://www.brics.dk/ixwt/recipes.dtd">
<collection>
  ...
</collection>
```

As a supplement to the system identifier, a document type declaration may contain a *public identifier*, which is an alternative way of specifying the DTD schema. For example, XHTML documents often contain the following declaration:

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Here, the string following `PUBLIC` is the public identifier, and the string on the next line is the system identifier. This specific public identifier is technically an unregistered identifier (that is, not centrally assigned, which is specified by the leading '-'), it is owned by the W3C, it refers to a DTD for the 'XHTML 1.0 Transitional' language, and it is written in English. The notion of public identifiers is a relic from SGML; usually, public identifiers are simply ignored. However, they do come in handy in situations where the DTD schema may exist at many different locations. For example, W3C's HTML/XHTML validator reads the public identifier to determine the version of HTML or XHTML being used; it cannot determine the version from the system identifier since it is not required that the schema resides at a fixed location. The URN mechanism (see Section 1.3) provides a more general solution to the issue of naming resources without specific locations, and some systems allow URNs as system identifiers.

One might encounter document type declarations of the form

```
<!DOCTYPE root [ ... ]>
```

where '. . .' consists of declarations of elements, attributes, and so on, as described below. Such *internal* declarations that appear within the instance document have exactly the same meaning as if they were moved to a separate file being referred to with a system identifier. The document type declarations may even contain a mix of internal and external declarations. However, most often it is a much better idea to keep the DTD schema separately from the instance documents, such that many instance documents can share the same schema.

DTD schemas may contain comments using the same notation as in XML:

```
<!-- this is a comment -->
```

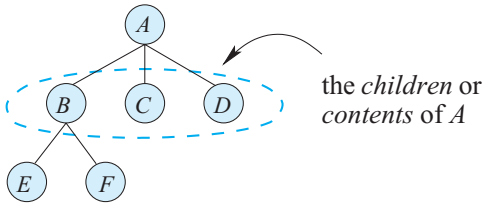

Figure 4.2 The *contents* of an element.

As usual, comments have no formal meaning, but they are often used in DTD schemas to explain extra restrictions that DTD is unable to express formally. For example, in the DTD schema for XHTML, one may find a comment like this:

```
<!-- anchors shouldn't be nested -->
```

Likewise, DTD schemas may contain processing instructions.

4.3.2 Element Declarations

A DTD schema consists of declarations of elements, attributes, and various other constructs, as explained in the following. An *element declaration* looks as follows:

```
<!ELEMENT element-name content-model>
```

where *element-name* is an element name, such as `table` or `ingredient`, and *content-model* is a description of a *content model*, which defines the validity requirements of the contents of all elements of the given name. (Recall that the *contents* of an element node is the sequence of its child nodes, which does not include nodes further down the tree; see Figure 4.2.) Every element name that occurs in the instance document must correspond to one element declaration in order for the document to be valid. Moreover, the contents of the element must match the associated content model as defined below.

Content models come in four different flavors:

empty: If the content model of an element is `EMPTY`, then the element must be empty. As previously explained, being empty means that it has no contents, but this says nothing about attributes.

any: The content model `ANY` means that the contents of the element can consist of any sequence of character data and elements. Still, each of these elements must be declared by corresponding element declarations (so using `ANY` is not a solution for describing open content models). The `ANY` content model is mostly used during development of DTD schemas for elements that have not been described yet – it is rarely used in the final schemas.

mixed content: A content model of the form

(#PCDATA | e_1 | e_2 | ... | e_n)*

where each e_i is an element name, means that the contents may contain arbitrary character data, interspersed with any number of elements of the specified names (n may be zero in which case the '*' is optional and only character data is permitted).

element content: To specify constraints on the order and number of occurrences of child elements, a content model can be written using the following variation of the regular expression notation presented in Section 4.2:

- the alphabet consists of all element names;
- concatenation is written with comma (,) instead of using juxtaposition of the operands (but the remaining operators, ?, *, +, and |, and also parentheses behave as explained earlier); and
- only a restricted form of regular expressions called *deterministic regular expressions* is permitted, which makes it easier to check whether or not a sequence of element names matches an expression. (We will not tire the reader with technicalities here and refer to the XML specification for details.)

For the contents of an element with such a content model to be valid, it must match the regular expression.

Element declarations associate content models with element names.

The various constructs are summarized in Figure 4.3. So far, we have not mentioned comments and processing instructions. Implicitly, comments and processing instructions may occur anywhere in the instance document; their use cannot be constrained, except with EMPTY since 'empty contents' means 'no elements, character data, comments, or processing instructions whatsoever'. (The fact that comments and processing instructions may appear in the DTD schema has nothing to do with this.)

At this point, we might wonder about the design of the DTD language. First, either arbitrary character data is permitted in the contents or no character data is permitted. Why are we not allowed to impose constraints on character data, for example, such that only

Construct	*Meaning*
EMPTY	empty contents
ANY	any contents
#PCDATA	character data
element name	an element
,	concatenation
|	union
?	optional
*	zero or more repetitions
+	one or more repetitions

Figure 4.3 Constructs used in content model descriptions in DTD.

whitespace and digits are allowed in the contents of certain elements? Second, if character data is to be permitted in the contents, then we have no choice but using the mixed content model, which cannot constrain the order and number of occurrences of the child elements. Why is it not possible to use character data together with the element content model? This question is related to a tradition of roughly classifying SGML and XML languages as either *document oriented* or *data oriented* depending on whether they use the mixed content model or not (as discussed in Section 2.5), but such an artificial classification should not be necessary. Third, why do we need the restriction to deterministic content models? After all, it has been known for more than thirty years how to perform efficient pattern matching on general regular expressions. Clearly, these limitations become practical problems in the real world. The simple answer to these questions is that a design goal was that XML (and hence also DTD) should be a subset of the SGML language, which has these unfortunate properties. From another point of view, the identification of these limitations, together with others that we mention later, has motivated the design of alternative schema languages.

Let us see some examples of element declarations. First, the declaration of `table` elements in the DTD for XHTML:

```
<!ELEMENT table
      (caption?, (col|colgroup)*, thead?, tfoot?, (tbody|tr)+) >
```

This declaration corresponds to the informal description given in Section 4.2. The following snippet is valid XHTML:

```
<table>
  <caption><em>Great Cities of the World</em></caption>
  <thead>
    <tr><td>City</td><td>Country</td></tr>
  </thead>
  <tbody>
    <tr><td>Aarhus</td><td>Denmark</td></tr>
    <tr><td>San Francisco</td><td>USA</td></tr>
    <tr><td>Paris</td><td>France</td></tr>
  </tbody>
</table>
```

whereas this one is not (although it is well-formed):

```
<table>
  <thead>
    <tr><td>City</td><td>Country</td></tr>
  </thead>
  <thead>
    <tr><td>City</td><td>Country</td></tr>
  </thead>
  <tBodY>
    <tr><td>Aarhus</td><td>Denmark</td></tr>
```

```
    <tr><td>San Francisco</td><td>USA</td></tr>
    <tr><td>Paris</td><td>France</td></tr>
  </tBodY>
  <caption><em>Great Cities of the World</em></caption>
</table>
```

because the `caption` element appears in the wrong place, there are two `thead` elements, and `tbody` is misspelled in the contents of `table`.

One more example: In Section 2.7, we informally stated that `comment` elements in RecipeML should contain just character data. This requirement can be formalized as follows:

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

This `comment` element is then valid:

```
<comment>
  Rhubarb Cobbler made with bananas as the main sweetener.
  It was delicious.
</comment>
```

but the following is invalid:

```
<comment><title>Rhubarb Cobbler</title> is delicious.</comment>
```

As mentioned, DTD does not support namespaces. Still, many XML languages that do use namespaces also have DTD descriptions, for example XHTML. How is this possible? Since DTD is not aware of the meaning of prefixes and namespace declarations, the DTD schema for XHTML simply assumes the empty prefix for all elements and declares that the `html` element must have an attribute named `xmlns` with the value `http://www.w3.org/1999/xhtml`. (Note that after the introduction of namespaces, this will technically not be an *attribute* but a *namespace declaration*.) The effect is that an XHTML instance document validates with respect to the DTD schema only if the document does not use the namespace mechanism beyond this, for example, by using an explicit namespace prefix. These issues with namespaces are, of course, something that newer schema languages have amended.

4.3.3 Attribute-List Declarations

An *attribute-list declaration* has the following form:

```
<!ATTLIST element-name attribute-definitions>
```

where *`element-name`* is an element name and *`attribute-definitions`* is a list of attribute definitions, each having three constituents:

```
attribute-name attribute-type default-declaration
```

The *attribute-name* is an attribute name, and *attribute-type* and *default-declaration* are an *attribute type* and a *default declaration*, respectively, as explained below. An attribute list declaration specifies a list of attributes that are permitted (or required) in elements of the given name. As for elements and element declarations, the instance document is invalid if it contains attributes that are not declared or do not satisfy their declarations.

As a simple example, the `align` attribute of `p` elements in XHTML is declared by

```
<!ATTLIST p align (left|center|right|justify) #IMPLIED>
```

Here, the attribute type specifies that the possible values of such attributes are `left`, `center`, `right`, and `justify`, and the default declaration `#IMPLIED` means that the attribute is optional and no default value is provided.

The most important categories of attribute types are the following:

string type: The attribute type `CDATA` (for *character data*) means that the attribute can have any value.

enumeration: An attribute type of the form

(s_1 | s_2 | ... | s_n)

where each s_i is some string, means that the value of the attribute must be among the strings $s_1, s_2, \ldots, s_n$.

name tokens: The attribute type `NMTOKEN` means that the attribute value must be a *name token*. In XML 1.1, almost all characters are permitted in name tokens, except those which are delimiters (for example, whitespace characters and commas); however, hyphens, underscores, colons, and periods are explicitly permitted. (This definition was more restrictive in XML 1.0.) The variant `NMTOKENS` denotes a whitespace separated nonempty list of name tokens.

identity/reference type: The attribute type `ID` means that the value of the attribute uniquely identifies the element containing the attribute. That is, no two attributes of type `ID` can have the same value in an XML document. Only one attribute of type `ID` is permitted per element.

The attribute type `IDREF` is used for references to elements with an `ID` attribute. The value of an attribute of type `IDREF` must match the value of some attribute of type `ID` in the same document. The attribute type `IDREFS` is like `IDREF` but permits multiple references (whitespace separated lists of names) as attribute values.

For all attribute types other than `CDATA`, the attribute value is – before the actual validation takes place – normalized by discarding any leading and trailing whitespace and replacing sequences of whitespace by a single space character. (The motivation for this whitespace normalization feature is not obvious; even the original designers of XML call it a design mistake.)

We have already seen an application of the enumeration type for the `align` attribute in XHTML. The `CDATA` type is, for example, used for both the `maxlength` and the `tabindex` attribute in `input` elements in XHTML:

```
<!ATTLIST input maxlength CDATA #IMPLIED
               tabindex  CDATA #IMPLIED>
```

Attribute-list declarations describe the attributes, and their types and default behavior.

These attributes are used to specify the maximal number of characters in a text input field and the field position in tabbing order, respectively. In both cases, only numbers make sense, but DTD does not have the ability to capture that requirement: the attribute type that comes closest permits any value. This is a typical example where a DTD schema 'permits too much' because a desired constraint is not expressible. Interestingly, the authors of the DTD schema for XHTML have inserted the informal comment

```
<!-- one or more digits -->
```

at that place in the schema as an attempt to make it clear to the human reader that not all values make sense. This example shows that an XML document may be valid XHTML with respect to the DTD schema without being syntactically correct in a strict sense; similar situations occur in most other XML languages.

The `NMTOKEN` type is, for example, used for certain `name` attributes in XHTML:

```
<!ATTLIST form name NMTOKEN #IMPLIED ...>
```

This declaration shows that `name="my.form"`, `name="87"`, and `name=" 87 "` are all valid in `form` elements (the latter because of the pruning of whitespace in non-`CDATA` attributes), whereas `name="my form"` and `name=""` are invalid.

The `ID` and `IDREF(S)` types can be used as a intra-document reference mechanism, for denoting keys when XML documents are used as databases, and for specifying anchors for easy addressing into documents. In each case, `ID` and `IDREF(S)` provide only very primitive support, however. Other technologies, such as XML Schema and XPath, are usually much better suited for these purposes. Nevertheless, we will use these types in our DTD description of RecipeML: the `id` attribute of `recipe` elements has type `ID`, and the `ref` attribute of `related` elements has type `IDREF`:

```
<!ATTLIST recipe id ID #IMPLIED>
<!ATTLIST related ref IDREF #REQUIRED>
```

The following RecipeML document is then valid (abbreviated with '...'):

```
<collection>
  <description>My Valid Recipe Collection</description>

  <recipe id="r101">
    <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
```

```
    ...
    <related ref="r103">
       this goes well with Linguine Pescadoro
    </related>
  </recipe>

  <recipe id="r102">
    <title>Ricotta Pie</title>
    ...
  </recipe>

  <recipe id="r103">
    <title>Linguine Pescadoro</title>
    ...
  </recipe>
</collection>
```

whereas the following is invalid because it contains two `ID` attributes with the value `r101` (both are named `id`) and also an `IDREF` attribute (named `ref`) whose value `r12345` does not match that of an `ID` attribute:

```
<collection>
  <description>My Invalid Recipe Collection</description>

  <recipe id="r101">
    <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
    ...
    <related ref="r12345">Spiced Beef Stew is also great</related>
  </recipe>

  <recipe id="r101">
    <title>Ricotta Pie</title>
    ...
  </recipe>

  <recipe id="r113">
    <title>Linguine Pescadoro</title>
    ...
  </recipe>
</collection>
```

One may think of this mechanism as going beyond purely tree-structured data: an `IDREF` attribute may be thought of as a special pointer from its containing element to the element with the corresponding `ID` attribute.

In addition to the main attribute types described above, the DTD language contains three rather obscure types: `ENTITY`, `ENTITIES`, and `NOTATION`. These are used in conjunction with *entity declarations* and *notation declarations* as described in the next section.

Construct	*Meaning*
`#REQUIRED`	required
`#IMPLIED`	optional, no default
`"value"`	optional, `value` is default value
`#FIXED "value"`	as the previous, but only this value is permitted

Figure 4.4 Default declarations for attributes in DTD.

The third constituent of an attribute declaration – the default declaration – specifies whether the attribute is required or optional and potentially also a default value. (It is misleading that this constituent is called a default declaration since it might not specify a default.) The following kinds are possible:

required: `#REQUIRED` means that the attribute must be present. (If the attribute is absent, then no default is provided and the document is invalid.)

optional: `#IMPLIED` means that the attribute is optional. No default is provided if the attribute is absent.

optional, but default provided: `"value"`, where `value` is a legal attribute value, means that the attribute is optional, but if it is absent, this value is used as a default.

fixed: `#FIXED "value"` means that the attribute is optional; if it is absent, then this value is used as a default, but if it is present, then it must have this specific value.

These variants are summarized in Figure 4.4. The following declaration from the DTD schema for XHTML illustrates the first three variants:

```
<!ATTLIST form
    action      CDATA          #REQUIRED
    onsubmit    CDATA          #IMPLIED
    method      (get|post)     "get"
    enctype     CDATA          "application/x-www-form-urlencoded">
```

This declaration shows that the `action` attribute in `form` elements is required, and the `onsubmit` attribute is optional. The `method` attribute is also optional, but if it is omitted, the default value `get` is inserted by the DTD processor, and similarly for the `enctype` attribute. (The actual attribute-list declaration for `form` contains other attributes, which we ignore here.) A DTD processor will then validate the following part of an instance document (we here concentrate on the attributes and ignore the contents of the element):

```
<form action="http://www.brics.dk/ixwt/examples/hello.jsp">
  ...
</form>
```

and normalize it as follows:

```
<form action="http://www.brics.dk/ixwt/examples/hello.jsp"
      method="get"
      enctype="application/x-www-form-urlencoded">
  ...
</form>
```

As usual, the ordering of attributes and whitespace between them are insignificant.

The 'fixed' variant seems strange. It certainly conveys little information to say that a certain attribute after normalization must have one specific value. A typical use of `#FIXED` is to fake namespace declarations, as in the DTD schema for XHTML:

```
<!ATTLIST html xmlns CDATA #FIXED "http://www.w3.org/1999/xhtml">
```

In Chapter 2, we mentioned how the special attribute named `xml:space` can be used to indicate significance of whitespace. When this attribute is used, it is often declared in the schema using `#FIXED` so that the instance document author does not have to worry about it:

```
<!ATTLIST pre xml:space (preserve) #FIXED "preserve">
```

However, irrespective of this attribute, the DTD processor never throws away any whitespace – this is left to the application.

When more than one attribute-list declaration is provided for a given element name, the declarations are merged. If an attribute of a given name is specified more than once for an element, then the first one takes effect, but friendly XML parsers issue a warning if this situation occurs.

4.3.4 Conditional Sections, Entity, and Notation Declarations

The features described in this section are not essential to the workings of XML and DTD, so you might skip this section in a first reading. In existing DTD schemas, you might encounter applications of them, but they are rarely indispensable when developing new schemas.

First, *internal entity declarations* constitute a poor man's macro mechanism. If our DTD schema contains

```
<!ENTITY copyrightnotice "Copyright &#169; 2005 Widgets'R'Us.">
```

then the DTD processor may convert the following fragment of an instance document

```
A gadget has a medium size head and a big gizmo subwidget.
&copyrightnotice;
```

into

```
A gadget has a medium size head and a big gizmo subwidget.
Copyright &#169; 2005 Widgets'R'Us.
```

If the copyright notice appears often, this mechanism allows us to save some space (which was a big concern when SGML was developed). The contents of such a declaration may consist of any well-formed XML data, including markup. The construct `&`*`entity`*`;`, where *`entity`* is the name of a previously declared entity, is called an *entity reference*. Some XML parsers that support validation with DTD actually perform the expansion of entity references, whereas others instead maintain special *entity reference nodes* in the XML tree representing the entity references together with pointers to their declarations.

The predefined entities mentioned in Section 2.4 are implicitly defined using internal entity declarations. We may also add another declaration:

```
<!ENTITY copy "&#169;">
```

so that we can write

```
&copy;
```

which is easier to remember than `&#169;` whenever we want a © symbol.

An *internal parameter entity declaration* is a macro definition that is only applicable within the DTD schema, and not in the instance document. For example, as in XHTML, we may define

```
<!ENTITY % Shape "(rect|circle|poly|default)">
```

(notice the `%` symbol) and then, within the schema, use references such as in

