Document Structure Description 2.0

Anders Møller, BRICS <amoeller@brics.dk>

December 2002. Minor updates March and September 2005.

Table of Contents

1. Introduction

A schema language for XML provides a notation for defining classes of XML documents by describing syntactic requirements for their structure and contents. This document contains the specification of the Document Structure Description 2.0 (DSD2) schema language for XML. The specification describes the syntax and semantics of DSD2 and the relation between DSD2 schemas and instance documents.

A prototype implementation of a DSD2 processor and a number of example DSD2 schemas are available at http://www.brics.dk/DSD/.

2. Terminology and Data Model

All XML documents mentioned in this specification are implicitly assumed to be well-formed according to the XML 1.0 specification [XML] and to conform to XML Namespaces [XMLNS].

An XML document may be represented as an ordered *tree* structure. We use the terminology from [XML], with the following modifications: A *node* in an XML tree is either an *element*, a *character*, a *comment*, or a *processing instruction*. We assume that entity references have been expanded and hence do not occur explicitly in the trees. Also, attribute values and line breaks are normalized, as required by any XML processor according to [XML]. DTD information is not represented in the tree. Namespace declarations are not regarded as attributes. We use the term *element name* instead of element type. The terms *parent*, *ancestor*, *child*, and *descendant* have the expected meaning in the tree structure. The *ordering* of the tree nodes is defined by a preorder left-to-right traversal. The *contents* of an element consist of the sequence of its child elements and characters (ignoring comments and processing instructions). A *string* is a sequence of Unicode characters. A *whitespace* character is a Unicode character whose code point is either #x9, #xA, #xD, or #x20. To avoid confusion between attributes in the instance document and in the schema document, we use the term *property* to refer to a schema document attribute.

A DSD2 *schema* is an XML document satisfying the syntactic requirements specified in the following section. Such a schema defines a family of XML documents, which are said to be *valid* relative to that schema. Additionally, a schema may define certain normalization features, such as default insertion. An *instance document* is an XML document that is intended to be valid relative to a given schema. A *schema processor* is a tool that given a schema and an instance document checks whether or not the instance document is valid relative to the schema, and, if it is valid, normalizes the instance document according to the schema description.

The syntax of DSD2 is defined below using an extended form of BNF, essentially as described in Section 6 of [XML]. For simplicity, the syntax of empty elements is shown only in the single tag form, the order of attributes is implicitly insignificant, and single quotes may be used instead of double quotes around attribute values, as usual in XML documents. Additional syntactic restrictions are specified in notes after the grammar fragments.

3. The DSD2 Language

A DSD2 schema is a document derivable from *SCHEMA* in the following grammar. The semantics of a schema defines the associated family of valid documents and their normalization.

Schema elements are recognized by the DSD2 namespace named `http://www.brics.dk/DSD/2.0`. The choice of namespace prefix is insignificant, although the grammar shown here uses the default namespace (that is, the elements names have no prefix). Namespace declarations are not shown in the grammar. Implicitly, all schema elements are allowed to contain additional attributes and child elements having the namespace named `http://www.brics.dk/DSD/2.0/meta`; such attributes and elements and their contents are ignored by the schema processor.

The grammar uses the following terminals:

VALUE	as the symbol `AttValue` in [XML], excluding the enclosing quotes
ANYCONTENTS	as the symbol `content` in [XML]
PENAME	a string matching `(Prefix ':')? LocalPart | Prefix ':'` where `Prefix` and `LocalPart` are as in [XMLNS]
PANAME	matches same strings as *PENAME*
PREFIX	either the empty string or a string matching the symbol `NCName` in [XMLNS] different from the string `xmlns`
NUMERAL	a nonempty string of digits (Unicode code points #x30 to #x39)
CHAR	a single Unicode character

3.1 Schemas

A *schema* contains a number of *rules*, *definitions*, and *sub-schemas*:

```
SCHEMA ::= <dsd ( root="PENAME" )? >
             ( RULE | DEFINITION | SCHEMA )*
           </dsd>
```

(Sub-schemas usually result from document inclusion using `import`.)

Example 1:

The following schema describes a simple "business card markup language":

```
<dsd xmlns="http://www.brics.dk/DSD/2.0"
     xmlns:m="http://www.brics.dk/DSD/2.0/meta"
     xmlns:bc="http://www.example.org/BusinessCards"
     xmlns:c="http://www.example.org/common"
     root="bc:collection">

  <m:doc> A DSD2 schema for Business Card Markup Language. </m:doc>

  <import href="http://www.example.org/common.dsd"/>

  <stringtype id="bc:numeral">
    <repeat min="1"><char min="0" max="9"/></repeat>
  </stringtype>

  <if><element name="bc:collection"/>
    <declare>
      <contents>
        <repeat><element name="bc:card"/></repeat>
      </contents>
    </declare>
  </if>

  <if><element name="bc:card"/>
    <declare>
      <attribute name="id">
        <stringtype ref="bc:numeral"/>
        <normalize whitespace="trim"/>
      </attribute>
      <contents>
        <element name="bc:name"/>
        <optional><element name="bc:email"/></optional>
      </contents>
    </declare>
  </if>

  <if><element name="bc:name"/>
    <declare>
      <contents>
        <string/>
        <normalize whitespace="trim"/>
      </contents>
    </declare>
  </if>

  <if><element name="bc:email"/>
    <declare>
      <contents><stringtype ref="c:email"/></contents>
    </declare>
  </if>

</dsd>
```

The namespace of the described language is `http://www.example.org/BusinessCards`. The stringtype `numeral` is defined a sequences of one or more digits. A `collection` element contains `card` elements. A `card` element may have an `id` attribute that matches `numeral`, and its contents contains one `name` element and optionally, also one `email` element. This schema assumes that a definition of `email` with namespace `http://www.example.org/common` is defined by the imported schema `http://www.example.org/common.dsd` (whose contents are not shown here).

