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

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 procedural programming language or use pre-defined sets of datatypes which might not be suitable.

Unlike W3C Schema[1], ISO 19757-2:2003 (RELAX NG) does not provide a 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 to create a datatype library which they then refer to from the schema.

Most RELAX NG validators provide built-in support for W3C Datatypes[2]. Many also support an API that allows users to develop datatype modules to define extra datatypes. But because these datatype libraries have to be programmed,t non-programmer users find them hard to construct.

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

1 Scope

This International Standard specifies an XML language that allows users to create datatype libraries. The datatype definitions in these libraries may be used by XML validators and other tools to validate content and test equality between values.

AB: need to add XPath to normative references

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 (Third Edition)*, W3C Recommendation, 30 October 2003, http://www.w3.org/TR/2003/PER-xml-20031030/

W3C XML-Names, *Namespaces in XML*, W3C Recommendation, 14 January 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114/

W3C XSLT-2.0, *XSL Transformations (XSLT) Version 2.0*, W3C Candidate Recommendation, 3 November 2005, http://www.w3.org/TR/xslt20/

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

3.5 DTLL document

an XML document with a `<dt:datatypes>` document element.

3.6 implementation

a DTLL processor that conforms to this International Standard.

3.7 extended implementation

a DTLL processor that conforms to this International Standard, and which provides additional functionality provided by the extension mechanisms of DTLL.

4 DTLL schema overview

AB: Can we avoid using the XSD datatypes in DTLL's schema?

JT: We could create a DTLL datatype library for the datatypes we need?

DTLL allows the specification of datatype libraries using an XML language. The schema for DTLL is interspersed as fragments within the narrative text of This International Standard. The schema language used is the compact syntax of RELAX NG, as defined by Annex C of ISO 19757-2:2003.

Concatenating these schema fragments gives a RELAX NG schema that normatively defines the grammar of the DTLL XML Language, and which appears in Annex A of this International Standard.

```
datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"
    default namespace dt = "http://www.dsdl.org/dtll/1.0"
    namespace local = ""
```

JT: We should **either** retain a `version` attribute to indicate the version of DTLL being used in a document **or** use the DTLL namespace to indicate the version, not both. I prefer using a `version` attribute, because it gives some hope of DTLL 1.0 implementations being able to process (in some way) DTLL 2.0 documents. That would mean changing the namespace to something like `http://www.dsdl.org/dtll`.

AB: I favour this too. Using Namespaces for versioning compatible languages is bad.

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.

JT: We could/should split this up like RELAX NG does, into one section that describes the normalization of a DTLL document (resolving `ns` attributes, `version` attributes, `<include>`s and combining datatype definitions) and one that describes the behaviour of what's left.

5 Document element

The document element of a DTLL document is `datatype-library`. It has a required `version` attribute (see 8.2.1) and an optional `ns` attribute (see 8.2.2).

AB: This hinges on what we mean by 'define'. Certainly an DTLL document may **make available** type definitions from many Namespaces by defining types and by importing/including definitions from other (or the same) Namespaces. In favour of the dataype-library name and a one-Namespace-per-instance approach is its parallels with Java/XSD and its conceptual tie-in with RELAX NG typelibs.

```
  start = datatype-library

   datatype-library = element datatype-library {
      version,
      ns?,
      extension-attribute*,
      top-level-element*
      }

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

6 Top-level elements

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

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

Note maps have been removed as per discussions in Atlanta.

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 { xs:anyURI },
               extension-attribute*,
               top-level-element*
             }
```

The `ns` attribute on `<include>` is used to override the Namespace of imported datatypes.

The `href` attribute is first transformed by escaping disallowed characters as specified in Section 5.4 of [XLink]. The IRI reference is then resolved into an absolute form as described in section 5.2 of [RFC 2396] using the base IRI of the `<include>` element.