```
<!ATTLIST area shape %Shape; "rect">
```

which is then equivalent to

```
<!ATTLIST area shape (rect|circle|poly|default) "rect">
```

The `%` symbol in the declaration and the reference indicates a parameter entity, whereas `&` is used for normal entity references.

An *external entity declaration* is a reference to another resource, which consists of XML or non-XML data. A reference to another XML file may be declared by, for example:

```
<!ENTITY widgets SYSTEM "http://www.brics.dk/ixwt/widgets.xml">
```

(A public identifier, as described earlier, may also be used.) Occurrences of the entity reference `&widgets;` will then result in the designated XML data being inserted in place of the entity reference, perhaps via an entity reference node as explained above.

A reference to a non-XML resource, called an *unparsed entity*, can be declared as in

```
<!ENTITY widget-image
         SYSTEM "http://www.brics.dk/ixwt/widget.gif"
         NDATA gif>
```

Here, the NDATA part refers to a *notation*, which describes the format of an unparsed entity, for example by the declaration

```
<!NOTATION gif
    SYSTEM "http://www.iana.org/assignments/media-types/image/gif">
```

which could be used as a description of GIF images. Notation declarations may also be used for description of processing instruction targets.

Of course, it does not make sense for an instance document to contain entity references to unparsed entities. Instead, these entities are used in conjunction with the special attribute types ENTITY and ENTITIES: the value of an attribute of type ENTITY must match the name of an unparsed entity, and ENTITIES must match a whitespace separated nonempty list of such names. When an XML parser that performs validation with DTD encounters such attributes in the instance document, it informs the application of the associated SYSTEM/PUBLIC identifiers and notation names.

Notations may also be used directly in attributes: the value of an attribute of type NOTATION must match the name of a notation declaration. This can be used to describe formats of non-XML data.

Finally, *conditional sections* allow parts of the schema to be included or excluded via a switch. Although it is not indispensable, the typical use of this mechanism is to let an external DTD schema contain a number of declarations and then select only some of them for use in the instance document. Conditional sections are typically combined with the parameter entity mechanism. For example, an external DTD schema may contain

```
<![%person.simple; [
  <!ELEMENT person (firstname,lastname)>
]]>
<![%person.full; [
  <!ELEMENT person (firstname,lastname,email+,phone?)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
]]>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
```

In the internal DTD schema, we can then define either

```
<!ENTITY % person.simple "INCLUDE" >
<!ENTITY % person.full   "IGNORE" >
```

to select the simple version of the `person` element, or

```
<!ENTITY % person.simple "IGNORE">
<!ENTITY % person.full   "INCLUDE">
```

to get the full version. The parameter entity references expand to the keywords `IGNORE` and `INCLUDE`, which result in, respectively, disabling and enabling the contents of the blocks. The conditional section mechanism is used extensively in the XHTML modularization framework that we described in Section 2.5.

We have deliberately left out a number of annoying technical details of the DTD language in this section. Most users of XML fortunately never need to know about them, but if you for some reason are forced in that direction, then look in the XML specification (its URL is provided among the online resources at the end of the chapter).

4.3.5 Checking Validity with DTD

A DTD processor (also called a *validating XML parser*) works by first parsing the instance document, that is, constructing its XML tree representation, and the DTD schema, including all external schema subsets. Parsing succeeds if the document is well-formed. Note that since the DTD language does not itself use an XML notation, a different parsing technique (typically based on a BNF-like grammar) is needed for parsing the DTD schema.

Then, the DTD processor checks that the name of the root element of the instance document is correct. In the next phase, most of the actual validation work is performed in a simple traversal of the XML document: for each element node, the processor looks up the associated element declaration and attribute-list declarations and (1) checks that the content of the element node matches the content model of the element declaration, (2) normalizes the attributes according to the declarations, which consists of insertion of default attributes and pruning of whitespace where applicable, (3) checks that all required attributes are present, and (4) checks that the values of the attributes match the associated attribute types. Entity references are either expanded during the traversal or kept in the tree as entity reference nodes – in either case, the validation treats them as if they were expanded. In the same traversal, all `ID` attributes are collected and checked for uniqueness.

After this phase, the only thing that remains is the check for `IDREF` and `IDREFS` attributes. For each of these, the processor checks that each reference corresponds to some `ID` attribute.

Finally, if no validation errors were detected, the processor outputs the normalized instance document, either in its textual form or in its tree representation to the application. Naturally, a DTD processor is not required to follow this exact sequence of phases as long as it has the same resulting behavior.

As a final note, instance documents that do refer to external DTD declarations but do not rely on attribute defaults and entity declarations from these external parts may contain a *standalone declaration* in the XML declaration:

```
<?xml version="1.1" standalone="yes"?>
```

This declaration tells the XML processors that if they are only interested in parsing the document but not in checking validity, then the external DTD declarations can safely be ignored.

4.3.6 Recipe Collections with DTD

We are now in a position to formalize the syntax for RecipeML using DTD, based on the informal description given in Section 2.7:

```
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description (#PCDATA)>

<!ELEMENT recipe
     (title,date,ingredient*,preparation,comment?,
                             nutrition,related*)>
<!ATTLIST recipe id ID #IMPLIED>

<!ELEMENT title (#PCDATA)>

<!ELEMENT date (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step*)>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition calories CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    protein CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>

<!ELEMENT related (#PCDATA)>
<!ATTLIST related ref IDREF #REQUIRED>
```

This DTD schema simply uses the `ELEMENT` and `ATTLIST` constructs to describe the RecipeML vocabulary of elements and attributes. A `collection` element has no attributes but contains a `description` element followed by zero or more `recipe` elements, and so

on. The schema captures most of the requirements that were informally expressed earlier; however, there are some notable exceptions: even a simple XML language as RecipeML reveals practical limitations of the expressiveness of DTD. The schema presented above inevitably has obvious shortcomings:

DTD cannot express all details of the syntax of our small XML language for recipe collections.

- we cannot express that the `calories` attribute must contain a non-negative number and, for example, `protein` must contain a value on the form *`N`*`%` where *`N`* is between 0 and 100;
- the `unit` attribute should only be allowed in an element when `amount` is present; and
- nested `ingredient` elements should only be allowed when `amount` is absent.

As previously discussed, the effect is that the schema allows too much, so our applications that operate on the recipe collections need to perform some extra checking of the input to make sure that it is meaningful.

Moreover, we would have liked the `comment` element to be allowed to appear anywhere in the contents of a `recipe` element, not just between `preparation` and `nutrition`. This property is, in fact, expressible, but only with a long-winded content model expression that unions all the possible places where `comment` should be permitted (and the requirement that the expression must be deterministic makes this extra tricky). As a result, the schema in this case is too restrictive compared to our informal description.

Assume that we store this DTD schema in a file and make it available at the URL

```
http://www.brics.dk/ixwt/recipes.dtd
```

By inserting

```
<!DOCTYPE collection SYSTEM "http://www.brics.dk/ixwt/recipes.dtd">
```

in the headers of our recipe collection instance documents, we state that they are intended to conform to the designated DTD schema.

As described earlier, we can insert a fake namespace declaration in the schema:

```
<!ATTLIST collection
          xmlns CDATA #FIXED "http://www.brics.dk/ixwt/recipes">
```

Notice that in this simple example, we use only the fundamental features of DTD: element and attribute-list declarations. We have no need for parameter entities, notations, or the other murky legacies from SGML.

If we run a DTD schema processor on an invalid document, for example omitting the `description` element, we will get an error message that helps us locate the problem:

```
[Error] recipes.xml:366:14: The content of element type "collection"
must match "(description,recipe*)".
```

Here, `366:14` refers to the line and column of the end tag of the surrounding `collection` element. (This particular output is generated by the Xerces parser.)

If we accidentally had made a syntax error in the DTD schema, for instance, forgetting the parentheses around `#PCDATA`, then the schema processor might report the following:

```
[Fatal Error] recipes.dtd:5:23: A '(' character or an element type is
required in the declaration of element type "title".
```

It then aborts without performing any validation of the instance document. An easy way of checking that your newly written DTD schemas are syntactically correct according to the formal syntax of DTD is to run a DTD processor with a dummy instance document.

4.3.7 Limitations of DTD

DTD has a number of problems that limit its practical use:

1. DTD cannot constrain character data. In the contents of a particular element, either any character data is permitted or none at all. Obviously, we would like more control here, as mentioned earlier. What we need is a more powerful *datatype* mechanism for describing character data.
2. By the same token, the attribute types are too limited. DTD cannot specify that the value of a particular attribute must be, for example, an integer, a URI, a date, or whatever datatypes we might use.
3. Element and attribute declarations are entirely context insensitive. That is, descriptions cannot depend on attributes or element context, as we saw with `unit` attributes and `ingredient` elements in the DTD description of RecipeML. However, it is very common in the design of XML languages that certain declarations depend on whether or not a certain attribute exists and has a particular value or the current element has a particular ancestor element.
4. Character data cannot be combined with the regular expression content model. This means that if we need to permit character data in the contents, then we cannot control the order of elements or their number of occurrences.
5. The content models lack an ‘interleaving’ operator, as we just saw with the `comment` element in RecipeML.
6. DTD provides very limited support for modularity, reuse, and evolution of schemas. The entity and conditional section mechanisms are low-level and difficult to apply elegantly. This makes it hard to write, maintain, and read large DTD schemas and to define families of related schemas.
7. The normalization features in DTD are limited. Although there is a default mechanism for attributes, there is none for element contents. Also, using the special `xml:space`

attribute does not cause any removal of insignificant whitespace in character data by the DTD processor – it only indicates the intended meaning to the application.

8. DTD does not permit embedded structured self-documentation. Comments are allowed, but they cannot contain markup.
9. The `ID`/`IDREF` mechanism is too simple. Often, it is desirable to be able to specify a more restricted scope of uniqueness for `ID` attributes than the entire instance document. It is also inconvenient that only individual attribute values can be used as keys. It would be more useful if the key could consist of multiple attribute values or even character data.
10. DTD does not itself use an XML notation. A practical consequence of this design is that it is not possible to write a DTD description of the DTD language itself, which otherwise would have been handy for checking that a DTD schema is indeed a syntactically correct DTD schema. Instead, we need yet another formalism (Extended BNF) for describing the syntax of the DTD language. It also means that we cannot use standard XML tools to manipulate DTD schemas.
11. DTD does not support namespaces.

As an additional annoyance, the specification of DTD is mixed into the specification of the XML notation, which makes it difficult to separate the two.

Nevertheless, DTD is still widely used, sometimes for expressing rough approximations of available elements, attributes, and the constraints of their use, perhaps as supplements to schemas written in a more complicated schema language, such as XML Schema.

4.4 XML Schema

Shortly after XML 1.0 (and thereby also DTD) was approved as a W3C recommendation, the W3C initiated the development of the next generation schema language to attack the problems with DTD. In 1999, a note *'XML Schema Requirements'* was published by the XML Schema Working Group, outlining the goals of the project. This note lists some judicious guiding *design principles*, some of which are that the language should be

- more expressive than XML DTD (this goal is clearly rather vague, but it indicates the overall direction);
- expressed in XML (we saw above the problems that resulted from DTD not being an XML language);
- self-describing (meaning that it should be possible to describe the syntax of XML Schema in the XML Schema language itself); and
- simple enough to implement with modest design and runtime resources (which sets a limit to the first goal listed above).

In addition, the working group proclaimed that the specification of XML Schema should be prepared quickly (otherwise competing schema languages might gain a foothold) and be precise, concise, human-readable, and illustrated with examples. Some of the more technical requirements were that the language should

- contain a mechanism for constraining the use of namespaces;
- allow creation of user-defined *datatypes* for describing character data and attribute values;
- enable inheritance for element, attribute, and datatype definitions;
- support evolution of schemas; and
- permit embedded documentation within the schemas.

In 2001, the working group published the XML Schema W3C recommendation. The specification comprises two parts: Part 1, *Structures*, describes the core of XML Schema including declaration of elements and attributes; Part 2, *Datatypes*, describes the built-in datatypes and their various facets.

Unfortunately, the resulting language does not fulfill all the original requirements: although the language does provide good support for namespaces and datatypes, which are two of the most pressing issues with DTD, it is not simple (Part 1 alone is more than 130 dense pages, and even XML experts do not find it human-readable), and it is not fully self-describing (there is a schema for XML Schema, but it does not capture all syntactical aspects of the language). Furthermore, as we shall see later, despite its complexity this ambitious schema language still cannot express the full syntax of our little RecipeML language.

4.4.1 Overview

Before delving into the details of the language, let us begin with a birds-eye view. To facilitate reuse of descriptions and improve the structure of schemas, XML Schema contains a sophisticated type system inspired by those known from programming languages, in particular object-oriented languages. The following four constructs are the most central ones in XML Schema, and they all hinge on the notion of types:

- A *simple type definition* defines a family of Unicode text strings.
- A *complex type definition* defines a collection of requirements for attributes, subelements, and character data in the elements that are assigned that type.
- An *element declaration* associates an element name with either a simple type or a complex type. For an element in the instance document to be valid, it must satisfy all requirements defined by the type that is associated with the name of the element. If all elements in the instance document are valid, then the entire document is valid.

- An *attribute declaration* associates an attribute name with a simple type. (Since attribute values always contain unstructured text, only simple types make sense here.)

Intuitively, a *simple* type describes text without markup (in character data and attribute values), whereas a *complex* type describes text that may contain markup (that is, elements, attributes, and character data).

Two types or two elements cannot be defined with the same name within one schema, but an element or attribute declaration and a type definition may use the same name. An element or attribute declaration with no type description permits any value. We will later see how to develop advanced type definitions.

The following tiny but complete schema shows one element declaration named `student`, two attribute declarations named `id` and `score`, one complex type definition named `StudentType`, and one simple type definition named `Score`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:s="http://www.brics.dk/ixwt/students"
            targetNamespace="http://www.brics.dk/ixwt/students">

  <xsd:element name="student" type="s:StudentType"/>

  <xsd:attribute name="id" type="xsd:string"/>
  <xsd:attribute name="score" type="s:Score"/>

  <xsd:complexType name="StudentType">
    <xsd:attribute ref="s:id" use="required"/>
    <xsd:attribute ref="s:score" use="required"/>
  </xsd:complexType>

  <xsd:simpleType name="Score">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

XML Schema elements are identified by the namespace `http://www.w3.org/2001/XMLSchema`. In this example, we use the namespace prefix `xsd`, but, as usual, the choice of prefix is insignificant. As the example shows, the root element of an XML Schema document is named `schema`, and it (usually) contains a `targetNamespace` attribute that indicates the namespace being described by the schema, in this case `http://www.brics.dk/ixwt/students`. We also declare this namespace by a normal namespace declaration, here using the prefix `s`, so that we are able to refer to our own definitions in the schema.

In XML Schema, **definitions** create new types, and **declarations** describe constituents of the instance documents.

The declarations and definitions in the schema then *populate* the target namespace: the `element` construct declares the element name `student` and associates to it the type

`StudentType` (from our own namespace); the following two `attribute` constructs declare two attributes named `id` and `score`; the `complexType` construct defines the type `StudentType` as two attribute references; and the `simpleType` construct defines a type `Score`, which is used to describe that the legal values of the `score` attribute are integers between 0 and 100. Both `student`, `id`, `score`, `StudentType`, and `Score` then belong to the target namespace. Notice that some of the attribute values in the schema use namespace prefixes to locate the right definitions: since `StudentType` is one of our own types, we refer to it using the `s` prefix, whereas `string` and `integer` are built-in types, so for those we use the `xsd` prefix. Definition names are never prefixed – they always belong to the target namespace. The details of the attribute declarations and the simple type definition will be explained later.

Notice the syntax for element and attribute declarations:

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

and

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

The former associates a simple or complex type to an element name; the latter associates a simple type to an attribute name.

The following instance document consisting of a single element is valid according to the schema shown above:

```
<?xml version="1.0" encoding="UTF-8"?>
<stu:student xmlns:stu="http://www.brics.dk/ixwt/students"
             stu:id="19970233"
             stu:score="97"/>
```

The `student` element here resides in the namespace that was declared as the target namespace in the schema, and the element fulfills all requirements defined in the schema for elements of that name. Notice that also the two attributes are explicitly qualified using the `stu` prefix. Typically we do not want prefixes in attribute names; we shall see in Sections 4.4.4 and 4.4.5 how to avoid them.

The presentation of XML Schema in this chapter is reasonably complete. The essential language constructs are explained in sufficient detail for most schema authors. However, to keep the presentation concise, lots of technical details of this very complex language inevitably have to be omitted.

Example: Business Cards

We now consider a slightly more complicated example that we will also build on later in the book. Assume we want to create an XML-based language for *business cards* where each

card consists of a name, a title, an email address, an optional phone number, and an optional link to a logo image. An example document could be following, which we store in a file `john_doe.xml`:

```
<b:card xmlns:b="http://businesscard.org">
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.inc</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo b:uri="widget.gif"/>
</b:card>
```

We here assume that we own the domain `businesscard.org` such that we can be certain that no one else uses the namespace `http://businesscard.org`. Again, a more common design would not use a prefix in the `uri` attribute; we fix this in Sections 4.4.4 and 4.4.5.

To describe the syntax of our new language, we write a schema, `business_card.xsd`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <attribute name="uri" type="anyURI"/>

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="logo_type">
    <attribute ref="b:uri" use="required"/>
  </complexType>

</schema>
```

From here on, we use the empty namespace prefix for XML Schema elements in the examples to make the documents more readable.

This schema contains six element declarations followed by one attribute declaration and two complex type definitions. The one named `card_type` contains the constraints of `card` elements, and the one named `logo_type` is for the `logo` elements. The `card_type` definition resembles a DTD content model. It states that for an element to match this type, its content must be a sequence of a `name` element followed by a `title` and an `email` element and then optional `phone` and `logo` elements. The `logo_type` definition corresponds to a simple attribute-list declaration in DTD. It states that for an element to match this type, it must have an attribute named `uri` whose value is a URI. The type `anyURI` is one of the built-in simple types in XML Schema.

As in DTD, only contents and attributes that are explicitly declared are permitted. For example, a `card` element is not allowed to have any attributes, and a `logo` element is not allowed to have any contents. However, there are a few exceptions to this rule, as explained later.