In all following examples of schema fragments, the default namespace is implicitly assumed to be `http://www.brics.dk/DSD/2.0` unless otherwise stated. Also, the prefix `x` is assumed to be declared elsewhere.

3.1.1 Normalization and Validation

Given a schema and an instance document, the instance document is *processed* in seven phases as follows:

1. *Parsing:* The schema document and the instance document are parsed. Import instructions are processed (as defined in §3.1.3). If the documents are not well-formed XML, if import processing fails, or if the schema document is not a syntactically correct DSD2 document, then this phase fails.

2. *Normalization:* Each element in the instance document is normalized (as defined in §3.6.2). The following phases operate on the normalized document.

3. *Checking root:* If the outermost `dsd` schema element contains a `root` property, then its value matches the prefixed name of the root element in the document (as defined in §3.1.4).

4. *Checking declarations:* For each element in the instance document, it is checked that all attributes and contents of that element are declared by the set of applicable `declare` rules of the schema (as defined in §3.2.1 and §3.2.2).

5. *Checking requirements:* For each element in the instance document, it is checked that the element satisfies the set of applicable `declare` and `require` rules of the schema (as defined in §3.2.1 and §3.2.3).

6. *Checking uniqueness:* For each element in the instance document, each applicable `unique` rule of the schema is checked (as defined in §3.7.2). If successful, this produces a key set, which is used in the next phase.

7. *Checking pointers:* For each element in the instance document, each applicable `pointer` rule of the schema is checked (as defined in §3.7.3) relative to the key set generated in the previous phase.

The *outcome* of the processing is either

- the result "*valid*" together with the normalized instance document, if all phases succeed,
- the result "*invalid*", if a check fails during Phase 2 to 7 (usually, but not necessarily, processors also give informative error messages if the instance document is found to be invalid), or
- the result "*parse error*", if a failure occurs during Phase 1.

A tool that performs all the processing phases described above is called a *fully validating DSD2 processor*. A tool that only performs the phases 1 through 5 is called a *weakly validating DSD2 processor*.

Example 2:

The following instance document is valid relative to the schema shown in Example 1:

```
<collection xmlns="http://www.example.org/BusinessCards">
  <card id="1">
    <name>John Doe</name>
    <email>john.doe@example.org</email>
  </card>
  <card>
    <name>Joe Smith</name>
  </card>
</collection>
```

3.1.2 Referring to Schemas in Instance Documents

An instance document may refer to a DSD2 document by containing a processing instruction of the following form in the document prolog:

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

where *URI* is a URI referring to the DSD2 document. (All other processing instructions, including `dsd` processing instructions outside the prolog, are ignored by the schema processor.)

This means that the author of the instance document has intended it to be valid relative to the designated schema (not that the instance document necessarily is valid, nor that the URI necessarily refers to a DSD2 document).

Example 3:

To refer to the schema in Example 1, which is assumed to be located at `http://www.example.org/BusinessCards.dsd`, a schema reference processing instruction can be inserted into the instance document from Example 2:

```
<?dsd href="http://www.example.org/BusinessCards.dsd"?>
<collection xmlns="http://www.example.org/BusinessCards">
  ...
</collection>
```

3.1.3 Import

Import instructions have the following form:

`<import href="`*URI*`" />`

where *URI* is a URI referring to a document to be imported.

In the parsing phase, all occurrences of `import` elements of the DSD2 namespace in both the instance document and the schema document are processed. Processing an `import` instruction is done in the same way XInclude `include` instructions are processed [XINCLUDE] with two exceptions: 1) URI references in `import` instructions cannot contain fragment identifiers (only inclusion of whole documents is allowed). 2) If multiple documents are imported, they are processed in a top-down depth-first manner. Repeated imports with the same URI are ignored, that is, `import` instructions with a URL that already has been imported are removed.

3.1.4 Prefixed Names in Attribute Values

A *prefixed element name* is an occurrence of *PENAME*. A *prefixed attribute name* is an occurrence of *PANAME*. Together these are called *prefixed names*.

Every prefixed name that contains a `Prefix` part must be in scope of a namespace declaration that declares that prefix. The namespace name bound to such a prefixed name is defined as the namespace name of that declaration. For prefixed element names without a `Prefix` part, the associated namespace name is defined as the default namespace name in scope. Prefixed attribute names without a `Prefix` part are not associated to any namespace name. (These definitions extend the standard namespace mechanism to schema attributes whose values are prefixed names.)

The name of a given element or attribute *matches* a prefixed name if the following conditions are satisfied:

- if the prefixed name has a local part (`LocalPart`), then that value is the same as the local part of the given element or attribute name; and
- if a namespace name is bound to the prefixed name (that is, the prefixed name is *not* a prefixed attribute name without a `Prefix` part), then this namespace name is the same as that of the given element or attribute.