The value of the `href` attribute is used to identify a `<datatypes>` element. This shall be done 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 term of writing [RFC 3023]) and an element, which shall be a `<datatypes>` element, 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.

The <datatypes> element referenced by the href attribute 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 div element

<div> elements are used to partition a datatype library.

JT: I commented out "and to provide a convenient means of wrapping content so that it may be included using the mechanism of the <include> element" because that's a spec-level issue rather than a user-level convenience.

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

JT: ns and version attributes have been added (back) in. We must allow <div> to support these in order to support <include>.

6.3 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 8.3).

7 Datatype definitions

Datatype definitions are named or anonymous.

7.1 Named Datatypes

Named datatypes definitions are specified at the top level of the datatype library using <datatype> elements. Each named datatype definition has a name that uniquely identifies it specified in the name attribute (see 8.2.3).

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

If 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 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.

7.2 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*
      }
```

7.3 Defining Datatypes: Common Constructs

Certain constructs and common to both names and anonymous datatypes.

7.3.1 Whitespace Handling

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 isn't specified), leading and trailing whitespace is removed and all internal sequences of whitespace characters are replaced by a single space.

JT: It would be nice to have a normative reference to this behaviour; the closest definition is in XML Schema Datatypes, for the whiteSpace facet, but perhaps we don't want to point to that.

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

7.3.2 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
datatype-definition-element |= variable
datatype-definition-element |= regex
datatype-definition-element |= \list
datatype-definition-element |= condition
datatype-definition-element |= valid
datatype-definition-element |= except
datatype-definition-element |= choice
datatype-definition-element |= all
      datatype-definition-element |= extension-definition-element
```

7.3.2.1 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 8.2.3). The scope of a variable binding is the following siblings of the variable binding element and their descendants.

7.3.2.1.1 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 are 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*
          }
```

7.3.2.1.1.1 Example: Datatype Properties

For 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.

Question: are properties available to other properties?

Answer: only via the normal variable-binding

7.3.2.1.2 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*
          }
```

I suggest dropping one of either variables of properties

JT: The distinction is (now) important in that one is used when judging equality between values and the other isn't.

7.3.2.1.3 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 of that have been assigned to parameters are available through variable bindings within the datatype definition.

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

If no binding is specified for a parameter, its value is the empty string.

7.3.2.1.4 Value Specifiers

There are two built-in ways to specify a *selected value* for a property, variable, 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 7.3.2.1.5) then the selected value has to be valid against that type.

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

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

If a `value` attribute is specified, the selected value is the value of that attribute.

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

Since we need to provide support for `org.relaxng.datatypesameValue()` (why?), do we need to specify that some value should be usable for equivalence testing?

JT: That's the job of properties.

If a `select` attribute is specified, the XPath expression it contains is evaluated (see 8.1.1). If a type is specified (see 7.3.2.1.5) 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 are used to provide alternative methods (such as MathML or XSTL) 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
```

7.3.2.1.5 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 { xs:QName }, param*
          type |= 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 the string value of the first node in the node-set is used as the selected value. 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 qualifid name is that given in the `ns` attribute of element on which the `type` attribute appears or its nearest ancestor element that has a `ns` attribute, if there is one, or no Namespace IRI if there isn't.

The expanded qualified name given by the `type` attribute has to match the expanded qualified name of a datatype.

7.3.2.2 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.

7.3.2.2.1 Regex Parsing

The `<regex>` element specifies parsing via an extended regular expression. 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 is the matched substring; the type is `xpath:string`.

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

7.3.2.2.1.1 Example

For example, the regex:

```
(?'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'`

7.3.2.2.1.2 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.

JT: I've received a suggestion to just use a `flags` attribute instead.

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 }
```

By default, whitespace within the regular expression matches whitespace in the string. If `ignore-regex-whitespace="true"`, whitespace in the regular expression is removed prior to matching, and you need to use `"\s"` to match whitespace. This can be used to create more readable regular expressions.

Does it matter that ignore-regex-whitespace pertains to how the regex is interpreted; the other attrbutes to how it is applied? Is there any merit in reflecting this in the markup somehow?

JT: There's no distinction in other programming languages, so I'm not really tempted to make one here.