Connecting Instance Documents and Schemas

An instance document may refer to a schema with the `schemaLocation` attribute from the namespace `http://www.w3.org/2001/XMLSchema-instance`:

```
<b:card xmlns:b="http://businesscard.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://businesscard.org
                            business_card.xsd">
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.inc</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo b:uri="widget.gif"/>
</b:card>
```

The value of the `schemaLocation` attribute consists of two parts that are separated by whitespace: a namespace URI, which must match the target namespace of the schema, and a URI that locates the schema document. (It is not particularly pretty to use attributes in this way – it would probably have been more natural to use two attributes or sub-elements instead.) By inserting this attribute, the author asserts that the instance document is intended to be valid with respect to the schema. This mechanism is reminiscent of the `DOCTYPE` declarations in DTD, however, there are some important differences: First, XML Schema exploits the standard namespace mechanism to make the connection between instance document and schema rather than introducing some new specialized syntax. Second, `schemaLocation` attributes may appear in any elements in the instance document, whereas `DOCTYPE` is restricted to the root. Most often, `schemaLocation` attributes appear in the root elements, but they may also appear further down the tree if they only apply to subtrees. This gives

the instance document author the ability to combine XML languages in an ad hoc manner. Each `schemaLocation` attribute may, in fact, contain *multiple* pairs of namespace URIs and schema URIs; the effect is then that all the schemas apply independently of each other.

All attributes from the `http://www.w3.org/2001/XMLSchema-instance` namespace are implicitly always declared for all elements in the instance documents, so our schemas do not have to declare that, for example, the `schemaLocation` attribute is permitted.

The `schemaLocation` references are merely hints to the XML processors. It is not required that such references are inserted, neither are applications forced to take them into account. The main purpose of the schema references is to make the instance documents *self-describing* in the same way that most XML documents begin with an XML declaration that states the XML version and character encoding being used and also contain namespace declarations that identify the particular XML language being used. Typically, applications require their input to be valid relative to particular schemas that the application writer decides rather than to schemas that the instance document authors decide. Other schema languages use different approaches for connecting schemas and instance documents: we have seen that DTD uses the special `DOCTYPE` construct (see Section 4.3.1); the DSD2 language (see Section 4.5) instead uses a specific processing instruction; and RELAX NG (see Section 4.6) has no mechanism for specifying the association.

XML Schema contains no direct mechanism for enforcing a particular root element in the instance documents (however, there is a hack that we describe in Section 4.4.4). This means that, for example, an XML Schema description of XHTML cannot express the requirement that the root element must be an `html` element. The rationale behind this design choice is that the application that reads the instance document should be free to decide what may or may not be a root element. In practice, however, this just means that applications must check the name of the root element manually in addition to performing XML Schema validation.

4.4.2 Simple Types

A **simple type** describes text without markup; a **complex type** describes text that may contain markup.

A *simple type*, also called a *datatype*, is a set of Unicode strings together with a semantic interpretation of the strings. For example, `decimal` (from the XML Schema namespace) is a built-in simple type whose value space consists of all strings that represent decimal numbers, such as `3.1415`. Since the interpretation for this type is that the values represent decimal numbers, the two strings `3.1415` and `03.141500` are *equal*, and the values of the type are *ordered*, for example, `42` is *less than* `117`. Other simple types have different interpretations, and not all are ordered. These interpretations are, for example, used when atomizing values in XQuery (see Chapter 6).

XML Schema contains a number of *primitive* simple types, whose meanings are predefined, and various mechanisms for *deriving* new types from existing ones. The primitive simple types are listed in Figure 4.5. Some derived simple types that were expected to be commonly used have been included in the XML Schema specification; these are listed in Figure 4.6.

An element declaration can assign a simple type to an element name:

```
<element name="serialnumber" type="nonNegativeInteger"/>
```

Type	*Typical values*
string	*any Unicode string*
boolean	true, false, 1, 0
decimal	3.1415
float	6.02214199E23
double	42E970
duration	P1Y2M3DT10H30M (1 year, 2 months, 3 days, 10 hours, and 30 minutes)
dateTime	2004-09-26T16:29:00-05:00,2004-09-26T21:29:00Z
time	16:29:00-05:00, 21:29:00Z
date	2004-09-26, 2004-09-26-05:00
gYearMonth	2004-09, 2004-09-05:00
gYear	2004, 2004-05:00
gMonthDay	--09-26, --09-26-05:00
gDay	--26, --26-05:00
gMonth	--09, --09-05:00
hexBinary	48656c6c6f0a
base64Binary	SGVsbG8K
anyURI	http://www.brics.dk/ixwt/
QName	rcp:recipe, recipe
NOTATION	*depends on notations declared in the current schema*

Figure 4.5 Primitive simple types.

This particular element declaration assigns the built-in simple type `nonNegativeInteger` to elements named `serialnumber`. The effect is that the contents of such an element must consist of character data that matches `nonNegativeInteger` (perhaps with surrounding whitespace, see Section 4.4.9), and it cannot contain attributes or child elements.

New simple types can be derived from existing ones in three different ways:

- A `restriction` of an existing type, called a *base type*, defines a new type by restricting the set of possible values of the original one. This restriction can be performed on various *facets* of the type, as shown in Figure 4.7.

  The following defines a type for non-negative decimals with the built-in `decimal` type as base:

```
<simpleType name="nonNegativeDecimal">
  <restriction base="decimal">
    <minInclusive value="0"/>
  </restriction>
</simpleType>
```

  The next example illustrates that a single restriction may involve multiple constraining facets:

```
<simpleType name="score_from_0_to_100">
  <restriction base="integer">
    <minInclusive value="0"/>
```

Type	*Typical values*
normalizedString	*as* string *but whitespace facet is* replace
token	*as* string *but whitespace facet is* collapse
language	en, da, en-US
NMTOKEN	42, my.form, r103
NMTOKENS	42 my.form r103
Name	my.form, r103, rcp:recipe
NCName	my.form, r103
ID	*as* NCName
IDREF	*as* NCName
IDREFS	my.form r103
ENTITY	*as* NCName
ENTITIES	my.form r103
integer	42, -87, +42, 0
nonPositiveInteger	-87, 0
negativeInteger	-87
nonNegativeInteger	42, 0
unsignedLong	18446744073709551615
unsignedInt	4294967295
unsignedShort	65535
unsignedByte	255
positiveInteger	42
long	-9223372036854775808, 9223372036854775807
int	-2147483648, 2147483647
short	-32768, 32767
byte	-128, 127

Figure 4.6 Built-in derived simple types.

Facet	*Constraining*
length	length of string or number of list items
minLength	minimal length
maxLength	maximal length
pattern	regular expression constraint
enumeration	enumeration value
whiteSpace	controls white space normalization (Section 4.4.9)
maxInclusive	inclusive upper bound (for ordered types)
maxExclusive	exclusive upper bound
minInclusive	inclusive lower bound
minExclusive	exclusive lower bound
totalDigits	maximum number of digits (for numeric types)
fractionDigits	maximum number of fractional digits

Figure 4.7 Constraining facets of simple types.

```
    <maxInclusive value="100"/>
  </restriction>
</simpleType>
```

An enumeration restricts values to a finite set of possibilities:

```
<simpleType name="language">
  <restriction base="string">
    <enumeration value="EN"/>
    <enumeration value="DA"/>
    <enumeration value="FR"/>
  </restriction>
</simpleType>
```

Note that these facet restrictions operate on the semantic level, not the syntactic level. For example, restricting the `totalDigits` facet to `3` for the `decimal` type means that the values `123`, `0123`, and `0123.0` are all permitted, but `1234` and `120.05` are not.

The `pattern` constraining facet is a powerful mechanism for constraining values to regular expressions – the formalism that we generally introduced in Section 4.2. The actual syntax used in patterns is, in fact, a slight extension of the one presented in that section. The additions include character ranges (such as `[a-z]`) and repetitions (such as `x{3,7}`), which were also briefly mentioned earlier.

As an example, we can define a simple type for integers in the range 0 to 100 that are followed by a percentage character and where superfluous leading zeros are not allowed:

```
<simpleType name="percentage">
  <restriction base="string">
    <pattern value="([0-9]|[1-9][0-9]|100)%"/>
  </restriction>
</simpleType>
```

The first branch (`[0-9]`) describes a single digit, the second branch (`[1-9][0-9]`) describes two digits where the first is nonzero, and the third branch describes the value `100`. The entire branch structure is then followed by a percentage character.

Special symbols in patterns (such as `*`, `(`, and `|`) can be escaped by prefixing with a backslash (as in `\*`). Unicode character categories and blocks can be expressed conveniently with names, such as `\p{Sc}` for all currency symbols or `\p{IsHiragana}` for all hiragana characters.

For most facets, restrictions may be changed in further derivations unless the attribute `fixed="true"` is added to the constraining facet.

- A `list` of a type defines a whitespace separated sequence of values of that type:

```
<simpleType name="integerList">
  <list itemType="integer"/>
</simpleType>
```

- A `union` of a number of types denotes the union of their values:

```
<simpleType name="boolean_or_decimal">
  <union>
    <simpleType>
      <restriction base="boolean"/>
    </simpleType>
    <simpleType>
      <restriction base="decimal"/>
    </simpleType>
  </union>
</simpleType>
```

  Notice that we here use two 'dummy' restrictions inside the `union` – this is the only way to refer to an existing simple type inside a `union`.

Altogether, XML Schema provides useful mechanisms for describing datatypes. There are a few problematic issues nevertheless. First, one may argue that the `list` construct can easily be misused. If, during the design of a new XML language, one needs to describe a list of things, say integers, one could easily use the `list` construct to describe values such as the following:

```
<integerlist> 7 42 87 </integerlist>
```

However, one thereby loses some of the benefits of XML since additional parsing is necessary to split the character data into the integer constituents. Usually, a better solution is to add more markup to make the structure explicit:

```
<integerlist>
  <int>7</int>
  <int>42</int>
  <int>87</int>
</integerlist>
```

With this style, a standard XML parser suffices for determining the structure – at the price of more verbose markup.

A second problematic issue is that existing simple types cannot be used within regular expression patterns. For example, assume that a *product ID* is a string, such as `2005-09-26#0542`, consisting of a valid date followed by a `#` character and a four-digit integer. We can elegantly define simple types representing the date and the integer, but we cannot combine these with a concatenation operation as a regular expression – even though all constituents are regular languages.

Third, many other realistic examples show that the derivation mechanisms for simple types are often not sufficiently flexible. For example, the various hardwired types for date and time formats seem to be chosen more or less randomly as there exist other international standards for this kind of data that one might have preferred.

Simple types can be **derived** by restricting or combining existing types.

Finally, the type system of XML Schema is, in fact, much more involved than what we have seen so far. We come back to this issue in later sections.

Emulating DTD Features with Special Simple Types

Some of the built-in simple types have the same value spaces (see Figure 4.6) but different interpretations, for example, `NCName`, `ID`, and `ENTITY`. The types `ID`, `IDREF`, and others with names that are familiar from DTD are included for compatibility. The `ID`, `IDREF`, and `IDREFS` types provide the well-known functionality from DTD: attributes of type `ID` must have unique values, and every attribute of type `IDREF` must match the value of some `ID` attribute. An alternative and more powerful mechanism is more commonly used as explained in Section 4.4.10.

The `NOTATION` mechanism from DTD (see Section 4.3.4) can be expressed as `notation` declarations in XML Schema:

```
<notation name="gif" public="image/gif"
   system="http://www.iana.org/assignments/media-types/image/gif"/>
```

The values of attributes (or character data) of the type `NOTATION` must match such declarations. The usefulness of this feature is debatable.

Similarly, the values of type `ENTITY` or `ENTITIES` (or types derived from one of these) must match the names of unparsed entities as in DTD. However, XML Schema contains no mechanism for declaring unparsed entities; instead it relies on the presence of DTD declarations. Since XML Schema also contains no mechanism for declaring parsed entities, it is customary to utilize the one available in DTD. No other parts of the otherwise optional DTD schema are used when processing XML Schema schemas.

These DTD-like features are mostly used for emulating the special DTD mechanisms and are rarely used when developing new XML languages. The notion of conditional sections has no immediate counterpart in XML Schema, but the modularization features, which we describe in Section 4.4.7, are much more powerful.

4.4.3 Complex Types

In the previous section, we explained the meaning of assigning a simple type to an element name. An element declaration may alternatively assign a complex type to an element name, as in the schema for XML business cards:

```
<element name="card" type="b:card_type"/>
```

The meaning of such a declaration is that elements of that name must fulfill all requirements specified by the type. For complex types, these requirements may involve both attributes and contents, including child elements and character data.

A complex type can be defined using the XML Schema element `complexType` that has a `name` attribute, whose value is the name of the new type. The content of `complexType` can be of two kinds, *complex* and *simple*, as described below.

Constructing Complex Types with Elements and Attributes

The complex model can describe both elements and attributes. The elements are described through a variant of the regular expression formalism explained in Section 4.2. The alphabet then consists of all possible elements, and the various operators are written using the following XML syntax:

- An *element reference* is typically of the form

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

  where *name* is the name of an element that has been declared elsewhere. Such a reference matches a single element of the given name. (Notice the difference between an element named `element` with a `name` attribute and one with a `ref` attribute – the former is a declaration and the latter is a reference to a declaration.)

- Concatenation is expressed using a `sequence` element. The schema for business cards on page 117 shows an example.

- Union corresponds to a `choice` element. As an example, we might have defined the complex type `card_type` differently:

  ```
  <complexType name="alternative_card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <choice>
        <element ref="b:email"/>
        <element ref="b:phone" minOccurs="0"/>
      </choice>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>
  ```

  Compared to the definition on page 117, we now permit either an `email` element or a number of `phone` elements, but not both.

- The `all` construct is an additional operator for describing unordered contents. A content sequence matches an `all` expression if each constituent of the expression is matched somewhere in the content sequence, and conversely, all elements in the content sequence are matched by some constituent of the expression. One may think of this construct as a variant of `sequence` where the content order is irrelevant.

As an example, the following variant of `card_type` uses `all` instead of `sequence` to avoid restricting the ordering of the elements:

```
<complexType name="another_alternative_card_type">
  <all>
    <element ref="b:name"/>
    <element ref="b:title"/>
    <element ref="b:email"/>
    <element ref="b:phone" minOccurs="0"/>
    <element ref="b:logo" minOccurs="0"/>
  </all>
</complexType>
```

Unfortunately, the `all` construct is restricted in various ways (see pages 127 and 128).

- The `any` construct is a wildcard that matches any element. If the attribute `namespace` is present, only elements from specific namespaces are matched: its value is a whitespace separated list of namespace URIs; the special value `##targetNamespace` means the target namespace, and `##local` means the default namespace (the empty URI). Two other values are possible: `##any` (the default) means 'any namespace' and `##other` means 'any namespace except the target namespace'.

  This construct is particularly useful for defining *open* schemas where there is room for extension as in the following alternative definition of the type `card_type`:

```
<complexType name="yet_another_alternative_card_type">
  <sequence>
    <element ref="b:name"/>
    <element ref="b:title"/>
    <element ref="b:email"/>
    <element ref="b:phone" minOccurs="0"/>
    <element ref="b:logo" minOccurs="0"/>
    <any namespace="##other"
         minOccurs="0" maxOccurs="unbounded"
         processContents="skip"/>
  </sequence>
</complexType>
```

Here we explicitly permit elements from other namespaces than the target namespace to appear after the explicitly described elements. The `processContents="skip"` attribute instructs the schema processor to skip checking validity of these extra elements and their descendants. Other possible values are `strict` (the default), which means that checking must be performed, and `lax`, which means checking must be performed only on those elements where a schema description is available.

Another common use of `any` is for connecting sublanguages loosely. For example, we may in a schema for WidgetML (see Section 2.6) declare the `info` element, which may contain XHTML data, using the following type:

```
<complexType name="info_type">
  <sequence>
    <any namespace="http://www.w3.org/1999/xhtml"
         minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

To match this type, the contents must consist of one or more element from the XHTML namespace, but we do not in this schema specify which elements are permitted in that namespace. A more tight integration of sublanguages is possible with the `import` mechanism described in Section 4.4.7.

Additionally, the contents of `complexType` can be empty, which simply corresponds to the empty `sequence` of elements. The various constructs are summarized in Figure 4.8 (which also mentions a `group` construct that we explain later; see page 132).

A few restrictions apply to the `all` construct: `all` may only contain `element` references (not `sequence`, `choice`, or `all`), no element may be described more than once within a given `all` operator, and `sequence` and `choice` cannot contain `all` operators. Additionally, a complete expression cannot consist of one `element` or `any` declaration alone (just wrap it into a `sequence` or `choice` if you need to express a single element reference, as in the `info_type` example above). Also, all expressions must be *deterministic* as in DTD (see Section 4.3.2). The rationale behind these peculiar and often annoying restrictions is in most cases that XML Schema processors should be easier to implement.

A `complexType` may optionally also contain a number of *attribute references* that typically have the form

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

where *name* is the name of the attribute that has been declared elsewhere. Such a declaration states that elements that are assigned the complex type containing this declaration may

Construct	*Meaning*
`element`	element reference
`sequence`	concatenation
`choice`	union
`all`	unordered sequence
`any`	any element
`group`	named subexpression

Figure 4.8 Constructs used in complex content model descriptions.

have an attribute of the given name and with a value that matches the given type. Attribute references must be placed *after* the content model description.

Every attribute reference can have an attribute named `use` whose value can be `optional` (the default) or `required` with the obvious meaning.

Similarly, for the content model part, each `element`, `sequence`, `choice`, `all`, and `any` element (and also `group`, which we describe later) occurring in a type definition may have attributes named `minOccurs` and `maxOccurs` that define *cardinalities* of the declarations. The values can be non-negative integers, and the special value `unbounded` is also permitted for `maxOccurs`. By default, both attributes have the value `1`. For example, the expression

```
<element ref="r:recipe" minOccurs="0" maxOccurs="unbounded"/>
```

matches any sequence of zero or more `recipe` elements (assuming that the prefix `r` is properly declared), and

```
<choice maxOccurs="unbounded">
  <element ref="xhtml:th"/>
  <element ref="xhtml:td"/>
</choice>
```

matches any sequence of one or more elements, each named `th` or `td` (from the right namespace). For the `all` construct, `minOccurs` must be 0 or 1, and `maxOccurs` must be 1 – and similarly for `element` constructs inside `all`.

We are now in a position to fully understand the complex type definitions in the business card example we saw earlier:

```
<complexType name="card_type">
  <sequence>
    <element ref="b:name"/>
    <element ref="b:title"/>
    <element ref="b:email"/>
    <element ref="b:phone" minOccurs="0"/>
    <element ref="b:logo" minOccurs="0"/>
  </sequence>
</complexType>

<complexType name="logo_type">
  <attribute ref="b:uri" use="required"/>
</complexType>
```

The first type definition describes a sequence of elements where the first three are required (since `minOccurs` and `maxOccurs` are both `1` by default) and the last two are optional (because of the `minOccurs="0"` attributes). The second type definition describes a single attribute, which is required in elements that are assigned that type.

With the complex content model, the `complexType` element may optionally contain an attribute `mixed="true"`, which means that, in addition to the elements that have been declared in the contents, also arbitrary character data is permitted anywhere in the contents. This resembles the mixed content model in DTD, except that we now have control of the order and number of occurrences of the elements in the contents. Without `mixed="true"`, only whitespace is permitted. If we wish to constrain the character data according to some simple type, we have to use the simple content model, which we describe later. However, it is not possible with XML Schema to constrain the character data if we want to permit both character data and elements in a content sequence.

The following example shows a complex type describing both mixed contents and an attribute (we assume that the `n` prefix identifies the target namespace):

