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 3.

ISO/IEC 19757-5 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*, Subcommittee SC 34, Document Description and Processing Languages.

— *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: Datatype Library Language — DTLL*

— *Part 6: Path-based integrity constraints*

— *Part 7: Character Reportoire Description Language — CRDL*

— *Part 8: Document Schema Renaming Language — DSRL*

— *Part 9: Datatype- and namespace-aware DTDs*

— *Part 10: Validation Management*

Introduction

Unlike W3C Schema[1], ISO 19757-2:2003 (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.

Thus the primary use case for a language for datatype libraries is to enable users to construct their own datatypes without having to resort to a general-purpose programming language, or having to use pre-defined sets of datatypes which might not be suitable.

DTLL is a powerful, XML-based language which meets this need. Users can use it to construct and extend their own libraries of datatypes using straightfoward declarative XML constructs. Such libraries are well-suited to being used in pipelining validation processes.

Document Schema Definition Languages (DSDL) — Part 5: Datatypes — Datatype Library Language (DTLL)

1 Scope

This International Standard specifies a XML language that allows users to create and extend datatype libraries for their own purposes. The datatype definitions in these libraries may be used by XML validators and other tools to validate content and make comparisons between values.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 19757. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 19757 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

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 (Second Edition)*, W3C Recommendation, 16 August 2006, http://www.w3.org/TR/2006/REC-xml-names-20060816

W3C XPath 1.0, *XML Path Language (XPath)*, Version 1.0, W3C Recommendation, 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

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 XSLT 1.0, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 16 November 1999, http://www.w3.org/TR/xslt

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

W3C XLink 1.0, *XML Linking Language (XLink) Version 1.0*, W3C Recommendation, 27 June 2001, http://www.w3.org/TR/2000/REC-xlink-20010627/

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

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

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

a candidate value is said to be of a particular datatype when it obeys the constraints of a datatype definition specified using DTLL

3.3 datatype definition

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

3.4 datatype library

a collection of datatype definitions that share the same XML Namespace

3.5 DTLL document

an XML document which is valid to the normative schema presented in this international standard, and which conforms to its provisions

3.6 forwards-compatible mode

a DTLL processor operating in "forwards-compatible mode" ignores language constructs which are labelled as having a version later than that understood by the processor, unless they are explicity labelled as requiring processing.

3.7 implementation

a DTLL processor that conforms to this part of ISO/IEC 19757

3.8 extended implementation

a DTLL processor that conforms to this part of ISO/IEC 19757, and which provides additional functionality provided by the extension mechanisms of DTLL

3.9 IRI

This International Standard uses the subset of IRIs which are compatible with URIs; references to IRIs shall be taken to mean IRIs or IRI references

4 DTLL schema overview

The schema for DTLL is interspersed as fragments within the narrative text of this International Standard and appears rendered against a grey background. The schema language used is the compact syntax of RELAX NG, as defined by ISO 19757-2:2003 Amendment 1.

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

NOTE Throughout, as per ISO 19757-2:2003 Amendment 1, RELAX NG keywords used as identifiers in a RELAX NG schema are prefixed with the "\" character.

```
default namespace dt = "http://purl.oclc.org/dsdl/dtll"
namespace local = ""
```

Datatype libraries are defined in ISO 19757-2:2003 as being identified by an IRI, with each datatype within a given datatype library being identified by a NCName. A DTLL 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

XPath 1.0 expressions are used in DTLL to bind values to variables or properties and to express tests in conditions. Conforming implementations of DTLL have also to implement five XPath functions from W3C XSLT 1.0:

— `document` (W3C XSLT 1.0, section 12.1)

— `format-number` (W3C XSLT 1.0, section 12.3)

— `current` (W3C XSLT 1.0, section 12.4)

— `function-available` (W3C XSLT 1.0, section 15)

— `element-available` (W3C XSLT 1.0, section 15)

AB: There was some discussion of Namespaces for XPath functions in Montreal, but looking through the XPath 1.0 spec I can't see we need to mention this.

