INTERNATIONAL STANDARD

ISO/IEC 19757-5

First edition
2011-10-01

Information technology — Document Schema Definition Languages (DSDL) —

Part 5: Extensible Datatypes

Technologies de l'information — Langages de définition de schéma de documents (DSDL) —

Partie 5: Types de données extensibles

Reference number
ISO/IEC 19757-5:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19757-5 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 34, *Document description and processing languages*.

ISO/IEC 19757 consists of the following parts, under the general title *Information technology — Document Schema Definition Languages (DSDL)*:

— *Part 1: Overview*

— *Part 2: Regular-grammar-based validation — RELAX NG*

— *Part 3: Rule-based validation — Schematron*

— *Part 4: Namespace-based Validation Dispatching Language (NVDL)*

— *Part 5: Extensible Datatypes*

— *Part 7: Character Repertoire Description Language (CREPDL)*

— *Part 8: Document Semantics Renaming Language (DSRL)*

— *Part 9: Namespace and datatype declaration in Document Type Definitions (DTDs)*

— *Part 11: Schema Association*

Introduction

This part of ISO/IEC 19757 specifies a powerful, XML-based language which enables users to create and extend their own libraries of datatypes using straightfoward declarative XML constructs. Such libraries are well-suited to being used in pipelining validation processes in conjunction with other XML schema languages.

Unlike W3C Schema[1], ISO/IEC 19757-2:2008 (RELAX NG) does not itself provide a declarative mechanism for users to define their own datatypes. If they are not satisfied with the two built-in types of `string` and `token`, RELAX NG users have had either to use a pre-written library bundled with their validator, or to program a datatype library using that validator's API. Such programmed datatype libraries are hard to construct for non-programmer users, and built-in datatype libraries are often insufficient for users' needs.

Information technology — Document Schema Definition Languages (DSDL) —

Part 5:
Extensible Datatypes

1 Scope

This part of ISO/IEC 19757 specifies an XML language that allows users to create and extend datatype libraries for their own purposes. The datatype definitions in these libraries can be used by XML validators and other tools to validate content and make comparisons between values.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IETF RFC 3023, *XML Media Types*, Internet Standards Track Specification, January 2001, http://www.ietf.org/rfc/rfc3023.txt

IETF RFC 3987, *Internationalized Resource Identifiers (IRIs)*, Internet Standards Track Specification, January 2005, http://www.ietf.org/rfc/rfc3987.txt

ISO/IEC 19757-2:2008, *Information technology — Document Schema Definition Language (DSDL) — Part 2: Regular-grammar-based validation — RELAX NG*

W3C XML, *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, W3C Recommendation, 16 August 2006, edited in place 29 September 2006, http://www.w3.org/TR/2006/REC-xml-20060816

W3C XML Names, *Namespaces in XML 1.0 (Third Edition)*, W3C Recommendation, 8 December 2009, http://www.w3.org/TR/2009/REC-xml-names-20091208/

W3C XPath 2.0, *XML Path Language (XPath) 2.0*, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xpath20-20070123/

W3C XPath 2.0 Functions, *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xpath-functions-20070123/

W3C XSLT 2.0, *XSL Transformations (XSLT) Version 2.0*, W3C Recommendation, 23 January 2007, http://www.w3.org/TR/2007/REC-xslt20-20070123/

W3C XLink 1.0, *XML Linking Language (XLink) Version 1.0*, W3C Recommendation, 27 June 2001, http://www.w3.org/TR/2001/REC-xlink-20010627/

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
candidate value
some character data in an XML document that is to have its datatype tested

3.2
datatype
named property of sequences of character data that validate against datatype definitions as described by this part of ISO/IEC 19757

3.3
datatype definition
formal specification of constraints upon XML character data for the datatype being defined

3.4
datatype library
collection of datatype definitions that share the same XML Namespace

3.5
Extensible Datatypes document
XML document which is valid to the normative schema presented in this part of ISO/IEC 19757, and which conforms to its provisions

3.6
forwards-compatible mode
operating mode in which an implementation ignores language constructs which are labelled as having a version later than that understood by the implementation, unless they are explicitly labelled as requiring processing

3.7
implementation
Extensible Datatypes implementation that conforms to this part of ISO/IEC 19757

3.8
extended implementation
implementation that conforms to this part of ISO/IEC 19757, and which provides additional functionality provided by the extension mechanisms of Extensible Datatypes

4 Extensible Datatypes schema overview

The schema for Extensible Datatypes is interspersed as fragments within the narrative text of this part of ISO/IEC 19757 and appears rendered against a grey background. The schema language used is the compact syntax of RELAX NG, as defined by Annex C of ISO/IEC 29500:2008.