```
<element name="order" type="n:order_type"/>
<attribute name="id" type="unsignedInt"/>

<complexType name="order_type" mixed="true">
  <choice>
    <element ref="n:address"/>
    <sequence>
      <element ref="n:email" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="n:phone"/>
    </sequence>
  </choice>
  <attribute ref="n:id" use="required"/>
</complexType>
```

In this definition, an `order` element must contain elements according to the regular expression (`<choice>...</choice>`), together with arbitrary character data and a mandatory attribute named `id`.

As a counterpart to `any` for attributes, the attribute declarations in a complex type definition may be followed by an `anyAttribute` declaration. This declaration uses `namespace` and `processContents` attributes in the same way as `any`, as described earlier.

Type Derivation

In the following, we look into the mechanisms for deriving new complex types from existing types. These features facilitate reuse and comprehensibility and are directly inspired by the type systems found in object-oriented programming languages.

Constructing Complex Types with Simple Content

A complex type that describes attributes, but whose contents consist of character data only and no elements, can be described using a `simpleContent` construction that contains an

extension of a simple type where some attribute declarations are added. For example, we can build a complex type from a simple type (integer) and an attribute (named class, declared elsewhere):

```
<complexType name="category">
  <simpleContent>
    <extension base="integer">
      <attribute ref="n:class"/>
    </extension>
  </simpleContent>
</complexType>
```

An element matches this type if its contents constitute an integer and the element has no attributes, except an optional one named class. In this way, we effectively construct a complex type from simple types.

A complex type with *simple content* can also be constructed from an existing complex type, provided that it has simple content (as the category type shown above does, for example):

```
<complexType name="extended_category">
  <simpleContent>
    <extension base="n:category">
      <attribute ref="n:kind"/>
    </extension>
  </simpleContent>
</complexType>

<complexType name="restricted_category">
  <simpleContent>
    <restriction base="n:category">
      <totalDigits value="3"/>
      <attribute ref="n:class" use="required"/>
    </restriction>
  </simpleContent>
</complexType>
```

The type named extended_category extends the category type with an extra attribute declaration named kind (assumed to be declared elsewhere). Such an extension inherits the content type and attribute declarations from the base type. The type named restricted_category matches the same values as category, except that it only permits integers with at most three digits (see the discussion of the semantic versus the syntactic level of simple types in Section 4.4.2) and that the class attribute is no longer optional. Such a restriction also inherits the properties of the base type, but it may contain overriding restricting declarations. Note that a restriction always matches a subset of the values matched by the base type. The converse is *not* true for extensions since these may add extra tree components that are mandatory in all values of the derived type.

One combination of these constructs is not permitted: it is not possible to define a complex type with `simpleContent` as a `restriction` of a simple type, but in that case one could just use a simple type definition instead.

Derivation of Complex Types with Complex Content

We have so far seen two ways of constructing complex types: the complex model for describing XML structures with elements, and the simple model for controlling character data contents. As a third possibility, new complex types can be derived from existing ones having complex content models. (The complex type mechanism is indeed complicated!)

Assume that we have a description of 'basic' business cards containing just a `name` element:

```
<complexType name="basic_card_type">
  <sequence>
    <element ref="b:name"/>
  </sequence>
</complexType>
```

We might want to produce various extended versions of this description, all sharing a common base. That can be achieved using the `extension` mechanism for complex content types:

```
<complexType name="an_extended_card_type">
  <complexContent>
    <extension base="b:basic_card_type">
      <sequence>
        <element ref="b:title"/>
        <element ref="b:email" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Complex types can be derived by **extending** or **restricting** existing types.

The effect of this definition is that the original content model and the extension are concatenated, that is, an element matches `an_extended_card_type` if its contents are a `name` element followed by a `title` and an optional `email` element. It is not possible to extend content models by other means than concatenation. Also, it is not possible to define a complex type with complex content (that is, using `complexContent`) as an extension of a type with simple content (that is, either a simple type or a complex type defined with `simpleContent`). Generally for complex type derivations, the contents of `extension` can consist of `sequence`, `choice`, `all`, `group`, `attribute`, `attributeGroup`, and `anyAttribute` elements. For declaring mixed contents, a `mixed="true"` attribute can be placed in either the `complexType` or the `complexContent` element.

Complex types can also be derived by `restriction`:

```
<complexType name="a_further_derived_card_type">
  <complexContent>
    <restriction base="b:an_extended_card_type">
      <sequence>
        <element ref="b:name"/>
        <element ref="b:title"/>
        <element ref="b:email"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

Here, we construct a further derivation of `an_extended_card_type` where the `email` element is mandatory (recall that `minOccurs="1"` is the default). A type derived by restriction must repeat all the constituents of the content model of the base type, except that it is permitted to match fewer values. Attribute declarations need not be repeated, but, as in restrictions of complex types with simple contents, attribute declarations may be overridden by more restrictive ones.

Again, we see that type derivation by extension adds constituents to the valid documents, whereas type derivation by restriction confines the set of valid documents.

Generally, the type derivation mechanisms promote a well-structured reuse of descriptions, much like inheritance in object-oriented programming languages. Using the extension mechanism is reminiscent of defining subclasses in object-oriented languages by adding new fields. We shall see in Section 4.4.8 that the notion of subsumption known from object-oriented languages is also present in XML Schema.

Groups

Expressions used in complex types can be reused via the `group` construct. The following snippet is taken from an XML Schema description of XHTML:

```
<group name="heading">
  <choice>
    <element ref="xhtml:h1"/>
    <element ref="xhtml:h2"/>
    <element ref="xhtml:h3"/>
    <element ref="xhtml:h4"/>
    <element ref="xhtml:h5"/>
    <element ref="xhtml:h6"/>
  </choice>
</group>

<group name="block">
```

```
    <choice>
      <element ref="xhtml:p"/>
      <group ref="xhtml:heading"/>
      <element ref="xhtml:div"/>
      <group ref="xhtml:lists"/>
      <group ref="xhtml:blocktext"/>
      <element ref="xhtml:isindex"/>
      <element ref="xhtml:fieldset"/>
      <element ref="xhtml:table"/>
    </choice>
  </group>

  <complexType name="button.content" mixed="true">
    <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="xhtml:p"/>
      <group ref="xhtml:heading"/>
      ...
    </choice>
  </complexType>
```

In this example, a `heading` group is defined as the union of six element references, and this group is then subsequently used in another group definition named `block` and also in a complex type definition named `button.content`. The effect is as if the `choice` construction containing the six element references were inserted in place of the two group references.

Similarly, a collection of attribute declarations may be defined and reused multiple times using `attributeGroup`:

```
  <attributeGroup name="Focus">
    <attribute ref="xhtml:accesskey"/>
    <attribute ref="xhtml:tabindex"/>
    <attribute ref="xhtml:onfocus"/>
    <attribute ref="xhtml:onblur"/>
  </attributeGroup>

  <complexType name="SelectType" mixed="true">
    <choice maxOccurs="unbounded">
      <element ref="xhtml:optgroup"/>
      <element ref="xhtml:option"/>
    </choice>
    <attributeGroup ref="xhtml:Focus"/>
    ...
  </complexType>

  <complexType name="TextAreaType" mixed="true">
    ...
    <attributeGroup ref="xhtml:Focus"/>
  </complexType>
```

Four attribute declarations that are logically related are here collected into an attribute group called `Focus`, which is then used in two (abbreviated) complex type definitions. (As we have seen in earlier examples involving attribute declarations, this one also requires the attributes to have namespace prefixes in the instance documents; to avoid that, see Sections 4.4.4 and 4.4.5.)

Nil Values in Elements

When modeling databases, it is occasionally useful to be able to describe 'missing contents'. An element can be declared as *nillable*:

```
<element name="recipe" type="r:recipe_type" nillable="true"/>
```

This declaration means that the content of a `recipe` element in the instance document may be empty – even if its type `recipe_type` requires certain elements – provided that the `recipe` element contains a special `nil="true"` attribute from the XML Schema instance namespace:

```
<recipe xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:nil="true"/>
```

The attribute declarations are unaffected by this mechanism.

This mechanism provides a very simple form of conditional constraints: If a certain attribute is present and has a certain value (`xsi:nil="true"`), then one content model is applicable (the empty one), otherwise another content model is applicable (the one specified in the type). It would certainly be useful to have a more general form of such dependencies, as we discuss elsewhere in this chapter. Moreover, in many XML applications, nil values are often represented by absent elements instead of using `xsi:nil`.

4.4.4 Global versus Local Descriptions

In the schema examples we have seen so far, all element and attribute declarations and all type definitions appear at the top-level of the schema, that is, with the `schema` element as parent. Such descriptions are also called *global*. An alternative style is to use inlined, or *local*, descriptions. As an example, the following descriptions from our business card language use a purely global style:

```
<element name="card" type="b:card_type"/>
<element name="name" type="string"/>

<complexType name="card_type">
  <sequence>
    <element ref="b:name"/>
    <element ref="b:title"/>
```

```
    <element ref="b:email" maxOccurs="unbounded"/>
    <element ref="b:phone" minOccurs="0"/>
    <element ref="b:background" minOccurs="0"/>
  </sequence>
</complexType>
```

Notice that the complex type named `card_type` is only used once, namely in the declaration of `card` elements, and `name` elements only appear as children of `card` elements. The alternative declaration below has the same meaning but uses local descriptions for the complex type named `card_type` and the `name` child element:

```
<element name="card">
  <complexType>
    <sequence>
      <element name="name" type="string"/>
      <element ref="b:title"/>
      <element ref="b:email" maxOccurs="unbounded"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:background" minOccurs="0"/>
    </sequence>
  </complexType>
</element>
```

Notice how the `card_type` definition and the `name` element declaration have been inlined where they were used in the global style version.

Choosing between these two styles is largely a matter of personal preference of the schema author, however, there are some important technical differences:

- local type definitions are anonymous (as in the second `complexType` above), so they cannot be referred to for reuse – but on the other hand, we do not have to invent names for them;
- local element declarations can be *overloaded*, that is, two elements with the same name (and namespace) can have different types if one or both declarations are local – and similarly for attribute declarations;
- only globally declared elements can be starting points for validation (that is, root elements in the instance documents, if validating complete documents); and
- as we discuss in the next section on namespaces, local and global descriptions behave differently with respect to namespaces (in particular, local attribute declarations can describe non-prefixed attributes).

The schema author may exploit the third item listed above to express that only certain elements may occur as roots, but that implies a writing style where every reference to a non-root element name must repeat the associated type even if overloading is not used.

Local descriptions permit **overloading** of element and attribute declarations.

One concrete benefit of the purely global style is that it is always straightforward to find the description of elements with a given name. A benefit of applying the local style whenever possible is that descriptions more often appear directly where they are used, so this style tends to make the schemas more compact.

As an example of overloading, assume that we extend the business card language such that one XML document can contain a list of business cards, not just a single one. This list is expressed with a `cardlist` root element that contains a list of `card` elements. We also permit a `title` child element of `cardlist` for an XHTML description of the card list. The following instance document is then valid for this modified language:

```
<cardlist xmlns="http://businesscard.org"
          xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <title>
    <xhtml:h1>My Collection of Business Cards</xhtml:h1>
    containing people from <xhtml:em>Widget Inc.</xhtml:em>
  </title>
  <card>
    <name>John Doe</name>
    <title>CEO, Widget Inc.</title>
    <email>john.doe@widget.inc</email>
    <phone>(202) 555-1414</phone>
  </card>
  <card>
    <name>Joe Smith</name>
    <title>Assistant</title>
    <email>thrall@widget.inc</email>
  </card>
</cardlist>
```

Notice now that a `title` element appearing as a child of `cardlist` can contain arbitrary XHTML whereas a `title` element appearing as a child of `card` can contain only character data as before. With local descriptions of the `title` elements this design can be expressed as follows:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org"
        elementFormDefault="qualified">

  <element name="cardlist" type="b:cardlist_type"/>
  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <attribute name="uri" type="anyURI"/>
```

```
  <complexType name="cardlist_type">
    <sequence>
      <element name="title" type="b:cardlist_title_type"
               minOccurs="0"/>
      <element ref="b:card" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="cardlist_title_type" mixed="true">
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml"
           minOccurs="0" maxOccurs="unbounded"
           processContents="lax"/>
    </sequence>
  </complexType>

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element name="title" type="string"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="logo_type">
    <attribute ref="b:uri" use="required"/>
  </complexType>

</schema>
```

Compared to the original schema (page 117), we have here inlined the declaration of `title` elements. We use the `any` construct to establish the connection to the XHTML language, as explained earlier. The attribute named `elementFormDefault` is explained in the next section.

Overloading is particularly relevant for attributes. It is typical in XML languages that a given attribute name is used with different types in different elements. For example, the type for an `align` attribute in an `hr` element in XHTML is different from one in an `img` or `p` element.

Continuing the business card list example, we could apply local descriptions even further than the previous schema indicates, as the following variant shows:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org"
        elementFormDefault="qualified"
        attributeFormDefault="qualified">
```

```
  <element name="cardlist">
    <complexType>
      <sequence>
        <element name="title" minOccurs="0">
          <complexType mixed="true">
            <sequence>
              <any namespace="http://www.w3.org/1999/xhtml"
                   minOccurs="0" maxOccurs="unbounded"
                   processContents="lax"/>
            </sequence>
          </complexType>
        </element>
        <element name="card" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="name" type="string"/>
              <element name="title" type="string"/>
              <element name="email" type="string"/>
              <element name="phone" type="string" minOccurs="0"/>
              <element name="logo" minOccurs="0">
                <complexType>
                  <attribute name="uri" type="anyURI"
                             use="required"/>
                </complexType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>

</schema>
```

(The attribute named `attributeFormDefault` is explained in the next section.) Compared to the previous variant, we have here inlined all descriptions that were only used once. The resulting schema contains only a single global element declaration which has a high nesting depth.

The overloading mechanism is limited in an important way: two element declarations that have the same name and appear in the same complex type must have identical types. As an example, the following type is illegal:

```
<complexType name="illegal_type">
  <sequence>
    <element name="foo" type="some_type"/>
    <element name="foo" type="another_type"/>
  </sequence>
</complexType>
```

This limitation significantly reduces the theoretical expressiveness of XML Schema, but on the positive side, it greatly simplifies implementation of efficient XML Schema processors.

4.4.5 Namespaces

We have already encountered the influence of namespaces for XML Schema. Being an XML language itself, this schema language uses namespaces to identify the schema instructions. It also associates target namespaces (using the `targetNamespace` attribute) to the XML languages we are describing.

XML Schema does support XML languages without namespaces by omitting the `targetNamespace` attribute. Technically, the descriptions in the schema then populate the default (empty URI) namespace, and such a schema may be referred to using a `noNamespaceSchemaLocation` attribute in place of `schemaLocation` (see page 118).

So far, this is rather uncontroversial, but XML Schema also introduces some unconventional uses of namespaces that one should be aware of:

- First, XML Schema (as well as other XML languages, for example XSLT) uses namespace prefixes in certain attribute values. The namespace standard, as we discussed in Section 2.6, only describes prefixes on element names and attribute names. The meaning of prefixes in attribute values in XML Schema is not surprising though, as we have seen in the example schemas.
- A second, and more frustrating feature is that of *qualified* and *unqualified locals*, which is related to the discussion of global and local descriptions in the previous section.

  If 'unqualified locals' is enabled, then the name of a locally declared element or attribute must have *no* namespace prefix in the instance document; such an attribute or element is interpreted as belonging to the element declared in the surrounding global definition. For non-prefixed attributes, this is just the expected behavior (according to the usual interpretation of the namespace mechanism, as described in Section 2.6); however, this is not the case for elements: a non-prefixed element name is normally resolved by looking up the relevant default namespace declaration, as explained in Section 2.6.

  If 'qualified locals' is enabled, then the name of a locally declared attribute must have a namespace prefix, just like we have seen for globally declared attributes. For elements, however, it means that the namespace mechanism behaves just like we would expect from Section 2.6.

  These mechanisms may be controlled separately for elements and attributes, and both globally for the entire schema by `elementFormDefault` and `attributeFormDefault` attributes in the `schema` element and locally for each local declaration by a `form` attribute. The values of these special attributes are either `unqualified` or `qualified` corresponding to the two modes of operation. The value `unqualified` is the default for both elements and attributes. As explained above, for attributes, `unqualified` corresponds to the expected behavior of non-prefixed names, but this

is not the case for elements. So, in order to obtain a sane interpretation of namespace prefixes in XML Schema, one should always use

```
<schema ...
        elementFormDefault="qualified"/>
  ...
</schema>
```

Of course, this declaration is only relevant if using local element declarations.
To get rid of the prefixes on `uri` attributes in our business card example (see page 117), we simply change the attribute declaration from being global to local (and select unqualified local attributes by default):

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="logo_type">
    <attribute name="uri" type="anyURI" use="required"/>
  </complexType>

</schema>
```

For the remainder of the book, we use this version of the business card language. In short, unqualified local elements are bad practice: do not use them! For attributes, however, the situation is different: since the vast majority of attributes in existing XML languages are non-prefixed, most attribute declarations are local.

Qualified attributes, on the other hand, are certainly useful (although unqualified ones are more common). For example, the XLink language (see Section 3.6) can be incorporated into other languages using attributes that are qualified by the XLink namespace:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        targetNamespace="http://www.w3.org/1999/xlink"
        attributeFormDefault="qualified">

  <attribute name="href" type="anyURI"/>
  <attribute name="show" type="xlink:showType"/>
  <attribute name="actuate" type="xlink:actuateType"/>

  <attributeGroup name="simpleLink">
    <attribute name="type" type="string" fixed="simple"/>
    <attribute ref="xlink:href"/>
    <attribute ref="xlink:show"/>
    <attribute ref="xlink:actuate"/>
    ...
  </attributeGroup>
  ...