```
regex-flags |= attribute ignore-regex-whitespace { boolean }
```

Boolean values are `'true'` or `'false'`, with optional leading and trailing whitespace.

```
boolean = xs:boolean { pattern = "true|false" }
```

7.3.2.2.1.2.1 Example: Ignoring Whitespace in Regular Expressions

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

7.3.2.2.2 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 (i.e. if it matches `""`).

```
\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 7.3.2.1.5 for more details about how the type is specified.

7.3.2.2.2.1 Example: Parsing Lists

For example, if you have:

```
<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.

7.3.2.3 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.

7.3.2.3.1 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.

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

When there are multiple parsing methods, some of which have failed, how do conditions apply? Should condition/property be children of <parse>, with the select operation scoped accordingly?

JT: These questions don't apply now that things are structured differently.

7.3.2.3.2 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 7.3.2.1.4 and 7.3.2.1.5).

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

7.3.2.4 Logical Elements

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

7.3.2.4.1 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+
        }
```

7.3.2.4.2 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+
        }
```

7.3.2.4.3 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.

Is not <except> just condition, but with the expression negated? So can we omit it?

JT: No, because it can contain things like `<regex>` and `<valid>`, which can't be negated with `not()`.

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

7.3.2.5 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 8.3.

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

7.3.2.5.1 Example of definition extension element usage

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>
```

8 Common Constructs

8.1 Common Types

8.1.1 XPath Expressions

XPath 1.0 expressions are used to bind values to variables or properties and to express tests in conditions.

```
XPath = text
```

The context node for evaluating the XPath expression 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 7.3.2.1. The function library is the core XPath function library. The set of namespace declarations that are in-scope for the expression are those that are in-scope on the element on which the XPath is given.

AB: The above bears on JT's question: "How should the results of a regex parse be captured (individual variables for each named subexpression, or a node tree containing them)?"

JT also asks "Should the results of a list parse be captured?"

8.1.2 Regular Expressions

A regular expression as defined in XPath 2.0

```
regular-expression = text
```

8.1.3 Extended Regular Expressions

Extended regular expressions are regular expressions that can have named subexpressions. Named subexpressions are specified with the syntax `(?'`*`name`*`'`*`regex`*`)` where *name* is name of the subexpression and *regex* is the subexpression itself.

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

8.2 Common Attributes

8.2.1 `version` attribute

The value of the `version` attribute specifies the version of DTLL being used within the element on which it is specified. The version described by this International Standard is 1.0.

An 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 International Standard 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"`.

JT: I've amended this wording. We need to allow a `version` attribute on `<div>` so that `<include>` works properly.

8.2.2 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.

8.2.3 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 of the qualified name depends on the element on which the `name` attribute appears. If the `name` attribute appears 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 isn't. Otherwise, the unprefixed qualified name has no namespace IRI.

```
          name = attribute name { xs:QName }
```

8.2.4 Extension Attributes

Extension attributes are attributes in any non-null namespace other than the DTLL namespace. They can appear 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 }
```

8.3 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 appear within the document element or `<div>` elements

— definition extension elements, which appear within `<datatype>` elements

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

An extension element can be ignored by an implementation unless it has a `dt:must-understand` attribute with the value `"true"`. It is an error if an implementation encounters an extension element with a `dt:must-understand` attribute with the value `"true"` which it does not recognise.