Concatenating the schema fragments in this part of ISO/IEC 19757 gives a RELAX NG schema that normatively defines the grammar of Extensible Datatypes. The consolidated schema is shown in Annex A.

NOTE 1 Throughout, as per ISO/IEC 19757-2:2008, RELAX NG compact syntax keywords used as identifiers in the schema are prefixed with the "\" character.

NOTE 2 The null Namespace is bound to the prefix `"local"` so that it can be referenced later in the schema.

```
default namespace dt =
          "http://purl.oclc.org/dsdl/extensible-datatypes"
namespace local = ""
```

Datatype libraries are defined in ISO/IEC 19757-2:2008 as being identified by an IRI, with each datatype within a given datatype library being identified by a NCName. An Extensible Datatypes document presents one or more such datatype libraries to implementations. Each datatype definition has a qualified name; the Namespace IRI identifies the datatype library to which the datatype belongs, and the local part identifies the name of the datatype within that datatype library.

5 Common constructs

5.1 Common types

5.1.1 XPath expressions

W3C XPath 2.0 expressions are used to bind values to variables or properties and to express tests in conditions. Conforming implementations of Extensible Datatypes shall additionally implement the following functions from W3C XSLT 2.0:

— `document` (W3C XSLT 2.0, section 16.1)

— `format-number` (W3C XSLT 2.0, section 16.4)

— `function-available` (W3C XSLT 2.0, section 18.1.1)

These functions shall be callable within implementations using unqualified function names.

NOTE The `function-available` function can be used to test for the availability of XPath extension functions for the purposes of enhanced validation in extended implementations.

```
XPath = text
```

The context node for evaluating XPath expressions in Extensible Datatypes is a text node that is the only child of a root node, and whose value is the whitespace-normalized candidate value. The context position and context size are both 1. The set of variable bindings are the in-scope variables, as defined at 9.4.1. The set of namespace declarations that are in-scope for the expression are those that are in-scope for the element on which the XPath is given.

5.1.2 Boolean values

Where a boolean value is to be specified, the literal strings "`true`" and "`false`" are used.

```
boolean = "true" | "false"
```

5.1.3 Regular expressions

Regular expressions are defined in W3C XPath 2.0.

```
regular-expression = text
```

5.1.4 Arbitrary content

Extensible Datatypes document are governed by an open schema which, for purposes of extensibility, allows arbitrary content to occur at certain points. Such content can be any XML content other than elements or attributes associated with the Extensible Datatypes XML Namespace.

```
anything =
 mixed {
   element * - dt:* {
     attribute * - dt:* { text }*,
     anything
   }*
 }
```

5.2 Common attributes

5.2.1 `version` attribute

The value of the `version` attribute specifies the version of Extensible Datatypes being used within the element on which it occurs. The version described by this International Standard is "1.0"

5.2.2 `ns` attribute

The value of the `ns` attribute specifies the Namespace IRI of those datatypes defined within that element whose `name` attribute does not include a prefix, thus determining the datatype library to which these datatypes belong. This value shall be an IRI as defined by IETF RFC 3987.

```
ns = attribute ns { text }
```

5.2.3 `name` attribute

The `name` attribute specifies the name of a datatype, parameter, variable or property. The value of a `name` attribute is a qualified name. If no prefix is specified the Namespace IRI associated with the name depends on the element on which the `name` attribute occurs. If the `name` attribute occurs on a `datatype` element, the Namespace IRI is that given in the `ns` attribute of the `datatype` element or its nearest ancestor element that has a `ns` attribute, if there is one, or no Namespace IRI if there is not. Otherwise, the unprefixed name has no Namespace IRI.