</schema>
```

(Here, `attributeFormDefault` applies only to the `type` attribute since that is the only locally declared attribute. The `fixed` attribute is explained in Section 4.4.9.) Such a schema can then be imported into the host language (see Section 4.4.7), using, for example, the `simpleLink` attribute group in element declarations to describe simple links.

4.4.6 Annotations

Schemas can be annotated with human or machine readable documentation and other information:

```
<element name="card">
  <annotation>
    <documentation xmlns:xhtml="http://www.w3.org/1999/xhtml">
       The 'card' element represents one business card.
       See <xhtml:a href="manual.html">the manual</xhtml:a> for
       more information.
    </documentation>
    <appinfo xmlns:p="http://printers-r-us.com">
       <p:paper type="117"/>
    </appinfo>
```

```
    </annotation>
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="title" type="string"/>
        <element name="email" type="string"/>
        <element name="phone" type="string" minOccurs="0"/>
        <element name="logo" type="logo_type" minOccurs="0"/>
      </sequence>
    </complexType>
  </element>
```

Such `annotation` elements may appear in the beginning of the contents of most schema constructs. The `documentation` sub-element is intended for human readable information, whereas `appinfo` is for applications. The actual contents are ignored by the schema processor. Information may also be specified externally via `source` attributes in the `documentation` and `appinfo` elements.

Note that annotations can be structured, as opposed to simple `<!-- ... -->` XML or DTD comments. For example, the `documentation` annotation in this example contains XHTML markup.

4.4.7 Modularization

Modularization is crucial for real world schemas.

Schemas for realistic XML languages can become colossal due to large numbers of elements and attributes that need to be described. Also, such XML languages rarely exist in isolation but are developed in families of related languages and evolve from existing ones. As an example, the Danish OIOXML initiative for public administration is built of around 2500 interconnected XML Schema definitions. This calls for mechanisms for *modularization* of descriptions.

To support structuring, reuse, and evolution, schemas can be expressed as modules of manageable sizes that can be combined to describe complete XML languages. XML Schema provides three constructs for combining schemas in the form of instructions that may appear initially in the `schema` element:

- `<include schemaLocation="`*`URI`*`"/>` – composes with the designated schema having the *same* target namespace as this schema. The effect is as if the `include` instruction was replaced by the contents of the included schema.
- `<import namespace="`*`NS`*`" schemaLocation="`*`URI`*`"/>` – composes with the designated schema having a *different* target namespace *`NS`*. The `schemaLocation` attribute is optional; if omitted, the imported schema must be located by other means.
- `<redefine schemaLocation="`*`URI`*`"> ... </redefine>` – as `include`, but permits redefinitions, that is, definitions of types and groups (appearing as the contents of the `redefine` element) that then take precedence of the definitions of the same names

in the included schema. A type redefinition must be either an extension or a restriction of the original type, and similarly, a group redefinition must describe either a superset or a subset of the original definition (for some obscure reason).

The schema for business cards already has a quite manageable size, but as an example, let us nevertheless split it into two modules, one containing the description of the `logo` elements, and the other containing the rest. First, `business_card_logo.xsd`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="logo" type="b:logo_type"/>

  <complexType name="logo_type">
    <attribute name="uri" type="anyURI" use="required"/>
  </complexType>

</schema>
```

The second module, `business_card_misc.xsd`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

</schema>
```

These two modules can then be combined as in the following schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://businesscard.org">
```

```
  <include schemaLocation="business_card_misc.xsd"/>
  <include schemaLocation="business_card_logo.xsd"/>

</schema>
```

This schema then has the same meaning as the original one presented on page 117.

Now, because of this modularization, we know that all `logo` related information is located in the file `business_card_logo.xsd`. Assume that we want to define a variant of our XML language where the `logo` element also has a `contenttype` attribute (with any value), but otherwise is exactly as before. This design can be obtained using `redefine`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <include schemaLocation="business_card_misc.xsd"/>

  <redefine schemaLocation="business_card_logo.xsd">
    <complexType name="logo_type">
      <complexContent>
        <extension base="b:logo_type">
          <attribute name="contenttype" type="string"/>
        </extension>
      </complexContent>
    </complexType>
  </redefine>

</schema>
```

Notice how the new definition refers to the original one as base for extension. Except from this apparent self-reference, a redefinition of some name has effect for every use of that name, including ones that appear in the schema module containing the original definition. The redefinition mechanism should be used with caution, as excessive use tends to make the schemas incomprehensible.

Since all descriptions within one schema file populate the same target namespace, using the `import` mechanism is the only way an XML language consisting of multiple namespaces can be described.

We saw on pages 127 and 136 how XHTML could be integrated with our XML language for lists of business cards using the `any` construct. With `import`, we can now specify a more tight connection, assuming that we have an XML Schema description for XHTML:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        xmlns:xhtml="http://www.w3.org/1999/xhtml"
```

```
        targetNamespace="http://businesscard.org"
        elementFormDefault="qualified">

  <import namespace="http://www.w3.org/1999/xhtml"
          schemaLocation="xhtml.xsd"/>

  <element name="cardlist">
    <complexType>
      <sequence>
        <element name="title">
          <complexType mixed="true">
            <sequence>
              <group ref="xhtml:Block.mix"/>
            </sequence>
          </complexType>
        </element>
        <element ref="b:card" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>

  ...
</schema>
```

The '. . .' abbreviates the remaining descriptions, which are as before. We here import the schema for XHTML and refer to the `Block.mix` group definition in that schema when describing the valid contents of those `title` elements that occur as child of `cardlist`. The difference between this approach and the one using `any` is that we can now control exactly which contents are allowed. This example is inspired by W3C's modularization of XHTML, as briefly described in Section 2.5, where `Block.mix` is the name of the description of the valid contents of XHTML `body` elements.

4.4.8 Subsumption and Substitution Groups

We have seen how to derive existing types by extension or restriction. An XML value that matches a given type that has been defined as a restriction of some base type is guaranteed to also match the base type. In other words, an application that expects certain attributes and contents of an element of some type would not be surprised by values of a restriction of that type. The converse is *not* true for type extensions since these may add new mandatory elements or attributes. Nevertheless, both kinds of type derivation permit *subsumption*: If the schema at some point in the instance document requires an element to match a certain type B, then it is acceptable that the element instead matches any type D that, in some number of steps, is derived from B by restriction or extension. This property is trivial for type restrictions. However, for type extensions, it requires that the element has a special

attribute from the XML Schema instance namespace identifying the actual type. Consider again the following schema declarations and definitions:

```
<element name="card" type="b:basic_card_type"/>
<element name="name" type="string"/>
<element name="title" type="string"/>
<element name="email" type="string"/>

<complexType name="basic_card_type">
  <sequence>
    <element ref="b:name"/>
  </sequence>
</complexType>

<complexType name="an_extended_card_type">
  <complexContent>
    <extension base="b:basic_card_type">
      <sequence>
        <element ref="b:title"/>
        <element ref="b:email" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

and an instance document:

```
<card xmlns="http://businesscard.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="an_extended_card_type">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.inc</email>
</card>
```

This instance document is valid relative to the schema. The declaration of `card` elements mentions only a `name` child. The instance document does not satisfy that directly, but it does match a derived type and the `card` element contains a special `xsi:type` attribute that specifies the actual type.

This mechanism is typically used together with the modularization mechanisms: basic types are defined in core modules, and derived types are placed in other modules that extend the core in different directions.

The built-in type `anyType` is a complex type that directly or indirectly is a basis for all other types, that is, every type is derived in some number of steps from `anyType`. An element which is assigned this type may contain arbitrary contents and attributes. (This type resembles the `Object` class in Java, for example.) Similarly, the built-in type `anySimpleType` is a basis for all simple types, and is naturally a derivation of `anyType` as all other types. The main difference between `string` and `anySimpleType` is that the latter has no constraining

facets, but both match all Unicode strings. The type `anyType` is the default for element declarations, and `anySimpleType` is the default for attribute declarations.

Technically, the form of type definitions described earlier where a `complexType` directly contains content model and attribute descriptions (as the definition of `basic_card_type` above, for example) is, in fact, a shorthand for a `complexType` with a `complexContent` containing a `restriction` of `anyType`.

Independently from the `xsi:type` attribute mechanism, XML Schema contains a mechanism called *substitution groups*, which provides somewhat similar possibilities. An element declaration *D* can be placed in the substitution group of another element declaration *B*, provided that both are global and the type of *D* is in one or more steps derived from the type of *B*. The effect is that whenever a *B* element is required, a *D* element may be used instead – without using the `xsi:type` attribute in the element. The following schema fragments illustrate this effect:

```
<element name="cardlist">
  <complexType>
    <sequence>
      <element ref="b:basic-card" minOccurs="0"
                                  maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>

<element name="basic-card" type="b:basic_card_type"/>
<element name="extended-card" type="b:a_derived_card_type"
         substitutionGroup="b:basic-card"/>

<complexType name="basic_card_type">
  <sequence>
    <element name="name" type="string"/>
  </sequence>
</complexType>

<complexType name="a_derived_card_type">
  <complexContent>
    <extension base="b:basic_card_type">
      <sequence>
        <element name="title" type="string"/>
        <element name="email" type="string" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

A `cardlist` here contains a list of `basic-card` elements, but `extended-card` is declared as substitutable for `basic-card` elements. The following instance document is thus valid:

```
<cardlist xmlns="http://businesscard.org">
  <extended-card>
```

```
    <name>John Doe</name>
    <title>CEO, Widget Inc.</title>
    <email>john.doe@widget.inc</email>
  </extended-card>
  <basic-card>
    <name>Joe Smith</name>
  </basic-card>
</cardlist>
```

Compared to using the `xsi:type` attribute mechanism, we here do not rely on types from the schema in the instance document. Instead, the intended meaning is expressed by the element names.

In addition to all this, there are various mechanisms for controlling the use of derivation and substitution groups:

- If an element or type is declared to be *abstract* with `abstract="true"`, it cannot be used directly in instance documents. However, substitutable elements and derived types are still permitted.
- A complex type defined with the attribute `final="#all"` cannot be base for a derivation. The values `restriction` or `extension` prohibits only derivation by restriction or extension, respectively. For simple types, the values `list` and `union` can also be specified with a similar meaning. The `schema` element may contain an attribute `finalDefault` with a default value for the `final` attributes.
- The `block` attribute in a complex type definition can be used to constrain the use of type extensions, restrictions, and substitution groups. The possible values include `#all`, `restriction`, and `extension`, as above. For example, `block="extension"` means that types derived by extension of this complex type cannot be used in place of this type with the `xsi:type` attribute in the instance document. An element declaration may also contain `block="substitution"`, which prohibits substitution of another element where one matching this declaration is expected. As with `final`, a default value can be specified with a `blockDefault` attribute in the `schema` element.
- We have in Section 4.4.2 seen the use of the `fixed="true"` attribute in restrictions of simple types to prevent further changes of simple type facets. (Note that this is very different from the use of `fixed` in element and attribute declarations that we describe in Section 4.4.9.)

Clearly, these mechanisms are inspired by the type systems in object-oriented programming languages, and they serve similar purposes. However, things are made overly complicated by the derivation and substitution mechanisms.

The `xsi:type` attribute required in the instance documents when exploiting the subsumption mechanism serves to reduce the cost of checking validity and to determine the desired type among a potentially large set of possibilities. The substitution group mechanism can be seen as an alternative way of achieving similar goals.

4.4.9 Defaults and Whitespace Normalization

XML Schema provides a few normalization mechanisms as side-effects of the validation. As in DTD, defaults can be specified for attribute declarations:

```
<attribute name="uri" type="anyURI" default="anonymous.jpg"/>
```

If this declaration applies to a given element that does not have a `uri` attribute, the default `uri="anonymous.jpg"` is added.

A similar mechanism applies to elements:

```
<element ref="b:email" default="no email address available"/>
```

In the business card instance document, if an `email` element is empty, then the specified content is inserted by the schema processor. With this and the previous modification of the schema for business cards, processing the instance document

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email/>
  <phone>(202) 555-1414</phone>
  <logo/>
</card>
```

yields the following output:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>no email address available</email>
  <phone>(202) 555-1414</phone>
  <logo uri="anonymous.jpg"/>
</card>
```

Also, whitespace in attribute values and simple-typed element contents can be modified using the `whiteSpace` facet of simple type definitions:

```
<simpleType name="name_type">
  <restriction base="string">
    <whiteSpace value="collapse"/>
  </restriction>
</simpleType>
```

The value `replace` causes all tab, line feed, and carriage return characters in values that are assigned this type to be replaced by space characters; `collapse` behaves as `replace` but also collapses contiguous sequences of whitespace into single space characters and removes leading and trailing whitespace; and `preserve` (the default value of this facet) means that

no whitespace normalization should be performed. Regarding the built-in simple types, most of them have whitespace type `collapse`, except `string` which is of type `preserve`, and `normalizedString` which is of type `replace`.

Notice how these normalization mechanisms slightly generalize the corresponding mechanisms from DTD (see Section 4.3.3). In particular, normalization of element contents is now also possible; although, unfortunately, that only applies to elements that contain pure character data and no markup. As in DTD, the normalization takes place early in the process, before the actual validation.

The XML Schema counterpart to `#FIXED` from DTD is the `fixed` attribute, which can be used in place of `default` in both attribute and element declarations. The meaning is as in DTD, but now it also applies to elements: the attribute or element content is optional in the input instance document; if present, it must have the value specified by `fixed`; and if omitted, the value of `fixed` is used as a default.

In addition to the normalization features presented here, a schema processor may also contribute to the *post-schema-validation infoset* (PSVI). This slick name denotes pieces of information about the validation process that are made available to the application. For example, this information describes for each validated element which element declaration in the schema it matches and which types are assigned to attribute values. One use of this information is in XQuery (see Chapter 6) where values have types and control flow can be guided by type tests.

4.4.10 Uniqueness, Keys, and References

Recall that DTD can describe uniqueness and referential constraints using the `ID` and `IDREF` attribute types, and that XML Schema contains a compatibility feature for emulating this mechanism. XML Schema also contains a powerful alternative. First, recall some limitations with the DTD features: (1) they only apply to individual attributes, not combinations of attributes or element contents, and (2) the scope is always the entire XML document. XML Schema overcomes these limitations using XPath expressions for selecting fields and confining the scope.

The counterpart to `ID` is the `key` definition:

```
<element name="widget">
  <complexType>
    ...
  </complexType>

  <key name="my_widget_key">
    <selector xpath="w:components/w:part"/>
    <field xpath="@manufacturer"/>
    <field xpath="w:info/@productid"/>
  </key>
</element>
```

Notice how the key definition is specified separately from the type definition, in contrast to DTD where the ID attribute type also dictates the valid values. For each widget element, called the *current* element, the following steps are performed as a result of this particular key definition. The selector expression selects the set of part elements that have a components parent appearing in the contents of the current element. For each node in this set, two fields are produced as the values of the manufacturer attribute and the productid attribute below the info element, respectively. (We here assume that the selected part elements do, in fact, have such attributes – this must be declared elsewhere in the schema.) The selector expression is evaluated with the current element as context node and must result in a set of element nodes; the field expressions are evaluated with each selected element as context node, each one resulting in, at most, one attribute or simple-typed element node. In general, this results in an ordered list of field values for each selected node. Now, the document is only valid if there for each widget element are no two selected nodes having equal lists of field values. The notion of equality used here is the one defined for simple types (see Section 4.4.2).

XML Schema relies on a subset of **XPath** for expressing keys and references.

Additionally, the key definition causes an *identity-constraint table* to be constructed for the current element. This table is used to resolve keyref definitions, which intuitively correspond to the IDREF attribute types from DTD. The table contains for each key name the corresponding selected elements and their associated lists of field values. Assume that the widget element declaration also contains a keyref definition:

```
<keyref name="annotation_references" refer="w:my_widget_key">
  <selector xpath=".//w:annotation"/>
  <field xpath="@manu"/>
  <field xpath="@prod"/>
</keyref>
```

The selector and field parts are evaluated as before, but now, to be valid, each field value list being produced must instead match a corresponding field value list in the identity-constraint table with the name of the refer attribute. Intuitively, the selected annotation element is then a *reference* to the corresponding selected part element. The following instance document satisfies these key and keyref requirements:

```
<inventory xmlns="http://www.widget.inc">

  <widget>
    <components>
      <part manufacturer="Things'R'Us"><info productid="X1000"/></part>
      <part manufacturer="Gadgets4Ever"><info productid="A-42"/></part>
    </components>
  </widget>

  <widget>
    <components>
      <part manufacturer="Things'R'Us"><info productid="X800"/></part>
```

```
      <part manufacturer="Things'R'Us">
        <info productid="X1000"/>
      </part>
    </components>
    <annotation manu="Things'R'Us" prod="X800">
      Warning: The X800 model is really slippery when wet
    </annotation>
  </widget>

</inventory>
```

One identity-constraint table is constructed for each `widget` element; however, these tables are inherited upward in the instance document tree, so in some schemas the `keyref` definition does not always occur in the same element declaration as the corresponding `key` definition.

The next instance document is *invalid*:

```
<inventory xmlns="http://www.widget.inc">

  <widget>
    <components>
      <part manufacturer="Things'R'Us"><info productid="X1000"/></part>
      <part manufacturer="Gadgets4Ever"><info productid="A-42"/></part>
      <part manufacturer="Things'R'Us"><info productid="X1000"/></part>
    </components>
    <annotation manu="Things'R'Us" prod="X802">
      The X802 model is a vast improvement of the X800
    </annotation>
  </widget>

</inventory>
```

Here, the `key` requirement is violated by the two equal `part` constituents within the same `widget`, and the `keyref` selects a node that has no corresponding field list in the identity-constraint table.

For the XPath expressions in the `xpath` attributes, only a subset of the XPath 1.0 language is permitted in order to make implementation of XML Schema processors easier. The main restrictions are that only the `child`, `attribute`, and `descendant-or-self` axes can be used – the latter only in the abbreviated form (`//`), and that location step predicates are not allowed. One consequence is that the field nodes must be descendants of the element being validated.

XML Schema has a variant of `key` called `unique`. The main difference is that the designated fields are required to be present for `key`, whereas, for `unique`, a selected node is skipped if a designated field is absent. Note that `keyref` definitions can also match `unique` definitions.

At long last, this concludes our tour through the XML Schema language.

4.4.11 Recipe Collections with XML Schema

Having been introduced to the features of XML Schema, we are now in a position to make an XML Schema formalization of our RecipeML language. For comparison, we informally introduced the syntax of RecipeML in Section 2.7 and saw a DTD description in Section 4.3.6.