3.2 Rules

Rules describe validity restrictions for a given element:

RULE ::= `<declare>` *DECLARATION** `</declare>`
| `<require>` *BOOLEXP** `</require>`
| `<if>` *BOOLEXP* *RULE** `</if>`
| `<rule ref="`*PENAME*`"/>`
| *UNIQUE*
| *POINTER*

DECLARATION ::= *ATTRIBUTEDECL*
| `<required>` *ATTRIBUTEDECL** `</required>`
| `<contents>` (*REGEXP* | *NORMALIZE* | *CONTENTSDEFAULT*)* `</contents>`

ATTRIBUTEDECL ::= `<attribute` (`name="`*PANAME*`"`)? (`type="`*ATTRTYPE*`"`)? `>`
(*REGEXP* | *NORMALIZE* | *ATTRIBUTEDEFAULT*)*

```
        </attribute>
```

ATTRTYPE ::= `string` | `qname` | `qaname`

Additional syntactic restrictions:

- each `attribute` and `contents` schema element can contain at most one *NORMALIZE*, at most one *ATTRIBUTEDEFAULT*, and at most one *CONTENTSDEFAULT*, and additionally, each `attribute` schema element can contain at most one *REGEXP*; and
- every `attribute` declaration that contains a `normalize`, `default`, or *REGEXP* must have specified the `name` property and its value must have a nonempty local part.

(*UNIQUE* and *POINTER* are described in §3.7; `rule` is described in §3.5)

Example 4:

The schema in Example 1 could be extended with the following rule:

```
<if><element name="bc:card"/>
  <declare>
    <attribute name="kind">
      <union><string value="simple"/></string value="complex"/></union>
      <default value="simple"/>
    </attribute>
  </declare>
  <if><attribute name="kind"><string value="complex"/></attribute>
    <declare>
      <contents>
        <optional><element name="bc:title"/></optional>
        <optional><element name="bc:homepage"/></optional>
      </contents>
    </declare>
  </if>
</if>
```

The above rule could be placed in a new schema document, which imports the one from Example 1. In this way, new schemas can be derived from existing schemas, both through extension with new declaration rules and restriction with new requirement rules. This rule extends the description of `card` elements. They may now also have a `kind` attribute with value `simple` or `complex`, where `simple` is a default. If the `kind` is `complex` for a concrete `card` element, it may also have `title` and `homepage` elements in its contents.

Note that attribute declarations describe *optional* attributes unless explicitly declared `required`, whereas contents declarations describe *required* contents unless explicitly declared `optional`. This behavior reflects the most common usage of attributes and contents.

The following example illustrates that conditionals in `if` rules may use full boolean logic:

```
<if>
  <and>
    <element name="bc:title"/>
    <parent><element name="bc:card"/></parent>
  </and>
  ...
</if>
```

The condition makes the sub-rules (which are shown as "`...`") applicable only to `title` elements whose parent is a `card` element.

3.2.1 Applicable Rules

A rule in a schema is *applicable* to a given element if for every enclosing `if` rule, the associated *BOOLEXP* evaluates to true relative to the element (as defined in §3.3.1).

3.2.2 Declared Attributes and Contents

A *declaration* is an `attribute` or `contents` schema element that is derived as *DECLARATION* or *ATTRIBUTEDECL*.

A given attribute is *declared* by an `attribute` declaration if the following conditions are satisfied:

- If the declaration contains a `name` property, then its value matches the prefixed name of the given attribute (as defined in §3.1.4);
- if the declaration contains a regular expression, then the value of the given attribute matches the expression (as defined in §3.4.3);
- if the declaration has a `type` property with value `qname` or `qaname`, then the value of the given attribute matches *PENAME*, and additionally, if the `Prefix` part is present, the attribute is in scope of a namespace declaration that declares that prefix; and
- either the declaration contains a regular expression, or it contains neither a *NORMALIZE* nor an *ATTRIBUTEDEFAULT*.

Given the contents sequence of an element, a regular expression associated to a `contents` declaration *declares* those characters and elements in the sequence that are mentioned by the regular expression (as defined in §3.4.1).

All attributes and contents of a given element are *declared* by a set of `declare` rules if

- each attribute is declared by at least one of the `attribute` declarations in the `declare` rules,
- each element appearing in the contents is declared by at least one of the regular expressions that are associated to a `contents` declaration in the `declare` rules, and
- if at least one non-whitespace character appears in the contents, then all characters appearing in the contents are declared by at least one of the regular expressions that are associated to a `contents` declaration in the `declare` rules.

Example 5:

As an example, the declarations in the second `if` rule in the schema in Example 1 declare all attributes and contents of the each `card` element in the instance document in Example 2. Note that for contents that only contain elements and whitespace character data, the character data does not need to be declared.

3.2.3 Satisfying Requirements

Given an element and a set of `declare` and `require` rules, the element *satisfies* the rules if

- each *BOOLEXP* associated to one of the `require` rules evaluates to true relative to the element (as defined in §3.3.1),
- each *REGEXP* associated to a `contents` in one of the `declare` rules matches the contents of the element (as defined in §3.4.3), and
- for each `attribute` declaration that occurs in a `required` section in one of the `declare` rules, there exists an attribute in the element such that the `attribute` declaration declares that attribute (as defined in §3.2.2).

Example 6:

Requirements can be described with full boolean logic. For example, the following rule states that there cannot be both a `number` attribute and `min` or `max` attributes:

```
<require>
  <not><and>
    <attribute name="number"/>
    <or><attribute name="min"/><attribute name="max"/></or>
  </and></not>
</require>
```

Such a rule could typically occur in a conditional rule that probes the name of the current element and declares both `number`, `min`, and `max` attributes. Note that `require` rules do not *declare* anything by themselves but only restrict what has been declared elsewhere.