NOTE Names conform to the `text` pattern defined by ISO/IEC 29500-2:2008 and so are less constrained that names in some other schema specifications.

```
name = attribute name { text }
```

5.2.4 Extension attributes

Extension attributes are attributes in any non-null Namespace other than the Extensible Datatypes namespace. They can occur on any Extensible Datatypes element. The presence of such attributes shall not change the behaviour of the Extensible Datatypes elements defined in this part of ISO/IEC 19757: an implementation shall return the same result whether extension attributes are processed or not.

```
extension-attribute = attribute * - (local:* | dt:*) { text }
```

5.3 Extension elements

Extension elements are elements in any Namespace other than the Extensible Datatypes Namespace. They may be processed by extended implementations. There are three classes of extension element:

— top-level extension elements, which occur as children of the document element or `div` elements

— definition extension elements, which occur as children of `datatype` elements

— binding extension elements, which occur wherever a value can be bound (for example, to a variable)

```
extension-element =
  element * - dt:* {
    must-implement?,
    attribute * - ( dt:* | must-implement) { text }*,
    anything
  }
```

5.4 Versioning and compatibility

An Extensible Datatypes element is processed in forwards-compatible mode if it, or its nearest ancestor that has a `version` attribute, has a `version` attribute with a value greater than "1.0". When an element in the Extensible Datatypes namespace that is not described by this part of ISO/IEC 19757 is processed in forwards-compatible mode it, its attributes and its descendants shall be ignored unless it has a `must-implement` attribute with the value `true`, in which case an implementation shall halt and emit an error message.

```
must-implement = attribute must-implement { boolean }
```

6 Simplification

Before it is applied for validation, an Extensible Datatypes document is simplified into a single logical unit by processing any `include` elements and resolving any multiple occurrences of same-named data types into a single definition.

6.1 Include elements

`include` elements reference other Extensible Datatypes datatype libraries. They import datatypes from these libraries or redefine them using definitions in the host document.

```
\include =
      element include {
      ns?,
      attribute href { text },
      extension-attribute*,
      top-level-element*
      }
```

The `ns` attribute on `include` is used to override the namespace of imported datatypes as defined using the `ns` attribute in the document being included.

The `href` attribute specifies an IRI reference. This IRI reference is first transformed by escaping disallowed characters as specified in Section 5.4 of W3C XLink 1.0. If it is not absolute, the IRI reference is resolved into an absolute form as described in section 5 of IETF RFC 3987 using the base IRI of the `include` element.

The value of the `href` attribute is thus used to create a `datatypes` element, as follows. The IRI reference consists of the IRI itself and an optional fragment identifier. The resource identified by the IRI is retrieved. The result is a MIME entity: a sequence of octets labeled with a MIME media type. The media type determines how an element is constructed from the MIME entity and optional fragment identifier. When the media type is `application/xml` or `text/xml`, the MIME entity shall be parsed as an XML document in accordance with the applicable RFC (at the time of writing [RFC 3023]) and an element, which shall be a `datatypes` element in the Extensible Datatypes namespace, constructed from the result of the parse. In particular, the `charset` parameter shall be handled as specified by the RFC. This specification does not define the handling of media types other than `application/xml` and `text/xml`. The `href` attribute shall not include a fragment identifier unless the registration of the media type of the resource identified by the attribute defines the interpretation of fragment identifiers for that media type.

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

The `datatypes` element thus determined by the `href` attribute value is processed such that its `include` elements are resolved. It is not permitted for this to result in a loop. In other words, the `datatypes` element shall not require the dereferencing of an `include` element with an `href` attribute with the same value. This results in a number of datatype definitions. If the `include` element contains any `datatype` elements then for every datatype defined within the `include` element, there shall be a datatype definition in the referenced library with the same name. All datatype definitions from the referenced datatype library with the same name as a datatype definition within the `include` element are ignored.

The `include` element is treated the same as a `div` element with the same attributes, except for the `href` attribute. The first child of the equivalent `div` element is another `div` element whose attributes and children are the same as those on the referenced `datatypes` element, with the exception of those that are overridden by definitions within the `include` element as defined above. The remaining children of the equivalent `div` are the children of the `include` element.

6.2 Same-named datatypes

```
\combine = attribute combine { "choice" | "all" }
```

If, as a result of an inclusion (as described in section 6.1) or otherwise, two or more datatype definitions have the same expanded qualified name, they are combined together. For any name, there shall not be more than one `datatype` element with that name that does not have a `combine` attribute. For any name, if there is a `datatype` element with that name that has a `combine` attribute with the value `"choice"`, there shall be no `datatype` element with that name that has a `combine` attribute with the value `"all"`. Thus, for any name, if there is more than one `datatype` element with that name, then there is a unique value for the `combine` attribute for that name. After determining this unique value, the `combine` attributes are removed. If both `datatype` elements have a `param` element with the same name, those `param` elements shall specify the same type and value. Definitions are combined until there is exactly one `datatype` element for each name.