These five functions have to be callable within DTLL implementations using unqualified function names.

```
XPath = text
```

The context node for evaluating XPath expressions in DTLL 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

A regular expression as defined in W3C XPath 2.0.

```
regular-expression = text
```

5.1.4 Extended regular expressions

Extended regular expressions are regular expressions that can have named groups. Named groups are specified with the syntax `(?'`*`name`*`'`*`regex`*`)` where *name* is the name of that group and *regex* is the subexpression for that group. The group's name is used as a means of identifying that group for the purposes of datatype validation.

```
extended-regular-expression = text
```

5.1.5 Arbitrary content

DTLL is 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 DTLL 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 DTLL 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 has to 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 then 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, then 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.

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

5.2.4 Extension attributes

Extension attributes are attributes in any non-null Namespace other than the DTLL namespace. They can occur on any DTLL element. The presence of such attributes shall not change the behaviour of the DTLL elements defined in This International Standard.

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

5.3 Extension elements

Extension elements are elements in any Namespace other than the DTLL Namespace. 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-understand?,
    attribute * - dt:* { text }*,
    anything
  }
```

5.4 Versioning and compatibility

A DTLL 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 DTLL namespace that is not described by this part of ISO/IEC 19757 is processed in forwards-compatible mode then it, its attributes and its descendants have to be ignored unless it has a `must-understand` attribute with the value `true`, in which case a DTLL processor must halt and emit an error message.

```
must-understand = attribute dt:must-understand { boolean }
```

6 Simplification

Before it is applied for validation, a DTLL 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 DTLL 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.

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 then 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 bytes labeled with a MIME media type. The media type determines how an element is constructed from the MIME entity and optional fragment identifier. When the media type is `application/xml` or `text/xml`, the MIME entity 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 DTLL 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 { boolean }
```

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"`, then there shall not also be a `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. A pair of same-named definitions

```
<datatype name="name">
  params1
  tests1
</datatype>
<datatype name="name">
  params2
  tests2
</datatype>
```

is combined into

```
<datatype name="name">
  params
  <c>
    tests1
    tests2
  </c>
</datatype>
```

where *c* is the value of the `combine` attribute and *params* is the union of the `param` element children of the `datatype` elements. If both `datatype` elements have a `param` element with the same name, those `param` elements shall specify the same type and value.

JT: Most of this text is adapted from the same in RELAX NG. We need to define "the same" for type and value of parameters, though...

Pairs of definitions are combined until there is exactly one `datatype` element for each name.

7 Document element

The document element of a DTLL 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 19757-2:2003

```
  \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 data 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 that uniquely identifies it specified in the `name` attribute (see 5.2.3).

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

9.2 Anonymous datatypes

Anonymous datatypes 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` then no whitespace normalization is carried out. If `normalize-whitespace` has the value `replace` then all whitespace characters (spaces, tabs, newlines and carriage returns) are replaced by a space character. 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.

```
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 have to have names.

If no properties are assigned to a candidate value by a datatype definition, then it is assigned a name/type/value triple of `('', 'xpath:string', `*`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">
        #(?'RR'[0-9A-F]{2})(?'GG'[0-9A-F]{2})(?'BB'[0-9A-F]{2})
      </regex>
      <property name="red" type="hexByte" select="$RR"/>
      <property name="green" type="hexByte" select="$GG"/>
      <property name="blue" type="hexByte" select="$BB"/>
    </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.