The following rule states that `a` elements cannot be nested:

```
<if><element name="x:a"/>
  <require>
    <not><ancestor><element name="x:a"/></ancestor></not>
  </require>
</if>
```

3.3 Boolean Expressions

A *boolean expression* describes elements in the instance document:

```
BOOLEXP ::= <and> BOOLEXP* </and>
          | <or> BOOLEXP* </or>
          | <not> BOOLEXP </not>
          | <imply> BOOLEXP BOOLEXP </imply>
          | <equiv> BOOLEXP* </equiv>
          | <one> BOOLEXP* </one>
          | <parent> BOOLEXP </parent>
          | <ancestor> BOOLEXP </ancestor>
          | <child> BOOLEXP </child>
          | <descendant> BOOLEXP </descendant>
          | <this/>
          | <element ( name="PENAME" )? />
          | <attribute ( name="PANAME" )? > REGEXP? </attribute>
          | <contents> REGEXP* </contents>
          | <boolexp ref="PENAME"/>
```

Additional syntactic restrictions:

- this must have a unique or pointer ancestor; and
- every attribute expression that contains a *REGEXP* also contains a name property.

(boolexp is described in §3.5. this is used in §3.7.)

3.3.1 Evaluation of Boolean Expressions

A *this-binding* is either an element in the instance document or the special value *null*.

Given an element, called the *current element*, and a this-binding, a boolean expression *evaluates* to either *true* or *false* according to the following definition:

- and evaluates to true if and only if every sub-expression evaluates to true relative to the same current element and this-binding;
- or evaluates to true if and only if at least one sub-expression evaluates to true relative to the same current element and this-binding;
- not evaluates to true if and only if the sub-expression evaluates to false relative to the same current element and this-binding;
- imply evaluates to true if and only if either the first sub-expression evaluates to false or the second sub-expression evaluates to true relative to the same current element and this-binding;
- equiv evaluates to true if and only if either every sub-expression evaluates to true or every sub-expression evaluates to false relative to the same current element and this-binding;
- one evaluates to true if and only if exactly one sub-expression evaluates to true relative to the same current element and this-binding;
- parent evaluates to true if and only if there exists a parent element of the current element such that the sub-expression evaluates to true relative to that element and the this-binding;
- ancestor evaluates to true if and only if there exists an ancestor element of the current element such that

the sub-expression evaluates to true relative to that element and the this-binding;
- `child` evaluates to true if and only if there exists a child element of the current element such that the sub-expression evaluates to true relative to that element and the this-binding;
- `descendant` evaluates to true if and only if there exists a descendant element of the current element such that the sub-expression evaluates to true relative to that element and the this-binding;
- `this` evaluates to true if and only if the current element is the given this-binding;
- `element` evaluates to true if and only if the following condition is satisfied: if the `name` property is present, its value matches the prefixed name of the current element (as defined in §3.1.4);
- `attribute` evaluates to true if and only if the current element has an attribute where the following conditions are satisfied:
  - if the `name` property is present, its value matches the prefixed name of the attribute (as defined in §3.1.4), and
  - if a regular expression is specified, the value of the attribute matches that expression (as defined in §3.4.3); and
- `contents` evaluates to true if and only if the contents of the current element matches each of the associated regular expressions (as defined in §3.4.3).

Unless otherwise specified, boolean expressions are evaluated with the null this-binding. (See §3.7.)

Example 7:

The DSD2 meta-schema (see §4) contains the following rule:

```
<if><or><element name="normalize"/><element name="default"/></or>
  <require><not>
    <ancestor><and>
      <element name="if"/>
      <descendant>
        <and>
          <or>
            <element name="parent"/>
            <element name="ancestor"/>
            <element name="child"/>
            <element name="descendant"/>
            <element name="contents"/>
            <element name="boolexp"/>
          </or>
          <not><ancestor>
            <or>
              <element name="declare"/>
              <element name="require"/>
              <element name="unique"/>
              <element name="pointer"/>
            </or>
          </ancestor></not>
        </and>
      </descendant>
    </and></ancestor>
  </not></require>
</if>
```

This rule corresponds to the second additional syntactic restriction defined in §3.6 The expression uses an `ancestor` operation nested inside a `descendant` operation to find the boolean expression parts of the `if` elements in the instance document.

3.3.2 Mentioned Elements

An *alphabet* is a set of elements. Relative to an alphabet, a boolean expression *mentions* a set of elements:

- `and`, `or`, `not`, `imply`, `equiv`, and `one` mention the union of the elements mentioned by the sub-expressions;
- `parent`, `ancestor`, `child`, `descendant`, `this`, `attribute`, and `contents` mention no elements of the alphabet; and
- an `element` expression mentions each element from the alphabet on which the expression evaluates to true (as defined in §3.3.1).

(This definition is used in §3.4.1.)

3.4 Regular Expressions

Regular expressions describe sets of strings or contents sequences:

REGEXP ::= <sequence> *REGEXP** </sequence>
| <optional> *REGEXP* </optional>
| <complement> *REGEXP* </complement>
| <union> *REGEXP** </union>
| <intersection> *REGEXP** </intersection>
| <minus> *REGEXP* *REGEXP* </minus>
| <repeat ((number="*NUMERAL*")? | (min="*NUMERAL*")? (max="*NUMERAL*")?) > *REGEXP* </repeat>
| <string (value="*VALUE*")? />
| <char ((set="*VALUE*")? | min="*CHAR*" max="*CHAR*") />
| <stringtype ref="*PENAME*"/>
| <contenttype ref="*PENAME*"/>
| *BOOLEXP*

Additional syntactic restrictions: BOOLEXP and contenttype cannot be used inside a stringtype or in the *REGEXP* part of an attribute.