An XML Schema description of our recipe collections, `recipes.xsd`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.brics.dk/ixwt/recipes"
        targetNamespace="http://www.brics.dk/ixwt/recipes"
        elementFormDefault="qualified">

  <element name="collection">
    <complexType>
      <sequence>
        <element name="description" type="string"/>
        <element ref="r:recipe" minOccurs="0"
                                maxOccurs="unbounded"/>
      </sequence>
    </complexType>
    <unique name="recipe-id-uniqueness">
      <selector xpath=".//r:recipe"/>
      <field xpath="@id"/>
    </unique>
    <keyref name="recipe-references"
            refer="r:recipe-id-uniqueness">
      <selector xpath=".//r:related"/>
      <field xpath="@ref"/>
    </keyref>
  </element>

  <element name="recipe">
    <complexType>
      <sequence>
        <element name="title" type="string"/>
        <element name="date" type="string"/>
        <element ref="r:ingredient"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element ref="r:preparation"/>
        <element name="comment" type="string" minOccurs="0"/>
        <element ref="r:nutrition"/>
        <element ref="r:related" minOccurs="0"
                                 maxOccurs="unbounded"/>
      </sequence>
      <attribute name="id" type="NMTOKEN"/>
    </complexType>
  </element>
```

```
<element name="ingredient">
  <complexType>
    <sequence minOccurs="0">
      <element ref="r:ingredient"
               minOccurs="0" maxOccurs="unbounded"/>
      <element ref="r:preparation"/>
    </sequence>
    <attribute name="name" use="required"/>
    <attribute name="amount" use="optional">
      <simpleType>
        <union>
          <simpleType>
            <restriction base="r:nonNegativeDecimal"/>
          </simpleType>
          <simpleType>
            <restriction base="string">
              <enumeration value="*"/>
            </restriction>
          </simpleType>
        </union>
      </simpleType>
    </attribute>
    <attribute name="unit" use="optional"/>
  </complexType>
</element>

<element name="preparation">
  <complexType>
    <sequence>
      <element name="step" type="string"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>

<element name="nutrition">
  <complexType>
    <attribute name="calories" type="r:nonNegativeDecimal"
               use="required"/>
    <attribute name="protein" type="r:percentage" use="required"/>
    <attribute name="carbohydrates" type="r:percentage"
               use="required"/>
    <attribute name="fat" type="r:percentage" use="required"/>
    <attribute name="alcohol" type="r:percentage" use="optional"/>
  </complexType>
</element>

<element name="related">
  <complexType mixed="true">
```

```
      <attribute name="ref" type="NMTOKEN" use="required"/>
    </complexType>
  </element>

  <simpleType name="nonNegativeDecimal">
    <restriction base="decimal">
      <minInclusive value="0"/>
    </restriction>
  </simpleType>

  <simpleType name="percentage">
    <restriction base="string">
      <pattern value="([0-9]|[1-9][0-9]|100)%"/>
    </restriction>
  </simpleType>

</schema>
```

Some noteworthy comments to this schema:

- The schema is significantly wordier than the DTD version, but that is mostly due to the XML-based syntax.
- We remember to set `elementFormDefault="qualified"` to get the standard namespace semantics.
- We choose a mix of global and local descriptions where the element declarations with non-trivial types and the definitions that are used more than once are global and the remaining ones are local.
- We use `unique`, not `key`, to express the uniqueness constraint on the `id` attributes since these are optional in the `recipe` elements.
- The simple type definitions were not possible to express in DTD – the DTD version used `CDATA` types as an approximation. However, some of the simple type definitions in this example are rather clumsy, for example, it would have been convenient if we had a simpler construct for defining the type containing just the string '`*`' or if we could build unions of simple types without explicitly using the restriction/extension mechanism.
- Despite using this much more complicated schema language, we still cannot express that the `unit` attribute should only be allowed when `amount` is present, or that nested `ingredient` elements should only be allowed when `amount` is absent. Furthermore, we still cannot easily express that a `comment` element should be permitted anywhere within the other contents of a `recipe` element: at first, the `all` construct seems to be what we need, but `all` may not contain `sequence` (see page 127). (As with the DTD version, we could capture this requirement precisely with a long-winded content model that unions all the possible places where `comment` should be permitted.)

By inserting the following schema reference into the root element in our recipe collection, `recipes.xml`, we state that the document is intended to be valid according to `recipes.xsd`:

```
<collection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.brics.dk/ixwt/recipes
                                recipes.xsd"
            ... >
  ...
</collection>
```

In summary, we have now improved our description of RecipeML compared to the DTD version by formalizing the use of namespaces and providing more precise datatypes for the attribute values.

4.4.12 Limitations of XML Schema

Among the main benefits of XML Schema is support for **namespaces**, **modularization**, **datatypes**, and **type derivation**

We have earlier argued that the expressiveness of DTD for a number of reasons is insufficient. Unfortunately, it appears that XML Schema is not the ultimate solution to those problems, but let us now carefully examine the design of this language with a critical view as we did for DTD.

First of all, XML Schema certainly does improve on DTD in a number of ways, in particular regarding namespaces, modularization, and datatypes; however, there are still problems with both expressiveness and comprehensibility of the language:

1. XML Schema is generally too complicated and hard to use by non-experts. This is a problem since many non-experts need to be able to read schemas to write valid instance documents. Also, the complicated design necessitates an incomprehensible specification style, as the following randomly selected quote from the specification indicates: *'If the item cannot be ·strictly assessed·, because neither clause 1.1 nor clause 1.2 above are satisfied, [Definition:] an element information item's schema validity may be laxly assessed if its ·context-determined declaration· is not skip by ·validating· with respect to the ·ur-type definition· as per Element Locally Valid (Type) (§3.3.4).'* Obviously, only a few people will manage to read more than 100 pages of such text, but it is sadly necessary if one needs to understand the gory details of the language.

   One important factor of the complexity of the language is the type mechanism. Even without type derivations and substitution groups, this notion of types adds an extra layer of complexity: an element in the instance document has a *name*, some element declaration in the schema then assigns a *type* to this element name, and finally, some type definition then gives us the *constraints* that must be satisfied for the given element. In DTD, an element name instead directly identifies the associated constraints. (Adding to the confusion, the XML specification uses the term 'element type' for what we call the element name.)

The dual presence of the `xsi:type` attribute subsumption mechanism and the substitution group mechanism seems to reflect conflicting design approaches in the working group resulting in an unnecessarily complex specification.

2. As in DTD, element and attribute declarations are context insensitive (see Section 4.3.7). As an example, see the comments to the schema for RecipeML in Section 4.4.11. Such dependencies are quite common in XML languages, though. In fact, XML Schema itself contains numerous such dependencies. A few examples (all about element descriptions): '*`default` and `fixed` must not both be present*'; '*one of `ref` or `name` must be present, but not both*'; '*if `ref` is present, then all of <complexType>, <simpleType>, <key>, <keyref>, <unique>, `nillable`, `default`, `fixed`, `form`, `block` and `type` must be absent*'.

3. Although XML Schema itself uses an XML syntax, there is no complete schema for XML Schema (although there is an incomplete one). One reason why XML Schema is not completely self-describing is the context insensitivity mentioned in the previous point. This is, in a sense, an admission of failure: the working group has produced an XML-based language for describing syntax of XML-based languages, but it cannot describe its own syntax.

4. When describing mixed content, the character data in the contents cannot be constrained in any way. (Compare this with Problem 4 in Section 4.3.7.)

5. The notion of *unqualified local elements* is damaging to the namespace mechanism, as previously discussed.

6. A schema cannot enforce a particular root element in the instance documents (unless the schema has only one global element declaration, but that style of writing schemas is not always recommendable).

7. Element defaults cannot contain markup, only character data.

8. The use of the `xsi:type` attribute in instance documents (see Section 4.4.8) implies an unfortunate coupling between the instance document and one particular schema. It adds to the complexity of the instance documents, and also, it does not work well with multiple schemas describing different aspects of the same instance documents.

9. In Section 4.4.2, we discussed issues related to the limited flexibility of simple type definitions.

10. As mentioned on pages 127 and 128, the `all` construct is restricted in various ways, which limits its usefulness.

11. Although the overloading feature explained in Section 4.4.4 is often useful, it is restricted in two ways: first, it only works together with local definitions; second, two element declarations that have the same name and appear in the same complex type must have identical types.

Rumors are that XML Schema version 1.1 will contain some sort of conditional constraints, which might solve the problems with context insensitivity, but, presumably, new versions will not become less complicated than the current one.

4.4.13 Best Practices

One common approach to overcome the problems with XML Schema is to adhere to a set of guidelines on how the many features of XML Schema should or should not be used. Often, XML Schema allows a task to be solved in many different ways, each having different pros and cons. Of course, there are no definitive rules, but many companies and organizations define local rules that fit their particular needs.

One typical example is the collection of guidelines developed by the XML committee of the Danish Ministry of Science, Technology and Innovation, in cooperation with various other public authorities and also private companies. Some of their guidelines are:

- Always employ namespaces (using `targetNamespace`, see Section 4.4.1) when developing new XML languages.
- Never use `redefine` (see Section 4.4.7); it makes it too difficult to locate the definitions in force.
- Only use global definitions; local definitions do not promote reuse.
- Reuse existing definitions whenever possible.
- Never use `final`, `block`, `finalDefault`, and `blockDefault` since these constructs limit reuse.
- Don't use the `list` construct in simple type definitions (as we discussed in Section 4.4.2).
- Never derive complex types by restriction (see Section 4.4.3). Such derivations counter the object-oriented principle that subtypes are specializations of supertypes.
- Avoid substitution groups (see Section 4.4.8) since they tend to complicate the schema structure.
- Don't use `notation` (Section 4.4.2) or `appinfo` (Section 4.4.6); schemas should not make application specific bindings.
- Never use unqualified local elements, as we also argued in Section 4.4.5.

In addition, the guidelines contain naming conventions, similar to those available for programming languages, such as Java. More information about this project is available (in Danish) at `http://www.oio.dk/XML`. Another example of a collection of guidelines is developed by the United Kingdom e-Government Unit, having occasionally different points of view – see `http://e-government.cabinetoffice.gov.uk/Resources/Guidelines/`.

One lesson to learn from the widespread use of such guidelines is that even advanced applications can actually be developed using only a subset of the many features available in XML Schema.

4.4.14 Other Schema Languages

Unlike for many other XML technologies, it has proved difficult to reach a consensus about how a really good schema language for XML should look. There are several reasons for this situation: it appears to be an inherently difficult problem to design a schema language that is at the same time simple and highly expressive. People have different needs from a schema language. Some have a strong need for an object-oriented type system or for advanced normalization features, others have more modest requirements. Also, the official (W3C) proposals are not very good – we have seen the many problems with DTD and XML Schema.

Between DTD and XML Schema, four other schema language proposals were published as W3C notes, the experience of which were a starting point for the design of XML Schema: XML-Data, DCD (Document Content Description), DDML (Document Definition Markup Language), and SOX (Schema for Object-Oriented XML). Outside the W3C, numerous other schema languages have been developed. In the next sections, we will briefly look at two of these – DSD2 and RELAX NG.

4.5 DSD2 ★

The DSD2 (Document Structure Description 2.0) language is a successor to the DSD 1.0 language developed in cooperation by the University of Aarhus and AT&T Labs Research. The main design goals have been that this language should

- contain few and simple language constructs based on familiar concepts, such as boolean logic and regular expressions;
- be easy to understand, also by non-XML-experts; and
- have more expressive power than other schema languages for most practical purposes.

Compared to XML Schema, DSD2 is small (the specification is only 15 pages, excluding examples), and it is 100% self-describing (so there is a *complete* DSD2 schema for DSD2; see `http://www.brics.dk/DSD/dsd2.dsd`).

The central ideas in DSD2 can be summarized as follows:

- A schema consists of a list of *rules*. For every element in the instance document, all rules are processed. Rules can conditionally depend on the name, attributes, and context of the current element.
- Rules contain *declare* and *require* sections. A *declare* section specifies which contents (sub-elements and character data) and attributes that are allowed for the current

element. A *require* section specifies extra restrictions on contents, attributes, and context.

- Attribute values and element contents are described by regular expressions.
- Rule conditions and extra restrictions are described by boolean logic.

4.5.1 Recipe Collections with DSD2

With DSD2, the syntax of our recipe collection language can be expressed as follows.

```
<dsd xmlns="http://www.brics.dk/DSD/2.0"
     xmlns:r="http://www.brics.dk/ixwt/recipes"
     xmlns:c="http://www.brics.dk/DSD/character-classes"
     root="r:collection">

  <import href="http://www.brics.dk/DSD/character-classes.dsd"/>

  <if><element name="r:collection"/>
    <declare><contents>
      <sequence>
        <element name="r:description"/>
        <repeat><element name="r:recipe"/></repeat>
      </sequence>
    </contents></declare>
    <unique>
      <and><element name="r:recipe"/><attribute name="id"/></and>
      <attributefield name="id"/>
    </unique>
  </if>

  <if><element name="r:recipe"/>
    <declare>
      <attribute name="id"><stringtype ref="r:NMTOKEN"/></attribute>
      <contents>
        <sequence>
          <element name="r:title"/>
          <element name="r:date"/>
          <repeat><element name="r:ingredient"/></repeat>
          <element name="r:preparation"/>
          <element name="r:nutrition"/>
          <repeat><element name="r:related"/></repeat>
        </sequence>
        <optional><element name="r:comment"/></optional>
      </contents>
    </declare>
  </if>
```

```
<if><element name="r:ingredient"/>
  <declare>
    <required><attribute name="name"/></required>
    <attribute name="amount">
      <union>
        <string value="*"/>
        <stringtype ref="r:NUMBER"/>
      </union>
    </attribute>
    <attribute name="unit"/>
  </declare>
  <if><not><attribute name="amount"/></not>
    <require><not><attribute name="unit"/></not></require>
    <declare><contents>
      <repeat><element name="r:ingredient"/></repeat>
      <element name="r:preparation"/>
    </contents></declare>
  </if>
</if>

<if><element name="r:preparation"/>
  <declare><contents>
    <repeat><element name="r:step"/></repeat>
  </contents></declare>
</if>

<if>
  <or>
    <element name="r:step"/>
    <element name="r:comment"/>
    <element name="r:title"/>
    <element name="r:description"/>
    <element name="r:date"/>
  </or>
  <declare><contents>
    <string/>
  </contents></declare>
</if>

<if><element name="r:nutrition"/>
  <declare>
    <required>
      <attribute name="calories">
        <stringtype ref="r:NUMBER"/>
      </attribute>
      <attribute name="protein">
        <stringtype ref="r:PERCENTAGE"/>
      </attribute>
      <attribute name="carbohydrates">
```

```
        <stringtype ref="r:PERCENTAGE"/>
      </attribute>
      <attribute name="fat">
        <stringtype ref="r:PERCENTAGE"/>
      </attribute>
    </required>
    <attribute name="alcohol">
      <stringtype ref="r:PERCENTAGE"/>
    </attribute>
  </declare>
</if>

<if><element name="r:related"/>
  <declare>
    <contents><string/></contents>
    <required>
      <attribute name="ref"><stringtype ref="r:NMTOKEN"/></attribute>
    </required>
  </declare>
  <pointer><attributefield name="ref"/></pointer>
</if>

<stringtype id="r:NMTOKEN">
  <sequence>
    <repeat min="1"><stringtype ref="c:NAMECHAR"/></repeat>
  </sequence>
</stringtype>

<stringtype id="r:PERCENTAGE">
  <sequence>
    <union>
      <char min="0" max="9"/>
      <sequence>
        <char min="1" max="9"/>
        <char min="0" max="9"/>
      </sequence>
      <string value="100"/>
    </union>
    <string value="%"/>
  </sequence>
</stringtype>

<stringtype id="r:NUMBER">
  <sequence>
    <repeat min="1"><char min="0" max="9"/></repeat>
    <optional>
      <sequence>
        <string value="."/>
        <repeat min="1"><char min="0" max="9"/></repeat>
```

```
        </sequence>
      </optional>
    </sequence>
  </stringtype>

</dsd>
```

DSD2 has strong support for **conditional constraints** where declarations rely on attributes and other context.

This schema contains a number of conditional rules (the `if` elements) and some string type definitions (the `stringtype` elements). The conditional rules typically check just the name of the current element, but more complex tests can be made. The string type definitions contain commonly used regular expressions. The `import` construct is used to import a preexisting definition of the `NAMECHAR` type. All constructs are explained briefly in the following.

Notice that we use a particular namespace, `http://www.brics.dk/DSD/2.0`, to identify DSD2 elements. Annotations, such as human readable documentation, can be added using the namespace `http://www.brics.dk/DSD/2.0/meta`; such elements and attributes are ignored by the schema processor but can have meaning for other tools.

An instance document can refer to a DSD2 schema using a processing instruction before the root element:

```
<?dsd href="URI"?>
```

As in other schema languages, such a reference means that the instance document is intended to be valid relative to the given schema.

For the recipe collection example, the expressiveness of DSD2 permits us to eliminate *all* the shortcomings that we discussed for the DTD and XML Schema variants in Sections 4.3.6 and 4.4.11.

4.5.2 Rules

A *rule* defines constraints that must be satisfied for every element in the instance document for the document to be valid. Rules come in different forms:

- A *conditional* rule is an `if` element containing a boolean expression followed by a number of sub-rules. If the boolean expression evaluates to true for the current element, then the sub-rules apply.
- A *declare* rule contains declarations of attributes and contents. All attributes and contents of the current element must be declared by some rule, and conversely, all declarations must match the attributes and contents of the current element. Attributes are optional, unless declared within a `required` element.
- A *require* rule contains a boolean expression that must evaluate to true for the current element.
- *Unique* and *pointer* rules correspond to keys and references in XML Schema and are explained later.

Additionally, rules can be defined with a name and then reused multiple times, using `rule` elements with `id` and `ref` attributes.

As an example of a rule, consider the one for `recipe` elements:

```
<if><element name="r:recipe"/>
  <declare>
    <attribute name="id"><stringtype ref="r:NMTOKEN"/></attribute>
    <contents>
      <sequence>
        <element name="r:title"/>
        <element name="r:date"/>
        <repeat><element name="r:ingredient"/></repeat>
        <element name="r:preparation"/>
        <element name="r:nutrition"/>
        <repeat><element name="r:related"/></repeat>
      </sequence>
      <optional><element name="r:comment"/></optional>
    </contents>
  </declare>
</if>
```

This is a conditional rule containing a `declare` sub-rule that applies to elements named `recipe` from the recipe collection namespace. The `declare` rule contains (1) a declaration of the optional attribute `id` and specifies that the valid values are defined by the string type `NMTOKEN`, and (2) a `contents` declaration with two constituents, which are both regular expressions. The first constituent declares that the contents must consist of a sequence of a `title` element, a `date` element, any number of `ingredient` elements, and so on. (We explain the regular expression notation in a moment.) The second constituent declares that an optional `comment` element is also allowed. Whitespace between the elements is implicitly declared to be permitted (because neither regular expression mentions character data; this is explained later).

The rule-based foundation of DSD2 supports **extensibility** and **reuse** of descriptions.

The meaning of a content declaration with multiple regular expressions, such as the one above, is that each of them must match the parts of the actual contents that are mentioned in the expression. For example, the first constituent here does not mention `comment` elements, so all `comment` elements that might appear in the actual contents are ignored when considering this expression. The second constituent does mention `comment` elements, but no other elements or character data, so when considering this expression, only the `comment` elements in the actual contents are considered. And, as mentioned above, all contents must be matched by some constituent of a `contents` declaration. Another way to explain this behavior is that all applicable content declaration expressions are merged into one regular language.

This mechanism makes it straightforward to describe combinations of ordered and unordered content models: the optional `comment` element is permitted anywhere in the other contents, which are ordered. Additionally, content model descriptions can be extended simply by adding new rules to the schema, rather than being forced to modify existing ones or introducing a complicated type system into the language.

Construct	*Meaning*
`and`	conjunction
`or`	disjunction
`not`	negation
`imply`	implication
`equiv`	equivalence
`one`	exactly one subexpression is true
`parent`	subexpression true for the parent (false if at the root)
`ancestor`	subexpression true for some ancestor
`child`	subexpression true for some child
`descendant`	subexpression true for some descendant
`this`	used for uniqueness and pointer rules (explained later)
`element`	checks the name of the current element
`attribute`	checks the presence or value of an attribute in the current element
`contents`	pattern matching on the contents of the current element

Figure 4.9 Boolean expression constructs in DSD2.

4.5.3 Boolean Expressions

Boolean expressions consist of tests of element names, attribute presence and values, and content pattern matches that can be combined by the usual boolean operators, such as `and`, `or`, and `not`, as listed in Figure 4.9. Such expressions are always evaluated relative to a current element and result in either *true* or *false*. In addition, there are operators for moving to the parent, a child, ancestor, or descendant. As with most other syntactical categories, boolean expressions can be named and reused (with `boolexp` elements having `id` or `ref` attributes).

In the example schema, we particularly exploit conditional rules and boolean expressions in the description of `ingredient` elements:

```
<if><element name="r:ingredient"/>
  <declare>
    ...
  </declare>
  <if><not><attribute name="amount"/></not>
    <require><not><attribute name="unit"/></not></require>
    <declare><contents>
      ...
    </contents></declare>
  </if>
</if>
```

This construction allows us to distinguish between simple and composite ingredients based on the presence of the `amount` attribute.

Construct	*Meaning*
`sequence`	concatenation
`optional`	zero or one occurrence
`complement`	complement
`union`	union
`intersection`	intersection
`minus`	set difference
`repeat`	a number of occurrences (bounded or unbounded)
`string`	a specific or arbitrary string
`char`	a single character (in some interval)

Figure 4.10 Regular expression constructs in DSD2.

4.5.4 Regular Expressions

Regular expressions are used in DSD2 to express valid attribute values (string types) and content sequences (content types). In the former case, the alphabet consists of the Unicode characters, and in the latter case, all elements of the instance document are also included. Content of an element is here viewed as a sequence of elements and individual Unicode characters, rather than grouping together consecutive characters as in the normal XML tree model.

The notion of regular expressions is a generalized form of the one introduced in Section 4.2; a list of constructs is shown in Figure 4.10.

One example of a regular expression is the description of the values of `amount` attributes:

```
<union>
  <string value="*"/>
  <stringtype ref="r:NUMBER"/>
</union>
```

The `NUMBER` type is defined separately by a named string type.

Another example is the description of the contents of `collection` elements:

```
<sequence>
  <element name="r:description"/>
  <repeat><element name="r:recipe"/></repeat>
</sequence>
```

If character data should be permitted in this content, `string` or `char` constructs could simply be inserted in the expression.

String type expressions can be named and reused with the `stringtype` construct, as illustrated in the example. Similarly, content type expressions can be named and reused with the `contenttype` construct.

4.5.5 Normalization

DSD2 provides mechanisms for normalizing whitespace and character cases in attribute values and character data and for insertion of default attributes and contents.

Both `contents` and `attribute` declarations may contain a `normalize` directive, such as:

```
<normalize whitespace="compress" case="upper"/>
```

For the `whitespace` attribute, `compress` causes all consecutive whitespace characters to be replaced by a single space character; `trim` additionally removes leading and trailing whitespace; and `preserve` (the default) means that whitespace should be unmodified. For the `case` attribute, `upper` causes all characters to be converted to upper case, `lower` means lower case, and `preserve` means no changes.

Attribute defaults are specified within the attribute declarations:

```
<attribute name="uri">
  <default value="anonymous.jpg"/>
</attribute>
```

This declaration may be combined with the declaration of the attribute values, or specified separately.

Content defaults are similarly specified within the content declarations:

```
<contents>
  ...
  <default>no email address available</default>
</contents>
```

In contrast to XML Schema, content defaults may here contain markup, and, as with attribute defaults, the default value can be specified separately from the other aspects of the element concerned.

4.5.6 Modularization

The modularization mechanisms in DSD2 are quite simple. Schemas can be combined using `import` instructions:

```
<dsd ...>
  <import href="http://www.brics.dk/DSD/character-classes.dsd"/>
  ...
</dsd>
```

The effect is that the contents of the designated schema are inserted in place of the `import` instruction, except that repeated imports of the URI are ignored.

Together with the notion of conditional rules, this construct provides sufficient flexibility in building families of related schemas and extending basic descriptions located in other schema files. For example, one schema may import an existing one and extend the descriptions of certain elements with additional attributes or contents, simply by adding new rules in the importing schema.

4.5.7 Uniqueness and Pointers

DSD2 permits the description of uniqueness constraints and references, much in the same manner as XML Schema but without involving XPath. Instead, the boolean expression mechanism is used to select scope of uniqueness and key/reference fields. The DSD2 mechanism generalizes the corresponding features in XML Schema by not being restricted to downward axes.

This is probably the most complicated part of DSD2, so we here give just a brief introduction; for more details and examples, see the language specification.

The following uniqueness constraint appears in our schema for recipe collections in the description of `collection` elements:

```
<unique>
  <and><element name="r:recipe"/><attribute name="id"/></and>
  <attributefield name="id"/>
</unique>
```

The `unique` element contains a boolean expression followed by a number of field selectors. The effect is that whenever the DSD2 processor encounters a `collection` element in the instance document, it finds all elements in the instance document where the boolean expression evaluates to true, and then, for each of those, evaluates the field selectors with the `this` operation bound to that element to build a *key*. All keys that are built during the processing of an instance document must be unique.

The `pointer` mechanism is used in the description of `related` elements:

```
<if><element name="r:related"/>
  ...
  <pointer><attributefield name="ref"/></pointer>
</if>
```

Generally, a `pointer` element contains a optional boolean expression and a number of field selectors, essentially as `unique` rules. Each key that is produced by processing a `pointer` rule must match a key that has been produced by a `unique` rule. For the recipe example, this declaration effectively establishes references from the `related` element to the `recipe` elements.

4.6 RELAX NG ★

The RELAX NG schema language has been developed within the Organization for the Advancement of Structured Information Standards (OASIS) and is being standardized by ISO. It addresses only pure validation; RELAX NG processing has no side-effects such as normalization or post-schema-validation infoset contributions.

This language has been designed with the same fundamental goals as DSD2: simplicity and expressiveness. However, the design is quite different. DSD2 is based on a notion of

rules, which must be satisfied for every element in the instance document; RELAX NG is based on *grammars*, which is conceptually closer to the design of DTD and XML Schema.

The notion of **patterns** is central to RELAX NG.

Validation with RELAX NG proceeds as a top-down traversal of the instance document tree. To be valid, the root element must match a *pattern*, which has been specified in the schema. Generally, a pattern may match an element, an attribute, or character data. Element patterns can contain subpatterns that describe element contents and attributes, element subpatterns can themselves contain subpatterns, and so on. The pattern matching process may involve testing multiple choices, so generally, each node in the instance document may be visited many times during the traversal.

As an introductory example, the following very simple RELAX NG schema captures the structure of our business card XML language (see Section 4.4.1):

```
<element xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary=
           "http://www.w3.org/2001/XMLSchema-datatypes"
         ns="http://businesscard.org"
         name="card">
  <element name="name"><text/></element>
  <element name="title"><text/></element>
  <element name="email"><text/></element>
  <optional>
    <element name="phone"><text/></element>
  </optional>
  <optional>
    <element name="logo">
      <attribute name="uri"><data type="anyURI"/></attribute>
    </element>
  </optional>
</element>
```

The meaning of the various constructs being used here will be clear from the following tour of RELAX NG. However, one thing to note here is that the RELAX NG language is identified by the namespace

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

A RELAX NG description of the RELAX NG language itself is available at

```
http://www.relaxng.org/relaxng.rng
```

It captures most – but not all – syntactical requirements of RELAX NG schemas.

4.6.1 Patterns and Grammars

The central kinds of patterns are the following:

`element` matches one element with a given name (or a set of possible names, as described later) and with certain contents and attributes. Example:

```
<element name="title">
  <text/>
</element>
```

This particular pattern matches a `title` element that contains character data but no attributes or child elements.

`attribute` matches one attribute:

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

This pattern matches a `unit` attribute with an arbitrary value.

`text` matches any character data or attribute value. For constraining the permitted values, see the explanation of datatypes below. The default content of `attribute` patterns is a `text` pattern (so the contents of the `attribute` pattern above could be omitted).

The contents of elements can be described with – as usual – a variant of regular expressions (see Section 4.2) with the following operators:

`group`: corresponds to concatenation.

`optional`: zero or one occurrence.

`zeroOrMore`: any number of occurrences.

`oneOrMore`: one or more occurrences.

`choice`: corresponds to union.

`empty`: the empty sequence.

`interleave`: all possible mergings of the sequences that match the subexpressions. This construct has a similar effect as specifying multiple content expressions in DSD2. The `interleave` construct is more powerful than the `all` construct in XML Schema: `interleave` permits the subexpressions to describe ordered contents, which `all` does not.

`mixed`: an abbreviation for `interleave` containing a `text` pattern in addition to whatever the `mixed` element contains. This resembles the notion of mixed content models in DTD and XML Schema.

These regular expressions can contain both `element`, `attribute`, and `text` patterns. In particular, note that attributes are described in the same expressions as the element contents. However, the actual ordering of attributes in elements in the instance document is irrelevant,

as always, and no more than one attribute of a given name is permitted in any element. This means that, for example, `attribute` patterns are prohibited within `oneOrMore` patterns.

A `oneOrMore`, `zeroOrMore`, `optional`, or `mixed` pattern that has more than one subexpression is treated as one where the subexpressions are wrapped into a `group` pattern. (This also applies to `define` and `list`, which are explained later.) Note that there is no easy way of specifying, for example, exactly 7 occurrences or 3 to 5 occurrences of some element; that inevitably requires some long expressions (unlike in XML Schema and DSD2), but such constraints are not common.

Unlike DTD and XML Schema, but like DSD2, RELAX NG does not require the content expressions to be deterministic. However, heavy use of ambiguity in expressions may significantly affect performance of a RELAX NG schema processor because it then can be necessary to traverse parts of the instance document tree many times to detect whether or not there exists a right combination of choices that leads to a proof of validity.

Name Classes

As an alternative for `element` and `attribute` patterns to match elements or attributes with one specific name, the names can be described more generally with *name classes*, which occur initially in the contents of the pattern elements:

`name` specifies one specific name:

```
<element>
  <name>title</name>
  ...
</element>
```

The '. . .' abbreviates the description of the attributes and contents of the element.

`choice` describes a choice between several name classes, such as the following:

```
<element>
  <choice>
    <name>ol</name>
    <name>ul</name>
    <name>dl</name>
  </choice>
  ...
</element>
```

This pattern matches an element named `ol`, `ul`, or `dl`. (Note that in this context, `choice` is a name class description, not a pattern.)

`anyName` matches any name from any namespace:

```
<element>
  <anyName/>
  ...
</element>
```

`nsName` matches any name within a specific namespace:

```
<attribute>
  <nsName ns="http://www.brics.dk/ixwt/recipes"/>
  ...
</attribute>
```

Both `anyName` and `nsName` can be restricted using `except` constructions, as in this example:

```
<nsName ns="http://www.brics.dk/ixwt/recipes">
  <except>
    <choice>
      <name>collection</name>
      <name>title</name>
    </choice>
  </except>
</nsName>
```

This name class matches any name from the RecipeML namespace but `collection` and `title`.

Namespaces

Any RELAX NG element can have an `ns` attribute. This attribute is relevant for `name`, `nsName`, `element`, and `attribute` where it specifies the applicable namespace. For example,

```
<name ns="http://www.brics.dk/ixwt/recipes" name="recipe"/>
```

matches the name `recipe` in the namespace `http://www.brics.dk/ixwt/recipes` and only that name. Elements without this attribute implicitly inherit one from the nearest ancestor where one is present, or take the empty string as value if none is found – except for the `attribute` pattern where the namespace is always the empty string unless the `ns` attribute is explicitly specified. The namespace prefixes being chosen in the instance documents cannot be constrained, as usual.

Pattern Definitions and References

Definitions and references allow description of **recursive** structures.

The language constructs described so far only permit description of the instance document tree down to a bounded distance from the root element. However, typical XML languages allow recursion, that is, the descendants of some element named `X` may contain other elements named `X`. In RecipeML, for example, `ingredient` elements may contain other `ingredient` elements to an unbounded depth. The way to express this design in RELAX NG is to *define* named patterns and use recursive references to these named patterns within the definitions.

Typically, a RELAX NG schema has the following structure:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         ns="...">
  <start>
    ...
  </start>

  <define name="...">
    ...
  </define>

  ...
</grammar>
```

The root element of the schema is a `grammar` element that contains (1) one `start` element containing the pattern for the root of the instance documents, and (2) a number of `define` elements that define named patterns. A `ref` element is a reference to a named pattern. Notice that this design resembles the use of global and local definitions in XML Schema, although the definition mechanism here works on all kinds of patterns and there is – fortunately – no such thing as unqualified local elements. Since every XML document must contain one root element, the `start` pattern must match exactly one element of some sort.

As an example of a typical element description, we can describe the valid contents of `recipe` elements from the RecipeML language as follows:

```
<element name="recipe">
  <interleave>
    <group>
      <element name="title"><text/></element>
      <element name="date"><text/></element>
      <zeroOrMore>
        <ref name="element-ingredient"/>
      </zeroOrMore>
      <ref name="element-preparation"/>
      <ref name="element-nutrition"/>
      <zeroOrMore>
        <ref name="element-related"/>
      </zeroOrMore>
    </group>
    <optional>
      <element name="comment"><text/></element>
    </optional>
  </interleave>
</element>
```

We show a complete RELAX NG schema for RecipeML in Section 4.6.3; this schema also contains the definitions of the named patterns that are used in the fragment shown here.

As we have seen, recursion is common. However, recursive definitions in RELAX NG are allowed only if passing through an `element` pattern. That is, the following is illegal:

```
<define name="illegal_recursion">
  <element name="foo"><empty/></element>
  <optional>
    <ref name="illegal_recursion"/>
  </optional>
</define>
```

(even though it might be mathematically well-defined, describing a sequence of `foo` elements) whereas this one is legal:

```
<define name="legal_recursion">
  <element name="foo"><empty/></element>
  <optional>
    <element name="bar">
      <ref name="legal_recursion"/>
    </element>
  </optional>
</define>
```

A similar restriction holds for definitions and references in XML Schema.

Annotations

In the schemas, elements and attributes that do *not* belong to the `http://relaxng.org/ns/structure/1.0` are ignored by the schema processor. This can be used to incorporate annotations, such as human readable documentation, within the schemas. To annotate a group of definitions, these can be embedded into a `div` element, which has no other meaning than logically grouping its contents.

4.6.2 Datatypes

RELAX NG relies on external languages for describing datatypes that govern which attribute values and character data are valid. Usually, the datatype sublanguage of XML Schema is used (see Section 4.4.2), but others are possible depending on the implementation.

The `data` pattern describes text by some externally defined datatype. This pattern can be viewed as a refinement of the `text` pattern described earlier.

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

This particular pattern describes `number` elements whose contents consist of character data that matches the built-in `integer` type from XML Schema. The `datatypeLibrary` attribute identifies the datatype vocabulary being used. If this attribute is omitted, one is inherited from the nearest ancestor having one. This means that the `datatypeLibrary` can conveniently be placed in the root element of the schema if only one library is being used. The datatype facets – in case of using XML Schema datatypes – can be constrained by a parameter mechanism:

```
<element name="interval">
  <data datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
        type="integer">
    <param name="minInclusive">42</param>
    <param name="maxInclusive">87</param>
  </data>
</element>
```

This particular pattern matches `interval` elements whose content is an integer in the interval 42 to 87 (see Figure 4.7). Other datatype libraries may have different parameters for their types.

In addition to using externally defined datatypes, there are a few simple mechanisms within RELAX NG itself for defining enumeration and list types and exclusions. First, a single value can be described using the `value` pattern:

```
<attribute name="checked">
  <value>checked</value>
</attribute>
```

This pattern matches an attribute whose name and value are both `checked`.

Enumerations can be expressed by placing multiple `value` patterns within a `choice` pattern:

```
<element name="enabled">
  <choice>
    <value>true</value>
    <value>false</value>
  </choice>
</element>
```

Not surprisingly, this pattern matches an element named `enabled` whose content is either the character data `true` or `false`.

The `list` pattern defines a concatenation of datatypes where whitespace is permitted between the individual values. Example:

```
<list>
  <oneOrMore><data type="decimal"/></oneOrMore>
  <choice>
    <value>cm</value>
```

```
      <value>in</value>
    </choice>
  </list>
```

(Compare this pattern with the limitations of the `list` construct in XML Schema.) The string

```
-0.748 0.558 cm
```

matches this type, for instance.

Exclusion of certain values can be done with the `except` pattern:

```
<data type="decimal">
  <except>
    <value>0</value>
  </except>
</data>
```

This matches all strings that represent decimal numbers, except the string `0`. This example raises an important point: restricting facets in XML Schema operates on the semantic level, as noted earlier, but we have not here specified the type of '`0`'. The effect is that this example pattern matches, for example, `000` and `0.000`, which was probably not the intention. This problem is solved by adding a type qualifier:

```
<data type="decimal">
  <except>
    <value type="decimal">0</value>
  </except>
</data>
```

In contrast to the previous pattern, this one matches all strings that represent decimal numbers, except the *number* 0 which has many syntactic representations.

The datatypes `string` and `token` are built into RELAX NG and can hence be used without the XML Schema datatype library. The meanings of these two types are as in XML Schema: they both represent strings of Unicode characters, the only difference being the pruning of whitespace with the `token` type when performing pattern matching. The `token` type is the default for `value` patterns.

The language specification forbids using `data` and `value` in mixed content models, that is, ones where both character data and elements are allowed in a content sequence. This means that, as in DTD and XML Schema, there is no way of constraining the character data in such content models.