TODO: re-explain property scoping and refer back to formal definition of scoping rules

9.4.1.2 Variables

The `variable` element binds a value to a variable. Variables are similar to properties except that their values are not used when judging equality. Variables are used for intermediate calculations.

```
  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 built-in ways to specify a *selected value* for a property, variable or parameter, or when testing the validity of a value: through the `value` attribute, which holds a literal value or through a `select` attribute, which holds an XPath expression. Implementations can also 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 has to 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.

TODO: Montreal note says to allow <value> element for literals

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

If a `select` attribute is specified, the XPath 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 XPath 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, then it has to assign to the variable the selected value specified by the `value` or `select` attribute instead. If an implementation supports one or more of the extension binding elements, then it has to use the first extension binding element it understands 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.

TODO: need to explain anonymous usage better

```
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). Properties and parameters are not to be set to node-sets; if the selected value is a node-set, then it must be converted to a string using a mechanism identical to that of the `string()` function described in section 4.2 of W3C XPath 1.0. Parameters are not permitted to be set to numbers or booleans; if the selected value is a number or boolean, then 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 has to match the expanded qualified name of a datatype.

JT: Need better wording here to say that the `type` attribute must point to a datatype defined by the DTLL document without restricting it to point to only those defined in the actual DTLL document (and those included within it). In other words, if DTLL document A includes DTLL documents B and C, then it's OK for DTLL document B to include a `type` attribute that points to a datatype defined in DTLL document C.

9.4.2 Parsing

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

9.4.2.1 Regex Parsing

The `regex` element specifies parsing via an extended regular expression language. To be a legal value, the entire whitespace-normalized candidate value has to be matched by the regular expression. (Although it is legal to use `^` and `$` to mark the beginning and end of the matched string, it is not necessary.)

The `regex` element also provides a number of variable bindings, one for each named subexpression. The name of each variable is the name of the subexpression; the value binding is the matched substring; the type is `xpath:string`

TODO: somewhere we need to forbid same-named variables etc; TODO: Montreal note says to mention built-in string type of DTLL here.

```
 regex =
   element regex {
     regex-flags*, extension-attribute*, extended-regular-expression
   }
```

EXAMPLE The regular expression:

`(?'year'?[0-9]{4})-(?'month'[0-9]{2})-(?'day'[0-9]{2})`

parsing the value:

`2003-12-19`

generates the variable bindings:

— `$year = '2003'`

— `$month = '12'`

— `$day = '19'`

9.4.2.1.1 Regular expression flags

Since the `regex` element always matches single strings, regular expressions are applied with the standard `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"` then the matching is case-insensitive, which means that the regular expression `"a"` will match the string `"A"`.

```
 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 processor prior to matching, and the expression itself must use the `\s` construct to match whitespace. This feature of DTLL can be used to create more readable regular expressions.

NOTE 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.

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

```
<regex ignore-regex-whitespace="true">
  (?'year'[0-9]{4})-
  (?'month'[0-9]{2})-
  (?'day'[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 has to 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 DTLL 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.

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. The candidate value is not valid if the test evaluates to false.

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

Tests are done through 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 succeeds and the condition is satisfied.

TODO: explain XPath flavour as per Montreal notes [maybe not necessary since we explain in early clauses]. Also, give example.

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

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 binding is specified, then the binding is to ".".

TODO: stop referring back to variables here, instead explain in full and note similarity

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

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 have to 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
```

EXAMPLE Extension definition elements can be used to hold documentation about the datatype. For example, an `eg:example` element might be used to provide example legal values of the datatype:

```
<datatype name="RRGGBBColour">
  <eg:example>#FFFFFF</eg:example>
  <eg:example>#123456</eg:example>

  <regex>#(?'RR'[0-9A-F]{2})(?'GG'[0-9A-F]{2})(?'BB'[0-9A-F]{2})</regex>
  …
</datatype>
```

Annex A
(normative)

RELAX NG schema for DTLL documents

```
default namespace dt = "http://purl.oclc.org/dsdl/dtll"
namespace local = ""