(stringtype and contenttype are described in §3.5)

Example 8:

Regular expressions are a well-known and powerful formalism for describing sets of sequences, like attribute values and contents sequences. The following regular expression could be used in a contents declaration to describe the valid contents of some element as sequences of elements matching elements mixed with digits and whitespace:

```
<repeat>
  <union>
    <boolexp ref="x:elements"/>
    <char min="0" max="9"/>
    <char set="&#x9;&#xA;&#xD;&#x20;"/>
  </union>
</repeat>
```

There are no restrictions on the use of the regular expression operators; for example, repeat, union, and sequence can be mixed freely. Note that the available operators include some non-standard ones, such as complement and intersection.

The following stringtype definition describes a format for valid date strings:

```
<stringtype id="x:date">
  <sequence>
    <union>
      <string value="jan"/>
      <string value="feb"/>
      ...
      <string value="dec"/>
    </union>
    <string value="-"/>
    <repeat number="2"><stringtype ref="x:digit"/></repeat>
    <string value="-"/>
    <repeat number="4"><stringtype ref="x:digit"/></repeat>
  </sequence>
</stringtype>
```

Most common formats, such as URIs, email addresses, etc. can be defined concisely by regular expressions. With the definition and inclusion mechanisms (see §3.1.1 and §3.5) libraries of often used regular expressions can be constructed and reused.

3.4.1 Mentioned Contents

Relative to an alphabet of elements, a regular expression *mentions* a set of characters and elements according to the following definition:

- `sequence`, `optional`, `complement`, `union`, `intersection`, `minus`, and `repeat` mention the union of what is mentioned by the sub-expressions;
- `string` and `char` mentions every Unicode character but no elements; and
- *BOOLEXP*: mentions a set of elements according to the definition in §3.3.2

(This definition is used in §3.4.3)

Example 9:

Relative to the alphabet consisting of the child elements, `name` and `email`, of the first `card` element in the instance document in Example 2, the simple regular expression `<optional><element name="bc:email"/></optional>`, which occurs in a contents declaration in Example 1, mentions only the `email` element. As explained in §3.4.3 this means that when checking that the contents of the `card` element matches this contents declaration, only the `email` element is considered. With this mechanism of matching against only the mentioned contents, it is simple to describe mixed ordered and unordered content models. Also, it is straightforward to inherit from existing schemas and extend existing content models with new declarations without modifying the original ones.

3.4.2 Languages of Regular Expressions

Relative to an alphabet of elements, every regular expression has an associated *language*, which is a set of contents sequences:

- `sequence`: the concatenation of the languages of the sub-expressions;
- `optional`: the union of the language of the sub-expression and the language containing just the empty sequence;
- `complement`: the complement of the language of the sub-expression relative to the given alphabet and the set of all Unicode characters;
- `union`: the union of the languages of the sub-expressions;
- `intersection`: the intersection of the languages of the sub-expressions;
- `minus`: the intersection of the language of the first sub-expression with the complement of the language of the second sub-expression;
- `repeat`: the language consisting of a number of concatenations of the language of the sub-expression (one concatenation yields the language itself, zero concatenations yield the language containing just the empty string), depending on the specified properties:
  - if `number="`*x*`"` is specified: *x* concatenations
  - if `min="`*x*`"` and `max="`*y*`"` are specified: from *x* to *y* (including both) concatenations
  - if `min="`*x*`"` is specified but `max` is not: *x* or more concatenations
  - if `max="`*y*`"` is specified but `min` is not: from zero to *y* (including *y*) concatenations
  - if neither `number`, `min`, or `max` is specified: zero or more concatenations;
- `string`: if the `value` property is specified, then the language containing just the given string; otherwise, the language of all Unicode strings;
- `char`: the strings consisting of a single character from the following set:
  - if `set="`*s*`"` is specified: all characters occurring in the string *s*
  - if `min="`*x*`"` and `max="`*y*`"` are specified: all characters between *x* and *y* (including both), according to the Unicode code point ordering
  - if neither `set` or `min` and `max` are specified: all Unicode characters;
- *BOOLEXP*: the set of elements of the alphabet for which the boolean expression evaluates to true (as defined in §3.3.1).

3.4.3 Regular Expression Matching

Given a regular expression and a contents sequence, the following steps are performed to check whether or not the sequence *matches* the expression:

1. Select the alphabet consisting of all elements that occur in the contents sequence.
2. Find the set of characters and elements that are mentioned by the expression relative to the alphabet (as defined in §3.4.1).
3. Find the sub-sequence of the contents sequence consisting of the characters and elements that occur in the mentioned set.

4. Check whether the sub-sequence is in the language of the expression relative to the alphabet (as defined in §3.4.2). If and only if this check succeeds, the contents sequence *matches* the regular expression.

(Note that Unicode strings, for instance attribute values, are a special case of contents sequences, so this definition of matching also applies to such values.)

Example 10:

The string "`jan-16-1976`" matches the definition of `date` from Example 8.

The following contents sequence matches both regular expressions in the contents declaration for `card` elements in Example 1:

```
<name>Nils Klarlund</name>
<title>Principal Technical Staff Member</title>
<address>Florham Park</address>
```

The sequence also matches the extensions in Example 4. However, none of those, in total four, regular expressions mention the `address` element, which is therefore not declared.