The `ID` and `IDREF(S)` types known from DTD are available through a special compatibility datatype library. For example, we can express that the `id` attributes of `recipe` elements must have unique values:

```
<element name="recipe">
  <optional>
    <attribute name="id">
      <data datatypeLibrary=
```

```
                "http://relaxng.org/ns/compatibility/datatypes/1.0"
              type="ID"/>
      </attribute>
    </optional>
    ...
  </element>
```

4.6.3 Recipe Collections with RELAX NG

With RELAX NG, the syntax of RecipeML can be expressed as follows:

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

  <start>
    <element name="collection">
      <element name="description"><text/></element>
      <zeroOrMore>
        <ref name="element-recipe"/>
      </zeroOrMore>
    </element>
  </start>

  <define name="element-recipe">
    <element name="recipe">
      <optional>
        <attribute name="id">
          <data datatypeLibrary=
                "http://relaxng.org/ns/compatibility/datatypes/1.0"
                type="ID"/>
        </attribute>
      </optional>
      <interleave>
        <group>
          <element name="title"><text/></element>
          <element name="date"><text/></element>
          <zeroOrMore>
            <ref name="element-ingredient"/>
          </zeroOrMore>
          <ref name="element-preparation"/>
          <element name="nutrition">
            <ref name="attributes-nutrition"/>
          </element>
          <zeroOrMore>
            <ref name="element-related"/>
          </zeroOrMore>
```

```
      </group>
      <optional>
        <element name="comment"><text/></element>
      </optional>
    </interleave>
  </element>
</define>

<define name="element-ingredient">
  <element name="ingredient">
    <attribute name="name"/>
    <choice>
      <group>
        <attribute name="amount">
          <choice>
            <value>*</value>
            <ref name="NUMBER"/>
          </choice>
        </attribute>
        <optional>
          <attribute name="unit"/>
        </optional>
      </group>
      <group>
        <zeroOrMore>
          <ref name="element-ingredient"/>
        </zeroOrMore>
        <ref name="element-preparation"/>
      </group>
    </choice>
  </element>
</define>

<define name="element-preparation">
  <element name="preparation">
    <zeroOrMore>
      <element name="step"><text/></element>
    </zeroOrMore>
  </element>
</define>

<define name="attributes-nutrition">
  <attribute name="calories">
    <ref name="NUMBER"/>
  </attribute>
  <attribute name="protein">
    <ref name="PERCENTAGE"/>
  </attribute>
  <attribute name="carbohydrates">
```

```
      <ref name="PERCENTAGE"/>
    </attribute>
    <attribute name="fat">
      <ref name="PERCENTAGE"/>
    </attribute>
    <optional>
      <attribute name="alcohol">
        <ref name="PERCENTAGE"/>
      </attribute>
    </optional>
  </define>

  <define name="element-related">
    <element name="related">
      <text/>
      <attribute name="ref">
        <data datatypeLibrary=
              "http://relaxng.org/ns/compatibility/datatypes/1.0"
              type="IDREF"/>
      </attribute>
    </element>
  </define>

  <define name="PERCENTAGE">
    <data type="string">
      <param name="pattern">([0-9]|[1-9][0-9]|100)%</param>
    </data>
  </define>

  <define name="NUMBER">
    <data type="decimal">
      <param name="minInclusive">0</param>
    </data>
  </define>

</grammar>
```

We define a grammar with `collection` as the only possible start element. To obtain a comprehensible structure of the schema, the most complicated element patterns are placed in separate definitions and the simple ones are inlined. For the `nutrition` element, its attributes are placed in a separate definition to permit easy extensibility, as we shall see later. The last two definitions contain datatypes that are used multiple times in other patterns. The names of the definitions have been chosen to convey their meaning informally and have no formal meaning.

The language features we have seen so far are powerful enough to describe certain conditional constraints, that is, validity constraints that depend on the presence and values of other elements and attributes. The definition above named `element-ingredient` for `ingredient` elements illustrate this ability. It contains a `choice` of two `group` patterns.

The first matches if the element contains an `amount` attribute whose value is either `*` or a `NUMBER` and an optional `unit` attribute but nothing else; the second pattern matches if there are no attributes but zero or more `ingredient` elements followed by one `preparation` element. In other words, the content model depends on the presence of the attributes. Generally, arbitrarily deep dependencies can be expressed in this way. This clearly goes beyond the possibilities of XML Schema. Compared to DSD2, however, certain dependencies can be rather cumbersome to express in RELAX NG; for example, if a content model of some element depends on the presence of an attribute in a distant ancestor element, then the pattern for that ancestor has to branch into separate subpatterns to 'remember' the relevant attributes until the element is encountered.

4.6.4 Modularization

The modularization mechanisms that we describe next are among the more complicated parts of RELAX NG. First, the `externalRef` pattern allows individual patterns to be stored in separate files. For example, assuming that `date.rng` is the URI of an XML file containing some pattern, then

```
<externalRef href="date.rng"/>
```

can be inserted in other schemas to use that pattern.

An entire grammar may generally be used as a pattern: a grammar is simply equivalent to its start pattern. This means that grammars may be embedded within other grammars. The scope of definitions within a grammar is limited to that grammar, excluding any embedded grammars. A special reference, `parentRef`, behaves as `ref` but refers to a definition in the immediately enclosing grammar instead of the current one.

Using the `include` construct, grammars stored in different files can be merged such that named definitions can be reused. Assume that `some_definitions.rng` contains the following:

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

  <define name="PERCENTAGE">
    <data type="string">
      <param name="pattern">([0-9]|[1-9][0-9]|100)%</param>
    </data>
  </define>

  <define name="NUMBER">
    <data type="decimal">
      <param name="minInclusive">0</param>
    </data>
  </define>

</grammar>
```

These definitions can then be included in another schema:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <include href="some_definitions.rng"/>
  ...
</grammar>
```

The effect is simply that the included definitions are available in the including schema. As in other schema languages, this mechanism makes it possible to reuse definitions and build families of related schemas from common modules.

The `include` element may itself contain `start` and `define` constructs that then override any included ones with the same names – much like `redefine` in XML Schema. As an alternative or supplement to replacing definitions entirely, the new definitions can be *combined* with the imported ones having the same names, either through a `choice` or a `interleave` operation. For example, we can make an extension of the RecipeML schema to allow an additional attribute in the `nutrition` elements by adding a second definition of `attributes-nutrition` and specifying `combine="interleave"`:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">

  <include href="recipes.rng"/>

  <define name="attributes-nutrition" combine="interleave">
    <optional>
      <attribute name="vitamin-C">
        <ref name="NUMBER"/>
      </attribute>
    </optional>
  </define>

</grammar>
```

Both `start` and `define` may contain such a `combine` attribute, specifying how to combine with the imported definitions. Notice how this resembles the use of multiple declarations for one element in DSD2, which behaves much like combining by interleaving, except that the subexpressions of `interleave` in RELAX NG must describe disjoint sets of elements.

Finally, a special pattern `notAllowed` can be used to simulate abstract definitions that must be redefined or combined with other definitions (using `combine="choice"`) before they can be used. The `notAllowed` pattern can also be used to exclude certain choices that were declared in included grammars. This construct can be compared with the use of `require` rules in DSD2.

4.6.5 A Non-XML Syntax

RELAX NG has a **compact non-XML syntax** as an alternative to the XML syntax.

An interesting aspect of RELAX NG is that it has an alternative syntax: a non-XML syntax which is more compact than the XML syntax presented above. The underlying language is exactly the same for both syntaxes, but the non-XML variant can be easier to read and write once you have learned it.

XML syntax	*Compact syntax*
<group> *X Y* </group>	*X*, *Y*
<optional> *X* </optional>	*X*?
<zeroOrMore> *X* </zeroOrMore>	*X**
<oneOrMore> *X* </oneOrMore>	*X*+
<choice> *X Y* </choice>	*X* | *Y*
<interleave> *X Y* </interleave>	*X*&*Y*
<empty/>	empty
<mixed/>	mixed

Figure 4.11 Regular expression operators in RELAX NG.

As an example, an `element` pattern is in this syntax written

```
element name { properties }
```

where *properties* describe the attributes and contents of the element. The regular expression operators are written in a familiar notation as shown in Figure 4.11. The full specification for the entire alternative syntax can be found from the RELAX NG project home page.

The RELAX NG schema for RecipeML shown above can be expressed as follows in the alternative syntax:

```
default namespace = "http://www.brics.dk/ixwt/recipes"
datatypes d = "http://relaxng.org/ns/compatibility/datatypes/1.0"

start =
  element collection {
    element description { text },
    element-recipe*
  }

element-recipe =
  element recipe {
    attribute id { d:ID }?,
    ((element title { text },
      element date { text },
      element-ingredient*,
      element-preparation,
      element nutrition { attributes-nutrition },
      element-related*)
     & element comment { text }?)
  }

element-ingredient =
  element ingredient {
    attribute name { text },
    ((attribute amount { "*" | NUMBER },
      attribute unit { text }?)
```

```
      | (element-ingredient*, element-preparation))
  }

element-preparation =
  element preparation {
    element step { text }*
  }

attributes-nutrition =
  attribute calories { NUMBER },
  attribute protein { PERCENTAGE },
  attribute carbohydrates { PERCENTAGE },
  attribute fat { PERCENTAGE },
  attribute alcohol { PERCENTAGE }?

element-related =
  element related {
    text,
    attribute ref { d:IDREF }
  }

PERCENTAGE = xsd:string { pattern = "([0-9]|[1-9][0-9]|100)%" }

NUMBER = xsd:decimal { minInclusive = "0" }
```

If the XML version shown in Section 4.6.3 is clear, this version should be self-explanatory. Several tools exist for converting between the XML syntax and the non-XML syntax, and for converting between RELAX NG and other schema languages.

This non-XML syntax is also used in the specification of the formal semantics of XQuery (see Chapter 6).

4.7 Chapter Summary

A schema is a formal description of the syntax of an XML language. Depending on the schema language being used, schemas may also define normalization properties, for example default insertion, and assign type information to the elements and attributes in the instance documents.

This chapter has covered four schema languages with many differences but also notable similarities, in particular, the uses of regular expressions. Choosing the right schema language is a controversial issue and there exist good arguments in favor of and against each one of them – but we will not go further into that discussion here.

The DTD language is characterized by being the first schema language for XML; it is reasonably simple but does not have sufficient expressive power for many practical situations. XML Schema, on the other hand, is vastly more complicated and provides good support for

namespaces, modularization, and datatypes. Its notion of types is important for other XML technologies, such as XQuery. Essential to this type system are the distinction between simple and complex types and the derivation mechanisms: derivation by extension adds constituents to the valid documents, whereas derivation by restriction confines the set of valid documents.

The two remaining schema language alternatives that we have seen – DSD2 and RELAX NG – excel in simplicity and expressiveness but are not supported by the W3C. According to a small survey made by Oracle, two-thirds of XML programmers use XML Schema for validating XML documents, and one fourth uses DTD leaving only few percents to the alternative schema languages. Recently, however, especially RELAX NG has been gaining momentum as a significant competitor to W3C's recommendations.

4.8 Further Reading

The DTD language is defined in the XML specification [15]. The XML Schema specification is split into two parts: one describing the main structures of the language [79], and the other describing the datatypes [9]. URLs of these W3C recommendations are given below in the Online Resources section. The complexity of the XML Schema specification is demonstrated by the overwhelming errata for the first edition; we highly recommend looking for the newest revised editions. The document describing the design requirements for XML Schema is also published through the W3C [57]. The XML Schema description of XML Schema (the schema for schemas) is an appendix in Part 1 of the specification. A comparison between DTD and XML Schema is presented in the paper [8], and a formalization of idealized XML Schema appears in [74].

There exist many very general 'best practices' documents for XML Schema; however, we will not recommend any here since most of them do not recognize the versatile nature of XML and often do not substantiate their advice sufficiently. Therefore: be skeptical about such guidance! On the other hand, guidelines that are tailored towards more narrow application domains, often developed by governments and large organizations, can be quite instructive. Examples include those mentioned in Section 4.4.13.

The DSD2 specification is available from the BRICS research center [61]. Information about the first version, DSD 1.0, is presented in the article [52].

The RELAX NG specification is an OASIS Committee Specification [23] and also an ISO Draft International Standard. A book dedicated to RELAX NG is freely available online [82].

From a mathematical point of view, schema languages are known to correspond various kinds of *tree automata* [64, 66] – provided that `ID` and `IDREF` attribute types and similar features are ignored. DTD essentially corresponds to the class of *local tree grammars*. The central structure of XML Schema – including the type derivation mechanism – corresponds to the slightly more expressive notion of *single-type tree grammars* (this is closely related to the overloading feature described in Section 4.4.4). Both DSD2 and RELAX NG correspond to the even more general notion of *regular tree grammars*. Having solid mathematical foundations can make it easier to reason about the schemas and allow for more elegant and efficient implementations. The DSD2 prototype implementation exploits a classical connection between regular expressions and *finite-state automata on strings* [45]; RELAX NG

is based on a notion of *hedge automata* [63], and implementations use theories of regular expression *derivatives* to obtain efficiency [22].

4.9 Online Resources

`http://www.w3.org/TR/xml11/`
The XML 1.1 W3C recommendation, which defines the DTD language intertwined with the actual XML notation.

`http://www.w3.org/XML/Schema`
W3C's XML Schema home page, with links to tools and examples.

`http://www.w3.org/TR/xmlschema-0/`
XML Schema Part 0: Primer – a non-normative and incomplete introduction to XML Schema.

`http://www.w3.org/TR/xmlschema-1/`
XML Schema Part 1: Structures – the main part of the XML Schema language specification.

`http://www.w3.org/TR/xmlschema-2/`
XML Schema Part 2: Datatypes – describes the built-in simple types in XML Schema and the related type derivation mechanisms.

`http://www.relaxng.org/`
Home page for RELAX NG, with lots of documentation and links to software and examples.

`http://www.brics.dk/DSD/`
The DSD2 Web site, contains the DSD2 specification, an Open Source Java implementation, and a few example DSD2 schemas.

`http://xml.apache.org/xerces2-j/`
The Apache Xerces parser for Java, with support for DTD and XML Schema.

`http://www.alphaworks.ibm.com/tech/xmlsqc`
The XML Schema Quality Checker from IBM alphaWorks. A useful tool for detecting bugs and other anomalies in schemas written in XML Schema.

4.10 Exercises

The main lessons to be learned from these exercises are:

- reading and writing schemas, with a focus on DTD and XML Schema; and
- how to use schema validation tools.

Exercise 4.1 Explain the difference between *well-formed* and *valid*.

Exercise 4.2 Browse through the DTD description of XHTML 1.0 Strict (see the XHTML specification) and find the solutions to the following questions:

(a) Describe, in English, the content model of `head` elements.
(b) What are the possible attributes in `h3` elements?
(c) Is it required that `input` elements only occur inside `form` elements?
(d) Can `a` elements be nested?

Exercise 4.3 Write for each of the following schemas a valid instance document that uses all declared elements at least once:

(a) *EX*`/store.dtd` (DTD).
(b) *EX*`/movies.xsd` (XML Schema).
(c) *EX*`/medical.dsd` (DSD2).
(d) *EX*`/family.rng` (RELAX NG).

Exercise 4.4 Validate the recipe collection against its XML Schema description. Also try validating the collection extended with the ravioli recipe from Exercise 2.6. If the extension breaks validity, modify it to become valid.

TIP

In Section 7.3.3, we shall see some small but useful Java command-line tools that perform DTD and XML Schema validation.

Exercise 4.5 Continuing Exercise 2.2, write a schema for your driving directions XML language using the following schema languages:

(a) DTD.
(b) XML Schema.
(c) DSD2.
(d) RELAX NG.

TIP

When writing schemas in XML Schema, use the XML Schema Quality Checker (see the online resources) to check that your schemas are in fact legal schemas according to the XML Schema specification.
A common way to check a DTD schema is to run a DTD processor on a dummy XML document with a document type declaration that refers to the schema. (See Section 4.3.6.)

Exercise 4.6 There exists an XML Schema description of XML Schema (see Part 1 of the XML Schema specification). Discuss how useful this schema is for detecting errors in schemas written in XML Schema. Compare this with the similar situation in other schema languages.

Exercise 4.7 Assume that we are developing an XML language for geographic information. In this language, the contents of an element named `point` must consist of elements of the following names: `address`, `latitude`, `longitude`, and `note`. In every such content sequence, the element names `address`, `latitude`, and `longitude` may each occur zero or once; `latitude` may appear if and only `longitude` appears; either `address` or `latitude` must appear, but never both; and there may be any number of `note` elements anywhere in the sequence.

Using each of the following schema languages, formalize the above description of `point` elements. Make sure that your descriptions do not impose restrictions beyond those mentioned above:

(a) DTD.
(b) XML Schema.
(c) DSD2.
(d) RELAX NG.

Exercise 4.8 Consider a toy variant of XHTML named *ToyXHTML* whose syntax may be described as follows. The root element named `doc` contains a sequence of `p` and `h1` elements. An `h1` element may contain character data but no subelements. Each `p` element contains text (character data) that may be marked up with `em` and `a` elements. Every `a` element has either a `name` attribute or an `href` attribute, and the value of the latter must be a URI.

(a) Clarify ambiguities in the above description.
(b) Formalize the syntax of ToyXHTML using your favorite schema language.

Exercise 4.9 Consider the schema at `http://www.w3.org/2005/04/schema-for-xslt20.xsd`. (As the name implies, this is a description of the XSLT 2.0 language, which is the topic of Chapter 5, but for this exercise we only study the syntax of the language.)

(a) Explain, in English, the description of `comment` elements, including the types it refers to.
(b) Explain the meaning of the simple type named `modes`.
(c) Explain the content model of `function` elements.

Exercise 4.10 Consider the schema *EX*`/bizarro.xsd`. Write a valid instance document. Explain (briefly) why your instance document is valid.

Exercise 4.11 Convert the DTD schema *EX*`/moviestudio.dtd` to the following schema languages:

(a) XML Schema.
(b) DSD2.
(c) RELAX NG.

You may find tools on the Web for solving these tasks, but try manually for this exercise.

Exercise 4.12 Convert the schema *EX*`/patients.xsd` written in XML Schema to DTD. Which limitations in DTD do you encounter?

Exercise 4.13 Explain how overloading of element declarations in XML Schema may be emulated in DSD2 and RELAX NG.

Exercise 4.14 Compare the following aspects of the expressiveness of the four schema languages described in this chapter:

- Namespace support.
- Description of attribute values and character data.
- Default values of attributes and elements.
- Context sensitive descriptions of elements and attributes.
- Uniqueness and referential constraints.
- Self-describability.
- Support by W3C.

Exercise 4.15 What is a *deterministic* regular expression? (Hint: see the XML specification.) Give an example of a non-deterministic regular expression that cannot be rewritten into an equivalent deterministic one.