EXAMPLE The following two definitions for the a colour specification

```
<datatype name="colour" combine="choice">
  <regex>#([0-9A-Fa-f]{2})([0-9A-Fa-f]{2})([0-9A-Fa-f]{2})</regex>
  <property name="red" type="hexByte" select="$_1"/>
  <property name="green" type="hexByte" select="$_2"/>
  <property name="blue" type="hexByte" select="$_3"/>
</datatype>

<datatype name="colour" combine="choice">
  <regex>#([0-9A-Fa-f])([0-9A-Fa-f])([0-9A-Fa-f])</regex>
  <property name="red" type="hexByte" select="concat($_1,$_1)"/>
  <property name="green" type="hexByte" select="concat($_2,$_2)"/>
  <property name="blue" type="hexByte" select="concat($_3,$_3)"/>
</datatype>
```

are, by the combination process, combined into a single definition equivalent to:

```
<datatype name="colour">
  <choice>
    <all>
      <regex>#([0-9A-Fa-f]{2})([0-9A-Fa-f]{2})([0-9A-Fa-f]{2})</regex>
      <property name="red" type="hexByte" select="$_1"/>
      <property name="green" type="hexByte" select="$_2"/>
      <property name="blue" type="hexByte" select="$_3"/>
    </all>
    <all>
      <regex>#([0-9A-Fa-f])([0-9A-Fa-f])([0-9A-Fa-f])</regex>
      <property name="red" type="hexByte" select="concat($_1,$_1)"/>
```

```
      <property name="green" type="hexByte" select="concat($_2,$_2)"/>
      <property name="blue" type="hexByte" select="concat($_3,$_3)"/>
    </all>
  </choice>
  </datatype>
```

If the `choice` attribute specified `"all"` then all definitions apply, so the two definitions

```
<datatype name="pricing-currency" combine="all">
  <regex>[A-Z]{3}</regex>
</datatype>

<datatype name="pricing-currency" combine="all">
  <regex>[A-Z]{3}</regex>
  <property name="currency-code" value="$_0"/>
  <condition test="$currency-code='EUR' or $currency-code='USD'"/>
</datatype>
```

are, by the combination process, combined into a single definition equivalent to:

```
<datatype name="pricing-currency">
  <all>
    <all>
      <regex>[A-Z]{3}</regex>
    </all>
    <all>
      <regex>[A-Z]{3}</regex>
      <property name="currency-code" value="$_0"/>
      <condition test="$currency-code='USD' or $currency-code='GBP'"/>
    </all>
  </all>
</datatype>
```

7 Document element

The document element of an Extensible Datatypes document is `datatypes`. It has a required `version` attribute (see 5.2.1) and an optional `ns` attribute (see 5.2.2).

```
start = \datatypes

\datatypes =
   element datatypes {
     version, ns?, extension-attribute*, top-level-element*
   }

version = attribute version { "1.0" }
```

8 Top-level elements

Top-level elements occur as children of the document element.

```
top-level-element = \include | named-datatype | \div | extension-top-level-element
```

8.1 `div` element

`div` elements are used to partition a datatype library.

NOTE Their use is equivalent to that of `div` elements in ISO/IEC 19757-2:2008.

```
\div =
  element div {
    ns?, version?, extension-attribute*, top-level-element*
  }
```

8.2 Top-level extension elements

Top-level extension elements can be used to hold content that is used within the datatype library (such as code lists used to test enumerated values), documentation, or other information that is used by extended implementations. For example, an extension top-level element can be used by an extended implementation to define extension functions (using XSLT, for example) that can be used in the XPath expressions used within the datatype library.

```
extension-top-level-element = extension-element
```

Top-level extension elements are treated in the same way as other extension elements (see 5.3).

9 Datatype definition

Datatype definitions within a datatype library can be either named or anonymous.