XPath = text
boolean = "true" | "false"
regular-expression = text
extended-regular-expression = text
anything =
  mixed {
    element * - dt:* {
      attribute * - dt:* { text }*,
      anything
    }*
  }
ns = attribute ns { text }
name = attribute name { text }
extension-attribute = attribute * - (local:* | dt:*) { text }
extension-element =
  element * - dt:* {
    must-understand?,
    attribute * - dt:* { text }*,
    anything
  }
must-understand = attribute dt:must-understand { boolean }
\include =
  element include {
    ns?,
    attribute href { text },
    extension-attribute*,
    top-level-element*
  }
combine = attribute combine { boolean }
start = \datatypes
\datatypes =
  element datatypes {
    version, ns?, extension-attribute*, top-level-element*
  }
version = attribute version { "1.0" }
top-level-element =
  \include | named-datatype | \div | extension-top-level-element
\div =
  element div {
    ns?, version?, extension-attribute*, top-level-element*
  }
extension-top-level-element = extension-element
named-datatype =
  element datatype {
    name,
    ns?,
    preprocess?,
    combine?,
    extension-attribute*,
    param*,
    datatype-definition-element*
  }
anonymous-datatype =
  element datatype {
    preprocess?, extension-attribute*, datatype-definition-element*
  }
```

```
preprocess =
  attribute normalize-whitespace { "preserve" | "replace" | "collapse" }
datatype-definition-element =
  property
  | variable
  | regex
  | \list
  | condition
  | valid
  | except
  | choice
  | all
  | extension-definition-element
property =
  element property { name?, type?, binding, extension-attribute* }
variable =
  element variable { name, type?, binding, extension-attribute* }
param = element param { name, type?, binding?, extension-attribute* }
binding = (literal-value | select), extension-binding-element*
literal-value = attribute value { text }
select = attribute select { XPath }
extension-binding-element = extension-element
type =
  (attribute type { text },
   param*)
  | anonymous-datatype
regex =
  element regex {
    regex-flags*, extension-attribute*, extended-regular-expression
  }
regex-flags =
  attribute case-insensitive { boolean }
  & attribute ignore-regex-whitespace { boolean }
\list =
  element list {
    attribute separator { regular-expression }?,
    extension-attribute*,
    type
  }
condition = element condition { extension-attribute*, test }
test = attribute test { XPath }
valid = element valid { extension-attribute*, type, binding? }
choice =
  element choice { extension-attribute*, datatype-definition-element+ }
all = element all { extension-attribute*, datatype-definition-element+ }
except =
  element except { extension-attribute*, datatype-definition-element+ }
extension-definition-element = extension-element
```

Bibliography

[1] *XML Schema Part 1: Structures Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[2] *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

Summary of editorial comments:

[5.1.1] XPath expressions

AB: There was some discussion of Namespaces for XPath functions in Montreal, but looking through the XPath 1.0 spec I can't see we need to mention this.

[6.2] Same-named datatypes

JT: Most of this text is adapted from the same in RELAX NG. We need to define "the same" for type and value of parameters, though...

[9.4.1.1] Properties

TODO: re-explain property scoping and refer back to formal definition of scoping rules

[9.4.1.4] Value specifiers

JT: Possibly make `value/select` optional if an extension binding element is provided with `dt:must-understand="true"`.

[9.4.1.4] Value specifiers

TODO: Montreal note says to allow <value> element for literals

[9.4.1.5] Type specifiers

TODO: need to explain anonymous usage better

[9.4.1.5] Type specifiers

JT: Need better wording here to say that the `type` attribute must point to a datatype defined by the DTLL document without restricting it to point to only those defined in the actual DTLL document (and those included within it). In other words, if DTLL document A includes DTLL documents B and C, then it's OK for DTLL document B to include a `type` attribute that points to a datatype defined in DTLL document C.

[9.4.2.1] Regex Parsing

TODO: somewhere we need to forbid same-named variables etc; TODO: Montreal note says to mention built-in string type of DTLL here.

[9.4.3.1] Conditions

TODO: explain XPath flavour as per Montreal notes [maybe not necessary since we explain in early clauses]. Also, give example.

[9.4.3.2] Validity tests

TODO: stop referring back to variables here, instead explain in full and note similarity