```
extension-element = element * - dt:* {
                      must-understand?,
                      attribute * - dt:* { text }*,
                      anything }

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

anything = mixed { element * {
                     attribute * { text }*,
                     anything }* }
```

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:

[1]

AB: need to add XPath to normative references

[4] DTLL schema overview

AB: Can we avoid using the XSD datatypes in DTLL's schema?

JT: We could create a DTLL datatype library for the datatypes we need?

[4] DTLL schema overview

JT: We should **either** retain a `version` attribute to indicate the version of DTLL being used in a document **or** use the DTLL namespace to indicate the version, not both. I prefer using a `version` attribute, because it gives some hope of DTLL 1.0 implementations being able to process (in some way) DTLL 2.0 documents. That would mean changing the namespace to something like `http://www.dsdl.org/dtll`.

AB: I favour this too. Using Namespaces for versioning compatible languages is bad.

[4] DTLL schema overview

JT: We could/should split this up like RELAX NG does, into one section that describes the normalization of a DTLL document (resolving `ns` attributes, `version` attributes, `<include>`s and combining datatype definitions) and one that describes the behaviour of what's left.

[5] Document element

JT: Given that (as explained above), a DTLL document may define many datatype libraries, I think that it's better to use the name `<datatypes>` for the document element. If we use `<datatype-library>`, there's no need to escape it in the RNC schema; "datatypes" needed to be escaped because it's a RNC keyword.

AB: This hinges on what we mean by 'define'. Certainly an DTLL document may **make available** type definitions from many Namespaces by defining types and by importing/including definitions from other (or the same) Namespaces. In favour of the dataype-library name and a one-Namespace-per-instance approach is its parallels with Java/XSD and its conceptual tie-in with RELAX NG typelibs.

[6] Top-level elements

Note maps have been removed as per discussions in Atlanta.

[6.1] Include elements

JT: This text is adapted from the equivalent text in the RELAX NG spec. Some `<xref>` or `<Xref>` elements need to be added for the external references.

[6.2] `div` element

JT: I commented out "and to provide a convenient means of wrapping content so that it may be included using the mechanism of the `<include>` element" because that's a spec-level issue rather than a user-level convenience.

[6.2] `div` element

JT: `ns` and `version` attributes have been added (back) in. We must allow `<div>` to support these in order to support `<include>`.

[7.1] 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...

[7.3.1] Whitespace Handling

JT: It would be nice to have a normative reference to this behaviour; the closest definition is in XML Schema Datatypes, for the whiteSpace facet, but perhaps we don't want to point to that.

[7.3.2.1.1.1] Example: Datatype Properties

Question: are properties available to other properties?

Answer: only via the normal variable-binding

[7.3.2.1.2] Variables

I suggest dropping one of either variables of properties

JT: The distinction is (now) important in that one is used when judging equality between values and the other isn't.

[7.3.2.1.4] Value Specifiers

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

[7.3.2.1.4] Value Specifiers

Since we need to provide support for `org.relaxng.datatypesameValue()` (why?), do we need to specify that some value should be usable for equivalence testing?

JT: That's the job of properties.

[7.3.2.1.5] 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.

[7.3.2.2.1.2] Regular Expression Flags

JT: I've received a suggestion to just use a `flags` attribute instead.

[7.3.2.2.1.2] Regular Expression Flags

Does it matter that ignore-regex-whitespace pertains to how the regex is interpreted; the other attrbutes to how it is applied? Is there any merit in reflecting this in the markup somehow?

JT: There's no distinction in other programming languages, so I'm not really tempted to make one here.

[7.3.2.3.1] Conditions

When there are multiple parsing methods, some of which have failed, how do conditions apply? Should condition/property be children of <parse>, with the select operation scoped accordingly?

JT: These questions don't apply now that things are structured differently.

JT: No, because it can contain things like `<regex>` and `<valid>`, which can't be negated with `not()`.

[8.1.1] XPath Expressions

AB: The above bears on JT's question: "How should the results of a regex parse be captured (individual variables for each named subexpression, or a node tree containing them)?"

JT also asks "Should the results of a list parse be captured?"

[8.2.1] `version` attribute

JT: I've amended this wording. We need to allow a `version` attribute on `<div>` so that `<include>` works properly.