9.1 Named datatypes

Named datatypes definitions for a datatype library are specified at the top level of the datatype library document using `datatype` elements. Each named datatype definition has a name specified in the `name` attribute that uniquely identifies it (see 5.2.3).

```
named-datatype =
  element datatype {
    name,
    ns?,
    preprocess?,
    combine?,
    extension-attribute*,
    param*,
    datatype-definition-element*
  }
```

9.2 Anonymous datatypes

Anonymous datatypes (see 9.4.1.5) are used to define datatypes that cannot be referred to by name.

```
anonymous-datatype =
  element datatype {
    preprocess?, extension-attribute*, datatype-definition-element*
  }
```

9.3 Whitespace processing

The `normalize-whitespace` attribute determines how a candidate value is whitespace-normalized prior to testing against the datatype definition elements. If `normalize-whitespace` has the value `preserve` no whitespace normalization is carried out. If `normalize-whitespace` has the value `replace` all whitespace characters (spaces, tabs, newlines and carriage returns) are replaced by a single space character (U+0020). Otherwise (if `normalize-whitespace` has the value `collapse` or is not specified), leading and trailing whitespace is removed and all internal sequences of whitespace characters are replaced by a single space character.

```
preprocess =
  attribute normalize-whitespace { "preserve" | "replace" | "collapse" }
```

9.4 Mechanisms for defining datatypes

A datatype definition consists of a number of elements that test values and define variables and properties. If a candidate value obeys the constraints specified by these elements, then it is a valid value for the datatype.

```
datatype-definition-element =
      property
      | variable
      | regex
      | \list
      | condition
      | valid
      | except
      | choice
      | all
      | extension-definition-element
```

9.4.1 Properties, variables and parameters

`param`, `property` and `variable` declare variables, and are thus known as variable-binding elements. The name of the variable is specified in the `name` attribute of the variable-binding element (see 5.2.3). The scope of a variable binding is the following siblings of the variable binding element and their descendants.

NOTE A variable or property specified within a `choice` is not available outside that `choice`.

9.4.1.1 Properties

The `property` element specifies a property of a candidate value. When a candidate value is validated against a datatype, it is associated with a name/type/value triple for each property. Two candidate values are considered to be equal if they have the same name/type/value triple. Equality in property values is evaluated based on the type of the property.

If only one property is specified for a candidate value, then it may have no name, in which case the `name` attribute is to be omitted. If more than one property is specified for a candidate value then all properties shall specify names.

If no properties are assigned to a candidate value by a datatype definition, then it is assigned a name/type/value triple of `('', '', val)` where *val* is the whitespace-normalized candidate value.

```
property =
  element property { name?, type?, binding, extension-attribute* }
```

EXAMPLE Consider:

```
<datatype name="color">
  <choice>
    <all>
      <regex ignore-regex-whitespace="true" case-insensitive="true">
        #([0-9A-F]{2})([0-9A-F]{2})([0-9A-F]{2})
      </regex>
      <property name="red" type="hexByte" select="$_1"/>
      <property name="green" type="hexByte" select="$_2"/>
      <property name="blue" type="hexByte" select="$_3"/>
    </all>
    <all>
      <regex case-insensitive="true">white</regex>
      <property name="red" type="hexByte" value="FF" />
      <property name="green" type="hexByte" value="FF" />
      <property name="blue" type="hexByte" value="FF" />
    </all>
  …
  </choice>
</datatype>
```

The candidate value `WHITE` will be assigned the name/type/value triples `(('red', 'hexByte', 'FF'), ('green', 'hexByte', 'FF'), ('blue', 'hexByte', 'FF'))`. The candidate value `#FFFFFF` will be assigned the same name/type/value triples; thus, the two values will be judged to be equal.

9.4.1.2 Variables

The `variable` element binds a value to a variable. Variables function identically to properties, except that their values are not used when judging equality.

NOTE Variables can used for intermediate calculations, for example for enhancing the readability of datatype definitions.

```
variable =
  element variable { name, type?, binding, extension-attribute* }
```

9.4.1.3 Parameters

When a candidate value is assessed against a datatype, a number of parameter values can be specified. The `param` elements within a `datatype` element specify which parameters may be assigned values, and the default values for those parameters that are not assigned values. The values that have been assigned to parameters are available through variable bindings within the datatype definition. If no binding is specified for a parameter, its value is the empty string.

```
param =
  element param { name, type?, binding?, extension-attribute* }
```

9.4.1.4 Value specifiers

There are two ways to specify a selected value for a property, variable or parameter, or when testing the validity of a value: through a `value` attribute, which holds a literal value, or through a `select` attribute, which holds a W3C XPath 2.0 expression. Implementations may define their own extension binding elements to provide a selected value. If a type is specified (see 9.4.1.5) then the selected value shall be valid against that type.

```
binding = ( literal-value | select ), extension-binding-element*
```