3.5 Definitions

Definitions allow rules and regular expressions to be named for grouping and reuse:

DEFINITION ::= `<rule id="`*PENAME*`">` *RULE** `</rule>`
| `<contenttype id="`*PENAME*`">` *REGEXP* `</contenttype>`
| `<stringtype id="`*PENAME*`">` *REGEXP* `</stringtype>`
| `<boolexp id="`*PENAME*`">` *BOOLEXP* `</boolexp>`

A *definition* is a `rule`, `contenttype`, `stringtype`, or a `boolexp` schema element that has an `id` property. A *reference* is a `rule`, `contenttype`, `stringtype`, or a `boolexp` schema element that has a `ref` property. (References are described in §3.2, §3.3 and §3.4)

Additional syntactic restrictions:

- The local part (`LocalPart`) of values of `id` and `ref` properties must be present.
- For any two definitions in a schema, their `id` properties must differ in the sense that one or both of the following conditions must be satisfied:
  - the local parts must be different; or
  - the associated namespace names must be different.
- For every reference in a schema, there must exist a definition in the same schema where
  - the local part of the `ref` property of the reference is the same as the local part of the `id` property of the definition and
  - the associated namespace name of the `ref` property of the reference is the same as the associated namespace name of the `id` property of the definition,

  and furthermore, that definition is of the same type as the reference (that is, if the reference is a `rule`, then the definition must also be a `rule`, etc.).

Example 11:

In Example 1, the `stringtype` definition of `numeral` is used to describe the valid values of the `id` attributes of `card` elements.

The `boolexp` reference to `elements` in Example 8 could be defined by:

```
<boolexp id="x:elements">
  <or>
    <element name="x:a"/>
    <element name="x:b"/>
  </or>
</boolexp>
```

3.5.1 Resolving References to Definitions

By the syntactic restrictions given above, a reference always uniquely identifies a definition of the same type. The semantics of a reference is defined as the semantics of the contents of the corresponding definition, with the following exception: If a definition directly or indirectly refers to itself (that is, if the subtree of the definition contains a reference to the same definition or to a definition that in some number of indirections refers to it) and the cycle does not include a `child`, `descendant`, or `contents` expression, then its semantics is that of the empty set of rules for a `rule` definition, the empty language for a `stringtype` or `contenttype` definition, and the constant true for a `boolexp`. (A schema processor may issue a warning if such a cyclic definition is detected.)

3.6 Normalization

Normalization declarations define how schema processors will modify whitespace and character cases and insert default attributes and contents:

NORMALIZE ::= `<normalize` (`whitespace="`*WHITESPACE*`"`)? (`case="`*CASE*`"`)? `/>`

WHITESPACE ::= `preserve` | `compress` | `trim`

CASE ::= `preserve` | `upper` | `lower`

ATTRIBUTEDEFAULT ::= `<default value="`*VALUE*`" />`

CONTENTSDEFAULT ::= `<default>` *ANYCONTENTS* `</default>`

Additional syntactic restrictions:

- every `normalize` must contain a `whitespace` or a `case` property; and
- `normalize` and `default` cannot occur inside an `if` rule whose associated boolean expression contains `parent`, `ancestor`, `child`, `descendant`, `contents`, or `boolexp`.

3.6.1 Whitespace and Case Normalization

A string or a contents sequence is *whitespace compressed* by replacing all sequences of two or more consecutive whitespace characters by a single space character (#x20).

A string or a contents sequence is *whitespace trimmed* by performing whitespace compression and removing all leading and trailing whitespace characters. (A *leading* whitespace character is a whitespace character that is not preceeded by any element or non-whitespace character. Similarly, a *trailing* whitespace character is a whitespace character that is not followed by any element or non-whitespace character.)

A string or a contents sequence is *upper-cased* by replacing each lower case character by the corresponding upper case character (according to the Unicode definition).

A string or a contents sequence is *lower-cased* by replacing each upper case character by the corresponding lower case character (according to the Unicode definition).

3.6.2 Normalization of an Element

An attribute name *matches* an `attribute` declaration that contains a `whitespace` or a `default` if the value of the `name` property of the declaration matches the prefixed name of the given attribute (as defined in §3.1.4).

An element is normalized in eight steps:

1. Find the set of declarations that are applicable to the given element (as defined in §3.2.1).
2. Perform *default attribute insertion* as follows: Consider in turn each `default` declaration occurring in an `attribute` declaration found in Step 1, in reverse order of occurrence in the schema. If the given element does not contain an attribute whose name matches the `attribute` declaration that encloses the `default`, then insert the following new attribute in the given element, according to the `attribute`

declaration:
  - The local part of the name of the new attribute is chosen as the local part of the `name` property;
  - the value of the new attribute is determined by the `value` property of the `default` declaration;
  - if the `name` property has no prefix, then the name of the new attribute also has no prefix;
  - if the `name` property has a prefix, then the prefix of the name of the new attribute is chosen as an arbitrary string that matches *PREFIX* and is not already used in a namespace declaration which has the given element in scope, and then, a new namespace declaration is inserted in the element, such that the new prefix is associated with the namespace name bound to the prefix of the value of the `name` property (as defined in §3.1.4).
3. Perform *attribute whitespace normalization* as follows: Consider in turn each attribute in the element.
   - Find the set of `normalize` declarations that have a `whitespace` property specified and occur in an `attribute` declaration found in Step 1 and where the name of the attribute matches the `attribute` declaration.
   - If that set is nonempty, consider the value of the `whitespace` property in that declaration in the set which occurs last in the schema.
     - If the value is `compress`, then perform whitespace compression of the value of the attribute (as defined in §3.6.1).
     - If the value is `trim`, then perform whitespace trimming of the value of the attribute (as defined in §3.6.1).
4. Perform *attribute case normalization* as follows: Consider in turn each attribute in the element.
   - Find the set of `normalize` declarations that have a `case` property specified and occur in an `attribute` declaration found in Step 1 and where the name of the attribute matches the `attribute` declaration.
   - If that set is nonempty, consider the value of the `case` property in that declaration in the set which occurs last in the schema.
     - If the value is `upper`, then upper-case the value of the attribute (as defined in §3.6.1).
     - If the value is `lower`, then lower-case the value of the attribute (as defined in §3.6.1).
5. Find the set of declarations that are applicable to the given element (as defined in §3.2.1). (Note that this set may have changed since Step 1 due to the attribute normalization in Step 2-4.)
6. If the contents of the element contain no non-whitespace characters and no elements, then perform *default contents insertion* as follows:
   - Find the set of `default` declarations that occur in a `contents` declaration found in Step 5.
   - If that set is nonempty, replace the contents of the given element by a copy of the contents (including all sub-trees) of that declaration in the set which occurs last in the schema. Insert appropriate namespace declarations in the contents to ensure that the inserted elements and attributes belong to the same namespaces as in the schema document.
7. Perform *contents whitespace normalization* as follows:
   - Find the set of `normalize` declarations that have a `whitespace` property specified and occur in a `contents` declaration found in Step 5.
   - If that set is nonempty, consider the value of the `whitespace` property in that declaration in the set which occurs last in the schema.
     - If the value is `compress`, then perform whitespace compression of the contents of the given element (as defined in §3.6.1).
     - If the value is `trim`, then perform whitespace trimming of the contents of the given element (as defined in §3.6.1).
8. Perform *contents case normalization* as follows:
   - Find the set of `normalize` declarations that have a `case` property specified and occur in a `contents` declaration found in Step 5.
   - If that set is nonempty, consider the value of the `case` property in that declaration in the set which occurs last in the schema.
     - If the value is `upper`, then upper-case the contents of the given element (as defined in §3.6.1).
     - If the value is `lower`, then lower-case the contents of the given element (as defined in §3.6.1).