If a `value` attribute is specified, the supplied text becomes the value of the property, variable or parameter.

```
literal-value = attribute value { text }
```

If a `select` attribute value is specified, the expression it contains is evaluated (see 5.1.1). If a type is specified (see 9.4.1.5) then the selected value is the string value of the result; otherwise, it is the result of evaluating the expression.

```
select = attribute select { XPath }
```

Extension binding elements can be used to provide alternative methods (such as MathML or XSLT) for specifying the value of a parameter, property or variable. If an implementation does not support any of the extension binding elements specified, it has to assign to the variable the selected value specified by the `value` or `select` attribute instead. If an implementation supports more than one of the extension binding elements, it shall use the first extension binding element it implements to calculate the value of the variable.

```
extension-binding-element = extension-element
```

9.4.1.5 Type specifiers

There are two ways to specify a type: via a `type` attribute and a number of parameter bindings, or via an anonymous `datatype` element. If parameters are specified for the type, the datatype definition for that type shall include those parameters.

```
type =
          (attribute type { text },
          param*)
          | anonymous-datatype
```

If no type is specified for a variable, parameter or property, the type used is the XPath type of the selected value (string, number, boolean or node-set). If the selected value of a property or parameter is a node-set, then it shall be converted to a string using a mechanism identical to that of the `string()` function described in section 2.3 of W3C XPath 2.0 Functions. Parameters are not permitted to be set to numbers or booleans; if the selected value is a number or boolean, the string value of that number or boolean is used as the selected value instead.

The `type` attribute specifies a datatype by name. The value of a `type` attribute is a qualified name. If no prefix is specified then the Namespace IRI of the qualified name is that given in the `ns` attribute of the element on which the `type` attribute occurs or the nearest ancestor element that has a `ns` attribute, if there is one, or no Namespace IRI if there is not one.

The expanded qualified name given by the `type` attribute shall match the expanded qualified name of a datatype definition declared in the same Extensible Datatypes instance after any simplification (see Clause 6) has taken place.

9.4.2 Parsing

Parsing of candidate values performs two functions: it tests whether a candidate value adheres to a particular format, and may cause a number of assignments to be made for further testing or assignment to properties and variables.

9.4.2.1 Regex Parsing

The `regex` element specifies parsing using a regular expression conforming to the the W3C XPath 2.0 regular expression language. To be a valid value, the entire whitespace-normalized candidate value shall be matched by the regular expression. When performing regular expression matching, implementations shall match the longest possible strings consistent with the regular expression.

EXAMPLE 1 Given the string "FFFF" and the regular expression "([A-Z]{1,2})([A-Z]{1,2})([A-Z]{1,2})" the groups matched are "FF", "F" and "F" respectively.

NOTE 1 Although it is legal to use ^ and $ to mark the beginning and end of the matched string, it is not necessary.

The regex element provides for variable bindings corresponding to matched subexpressions in the regular expression. The name of each variable shall have the format "_{number}", where "{number}" is the one-based ordinal of the sub-expression matched. The value binding of the variable is the matched substring; the type is "". A variable named "_0" shall have the value of the entire matched string.

NOTE 2 If a subexpression matches multiple times, the last matched value shall be bound to that group's variable.

```
regex =
  element regex {
    (regex-flags & extension-attribute*), regular-expression
  }
```

EXAMPLE 2 The string "ZXC" matched by the regular expression "([A-Z])([A-Z])([A-Z])" shall cause four variable bindings such that $_1 has the value "Z", $_2 has the value "X", $_3 has the value "C" and $_0 has the value "ZXC".

9.4.2.1.1 Regular expression flags

Since the regex element always matches single strings, regular expressions are applied with the s flag (signifying "single-line" or "dot-all" mode) set to true, such that the . meta-character matches all characters, including the newline character. The m flag (signifying "multi-line" mode) is always set to false, such that the "^" meta-character matches the start of the entire string and "$" the end of the entire string.

Two attributes modify the way in which regular expressions are applied. These are equivalent to the i and x flags available within XPath 2.0.

By default, the regular expression is case sensitive. If case-insensitive="true" matching is case-insensitive. The rules for case-sensitivity are those of W3C XPath 2.0.

```
regex-flags = attribute case-insensitive { boolean }? &
 attribute ignore-regex-whitespace { boolean }?
```

By default, whitespace within the regular expression matches whitespace in the string. If ignore-regex-whitespace="true", whitespace in the regular expression is removed by the implementation prior to matching. This feature of Extensible Datatypes can be used to create more readable regular expressions.

NOTE 1 Specifying ignore-regex-whitespace="true" is not the same as datatype normalize-whitespace="collapse">...</datatype, which causes preprocessing of the candidate value itself, not the regular expression.

NOTE 2 When ignore-regex-whitespace="true" a regular expression shall use patterns not containing whitespace characters (such as "\s") to match whitespace.

EXAMPLE This regular expressions is split over three lines to aid legibility:

```
<regex ignore-regex-whitespace="true">
  ([0-9]{4})-
  ([0-9]{2})-
  ([0-9]{2})
</regex>
```

9.4.2.2 Lists

The `list` element specifies parsing of the candidate value into a list of candidate values, simply using a `separator` attribute to provide a regular expression to break up the list into items.

The `separator` attribute specifies a regular expression that matches the separators in the list. The default is `"\s+"` (one or more whitespace characters). It is an error if the regular expression matches an empty string.

```
\list =
  element list {
    attribute separator { regular-expression }?,
    extension-attribute*,
    type
  }
```

Each item in the list shall be valid against the datatype specified by the `type` attribute (and child `param` elements) or the anonymous datatype specified by the child `datatype` element. See 9.4.1.5 for more details about how the type is specified.

EXAMPLE For the Extensible Datatypes type definition

```
<list separator="\s*,\s*">
  <datatype>
    <regex>[0-9]+</regex>
  </datatype>
</list>
```

then the candidate value `1, 2, 3, 45` is valid but the candidate value `sausages, egg, chips` is not.

NOTE 1 The method by which lists items are separated follows the tokenize function as specified in W3C XPath 2.0 Functions §7.6.4.

NOTE 2 The separator text matched by the specified regular expression is discarded and cannot be used in evaluating datatype validity.

9.4.3 Testing

There are two methods of testing values: testing general conditions with the `condition` element, and testing validity against another datatype with the `valid` element.

9.4.3.1 Conditions

The `condition` element tests whether a particular condition is satisfied by a value.

```
condition = element condition { extension-attribute*, test }
```

Tests are declared with a `test` attribute which holds an XPath expression. If the effective boolean value of the result of evaluating the XPath expression is true then the test shall succeed and the condition is satisfied. The candidate value is not valid if the test of any in-scope condition evaluates to false.

```
test = attribute test { XPath }
```

EXAMPLE The following type definition defines a type that must satisfy two conditions.

```
<datatype name="short">
  <condition test=". >= -32768" />
  <condition test=". &lt;= 32767" />
</datatype>
```

9.4.3.2 Validity tests

The `valid` element tests whether a selected candidate value is valid against a specified datatype definition. The value and type is selected as for variables (see 9.4.1.4 and 9.4.1.5). If no value binding is specified, then the value binding is to ".".

```
valid = element valid { extension-attribute*, type, binding? }
```

EXAMPLE The following type definition defines a type that must both be valid according to another type definition (that for "int") and satisfy two conditions.

```
<datatype name="short">
  <valid type="int"/>
  <condition test=". >= -32768"/>
  <condition test=". &lt;= 32767"/>
</datatype>
```

9.4.4 Logical elements

The `choice`, `all` and `except` elements can be used to combine tests.

9.4.4.1 Choice

The `choice` element tests whether the candidate value is valid against any of the tests it contains: the candidate value is only valid if it satisfies one or more of the tests. The first test that succeeds is the one used for assigning property values to the candidate value.

```
choice =
  element choice { extension-attribute*, datatype-definition-element+ }
```

9.4.4.2 All

The `all` element groups together tests that shall be satisfied — the candidate value is only valid if it satisfies all the tests.

```
all = element all { extension-attribute*, datatype-definition-element+ }
```

9.4.4.3 Except

The `except` element contains tests that are required to evaluate negatively when applied to a candidate value. The candidate value is only valid if it does not satisfy any of the tests contained in the `except` element.

```
except =
  element except { extension-attribute*, datatype-definition-element+ }
```

NOTE Any `property` elements within an `except` element are ignored.

9.4.5 Definition extension elements

Definition extension elements can be used at any point within a datatype definition for documentation, examples and additional tests. Their behaviour is described at 5.3.

```
extension-definition-element = extension-element
```