(Note that normalization declarations may be overridden by ones occurring later in the schema.)

Each element in the instance document, including the ones inserted as default contents, is normalized exactly once. (Since the default elements are also normalized, normalization may not terminate. A schema processor may issue a warning if such an infinite default insertion is detected.)

Example 12:

The result of upper-casing the string "`–Dark––Blue–`" (space characters are here written as "`–`") is "`–DARK––BLUE–`", and the result of whitespace trimming that string is "DARK-BLUE".

The schema in Example 1 declares that whitespace should be trimmed in id attributes of card elements and in contents of name elements. The instance document

```
<collection xmlns="http://www.example.org/BusinessCards">
  <card id=" 1 ">
    <name>
      John Doe
   </name>
  </card>
</collection>
```

would be normalized to

```
<collection xmlns="http://www.example.org/BusinessCards">
  <card id="1">
    <name>John Doe</name>
  </card>
</collection>
```

Describing normalization features in the schema has two advantages compared to other approaches: It can be used to show the instance document authors where whitespace is significant, and tools that process the instance documents may assume that insignificant whitespace has been removed, defaults have been inserted, etc., which can simplify the processing.

3.7 Uniqueness and Pointers

Uniqueness and *pointer* rules specify that certain combinations of values in the instance document must be unique or must uniquely identify other parts of the document:

UNIQUE ::= `<unique` (`key="`*VALUE*`"`)? `>`
(*BOOLEXP* *FIELD** | (`<select>` *BOOLEXP* *FIELD** `</select>`)*)
`</unique>`

POINTER ::= `<pointer` (`key="`*VALUE*`"`)? `>` *BOOLEXP*? *FIELD** `</pointer>`

FIELD ::= `<attributefield name="`*PANAME*`"` (`type="`*ATTRTYPE*`"`)? `>` *BOOLEXP*? `</attributefield>`
| `<chardatafield` (`type="`*ATTRTYPE*`"`)? `>` *BOOLEXP*? `</chardatafield>`

Additional syntactic restrictions: each unique, pointer, and select must have at least one *FIELD* descendant.

3.7.1 Evaluating Fields

Given an element, called a *base element*, a *FIELD* is evaluated as follows resulting in either a string or an evaluation failure:

1. If a *BOOLEXP* is present in the *FIELD*, then check that there is exactly one element, called the *selected element*, in the instance document where the *BOOLEXP* evaluates to true with the this-binding set to the base element. If there is not exactly one such element, the evaluation fails (so the following steps are skipped). If *BOOLEXP* is absent, then the *selected element* is the same as the base element.
2. If the *FIELD* is an attributefield, the string is chosen as the value of that attribute in the selected element whose name matches the name property (as defined in §3.1.4). If no such attribute exists, the evaluation fails.
3. If the *FIELD* is a chardatafield, the string is chosen as the concatenation of the characters that occur in the contents of the selected element.
4. The string is subsequently normalized by trimming whitespace as described in §3.6.1. (This normalization does not change the instance document.)
5. If the *FIELD* has a type property with value qname or qaname, then perform the following extra steps:
   - Check that the computed string matches *PENAME*. If not, the evaluation fails.

- If the value of the `type` property is `qname` or the computed string has a prefix, then replace the prefix by the associated namespace name (as defined in §3.1.4). (This does not change the instance document.) If the prefix is undeclared, the evaluation fails. (If the string has no prefix and the value of the `type` property is `qname`, then the default namespace name in scope followed by a colon (':') is prepended to the string. If the string has no prefix and the `type` property has the value `qaname`, then the string is unmodified in this step. This reflects the difference between unqualified element names and unqualified attribute names.)

The resulting string constitutes the result of the evaluation.

3.7.2 Checking Uniqueness

A `unique` rule is checked as follows, relative to a given element:

1. For each `select` part, perform the following steps using the boolean expression and *FIELD* list from the `select` part. If no `select` part is present, use the single boolean expression and *FIELD* list from the `unique` rule instead.
   1. Find the set of elements in the instance document where the boolean expression evaluates to true with the this-binding set to the given element (as defined in §3.3.1).
   2. For each of those elements, called the *base element*, perform the following steps:
      - Relative to the base element, construct a list of strings with one string for each *FIELD* (in the order of occurrence) as defined in §3.7.1. If that fails, the uniqueness check fails (so the following steps are skipped).
      - Construct a *key triple* consisting of
        - the base element,
        - the value of the `key` property (or the empty string if the `key` property is absent), and
        - the string list (called the *key value*).
2. Check that each of the string lists constructed in the previous step (for this particular `unique` rule and given element) is unique. If not, the uniqueness check fails.

When all `unique` rules have been checked for all elements in the instance document, a set of key triples, called the *key set*, has been produced. (This key set is used in §3.7.3when checking pointer rules.)

Example 13:

The following rule could be added to the schema in Example 1:

```
<unique>
  <and><element name="bc:card"/><attribute name="id"/></and>
  <attributefield name="id"/>
</unique>
```

This means that the `id` attributes in `card` elements in the instance document must have unique values.

With the following example, all `id1` and `id2` attributes must have unique values and all `id3` attributes must have unique values, but it is acceptable to have, for instance, the same value of an `id1` and an `id3` attribute:

```
<unique>
  <select>
    <attribute name="id1"/>
    <attributefield name="id1"/>
  </select>
  <select>
    <attribute name="id2"/>
    <attributefield name="id2"/>
  </select>
</unique>

<unique>
  <attribute name="id3"/>
  <attributefield name="id3"/>
</unique>
```

A more complex example:

```
<if><element name="x:inventory"/>
  <unique>
    <and>
      <element name="x:category"/>
      <ancestor><this/></ancestor>
    </and>
    <chardatafield>
      <and>
        <element name="x:product"/>
        <ancestor><this/></ancestor>
      </and>
    </chardatafield>
    <chardatafield>
      <and>
        <element name="x:manufacturer"/>
        <ancestor><this/></ancestor>
      </and>
    </chardatafield>
  </unique>
</if>
```

This means: For each `inventory` element, the combination of the character data in the `product` and `manufacturer` elements that appear in `category` elements in the `inventory` element must be unique. Note that `category` elements that occur in different `inventory` elements may have the same values of `product` and `manufacturer` without violating this rule. It is assumed that another rule requires each `category` element to have exactly one `product` and one `manufacturer` descendant element.

3.7.3 Checking Pointers

A `pointer` rule is checked as follows, relative to a given element and a key set:

1. Find the set of elements, called the *candidate elements*, in the instance document where the boolean expression associated to the `pointer` evaluates to true with the this-binding set to the given element (as defined in §3.3.1). If no boolean expression is specified, all elements in the instance document are candidate elements.
2. Relative to the given element, construct a list of strings with one string for each *FIELD* occurring in the `pointer` rule (in the order of occurrence) as defined in §3.7.1. If that fails, the pointer check fails (so the following steps are skipped).
3. Check that the key set contains exactly one key triple consisting of
   - a candidate element,
   - the value of the `key` property (or the empty string if the `key` property is absent), and
   - the string list.

   If the check succeeds, the given element is said to *point to* the candidate element. (Schema processors may conveniently convey this information to tools that subsequently process the normalized document; however, such a mechanism is left unspecified here.) If the key triple is not found, the pointer check fails.

Example 14:

Assume that the schema in Example 1 also described `cardref` elements with `idref` attributes. Continuing Example 13, we could then add the following rule:

```
<if><element name="bc:cardref"/>
  <pointer>
    <element name="bc:card"/>
    <attributefield name="idref"/>
  </pointer>
</if>
```

This means that the `idref` attribute of every `cardref` element must match the key value of a `card` element.

The following rule describes how `categoryref` elements refer to `category` elements (see Example 13):

```
<if><element name="x:categoryref"/>
  <pointer>
    <and>
      <element name="x:category"/>
```

```
      <ancestor>
        <and>
          <element name="x:inventory"/>
          <descendant><this/></descendant>
        </and>
      </ancestor>
    </and>
    <attributefield name="x:product"/>
    <attributefield name="x:manufacturer"/>
  </pointer>
</if>
```

Each `categoryref` element is assumed to contain a `product` and a `manufacturer` attribute (these are declared elsewhere). The pointer rule states that the values of these attributes must match a `category` element occurring in the same `inventory` element.

Example 15:

The `key` property can be used to further restrict pointers:

```
<unique key="section_id">
  <element name="x:section"/>
  <attributefield name="id"/>
</unique>

<unique key="section_name">
  <element name="x:section"/>
  <attributefield name="name"/>
</unique>

<if><element name="x:nameref"/>
  <pointer key="section_name">
    <element name="x:section"/>
    <attributefield name="ref"/>
  </pointer>
</if>
```

Here, the `ref` attribute in `nameref` elements must match the `name` attribute of a `section` element, ignoring the `id` attribute.

4. Meta-Schema (Non-Normative)

The URL http://www.brics.dk/DSD/dsd2.dsd refers to a DSD2 description of the DSD2 language itself. (The document includes http://www.brics.dk/DSD/character-classes.dsd containing some commonly used character classes.) This schema is *sound and complete* in the sense that an XML document is valid relative to this schema if and only if it is a syntactically correct DSD2 document.

5. References

[XML]
"Extensible Markup Language (XML) 1.0 Specification (Second Edition)", T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, 6 October 2000.

[XMLNS]
"Namespaces in XML", T. Bray, D. Hollander, A. Layman, 14 January 1999.

[XINCLUDE]
"XML Inclusions (XInclude) Version 1.0", J. Marsh, D. Orchard, 21 February 2002.