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 motivation for creating 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 suited for their needs.

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 interface that allows you to plug in datatype modules, written in the programming language of your choice, to define extra datatypes. But the fact that these datatype libraries have to be programmed means that 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 for specifying libraries of datatypes.

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/

3 Terms and definitions

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

3.1 target value

some character data in an XML document that is to have its datatype tested.

3.2 datatype

a *target 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 implementation

a DTLL processor that conforms to this International Standard.

3.5 extended implementation

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

4 Root element

DTLL specifies libraries of datatypes using an XML language. The schema for this language is defined in This International Standard using the compact syntax of the RELAX NG schema language, as defined by Annex C of ISO 19757:2.

Concaternating these schema fragments gives a RELAX NG language that normatively defines the grammatic syntax of the DTLL XML Language.

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

```
datatypes xs = "http://www.w3.org/2001/XMLSchema-datatypes"
default namespace dt = "http://www.jenitennison.com/datatypes"
namespace local = ""

start = \datatypes
```

`<datatypes>` is the document element.

The `version` attribute holds the version of the datatype library language. The current version is 0.4.

If a DTLL version 0.4 processor encounters a datatype library with a version higher than 0.4, it must treat any attributes or elements that it does not understand (that are not part of DTLL 0.4) in the same way as it would treat extension attributes or elements found in the same location.

```
\datatypes = element datatypes {
               attribute version { "0.4" },
               ns?,
               extension-attribute*,
               top-level-element*
             }
```

5 Top-level elements

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

Note maps have been removed as per discussions in Atlanta.

`<include>` elements includes DTLL content from elsewhere. An implementation must replace the content of include elements with the content of the resources referenced by their `href` attributes, so that onward interpretation of the DTLL document is identical to that which would have taken place if the referred-to document's content existed directly in the document containing the `include` element.

Can we leverage XInclude here instead? What happens when included content has different encoding, e.g.? If we use XInclude do we need to allow xml:base liberally?

```
\include = element include {
             attribute href { xs:anyURI },
             extension-attribute*
           }
```

It is an error for a datatype library to contain circular includes. If the datatype library A includes the datatype library B, then B must not include A or include any datatype library that (at any remove) includes A.

`<div>` elements are used to partition a datatype library and to provide a scope for `ns` attributes.

```
\div = element div {
         ns?,
```

```
        extension-attribute*,
        top-level-element*
      }
```

Extension top-level 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
```

6 Datatype definitions

Datatype definitions are named or anonymous.

Named datatypes definitions are specified at the top level of the datatype library using `<datatype>` elements. Each named datatype definition has a name that must uniquely identify it.

The name of the datatype definition is given in the `name` attribute. If this is unprefixed, the nearest ancestor `ns` attribute (including one on the `<datatype>` element itself) is inherited to provide the namespace for the datatype.

```
named-datatype = element datatype {
                   attribute name { xs:QName }, ns?,
                   extension-attribute*,
                   datatype-definition-element*
                 }
```

Anonymous datatypes are used to provide the datatype for a property or variable if that property or variable's type cannot be referred to by name.

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

Datatypes are referenced using qualified names. If the qualified name has no Namespace prefix, the nearest ancestor `ns` attribute (including one on the element that is referring to the datatype) is used to resolve the name.

```
datatype-reference = xs:QName
```

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

```
datatype-definition-element |= property
datatype-definition-element |= parse
datatype-definition-element |= condition
datatype-definition-element |= except
datatype-definition-element |= variable
datatype-definition-element |= local-map
datatype-definition-element |= extension-definition-element
```

6.1 Definition extension elements

Definition extension elements can be used at any point within a datatype definition. If a implementation does not recognise an extension definition element, it must ignore it and behave as if the value passed whatever test the extension definition element represented.

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

6.1.1 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>

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

6.2 Except

Certain aspects of a datatype definition can be negated using the `<except>` element. A target value obeys the constraints of a datatype definition only if none of the expressions of that definition's `<except>` elements evaluate true when the value is tested.

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

```
except = element except {
           extension-attribute*,
           negative-test+
         }

negative-test |= condition
negative-test |= variable
negative-test |= parse
```

7 Parsing

Parsing performs two functions: it tests whether a value adheres to a particular format, and assigns a tree value to a variable to enable pieces of the string value to be extracted, tested, assigned to properties and otherwise processed.

The `<parse>` element holds any number of parsing methods, one or more of which must be satisfied in order for the value to be considered valid. The `name` attribute, if present, specifies the name of the variable to which the tree resulting from the parse is assigned. The first successful parse of those specified within the `<parse>` element is used to give the value of this variable (thus an implementation does not have to attempt to perform further parses once one has been successful).

A datatype can specify as many `<parse>` elements as it wishes. All must be satisfied by a value for that value to be a legal value of the datatype.

```
parse = element parse {
          name?, preprocess*,
          extension-attribute*,
          parsing-method+
        }
```

7.1 Preprocessing

Before a value is parsed by a `<parse>` element, it can be preprocessed. This does not change the string value, but it may simplify the specification of the parsing method that's used.

The only built-in form of preprocessing is whitespace processing. The whitespace can be preserved (`'preserve'`), whitespace characters replaced by space characters (`'replace'`), or leading and trailing whitespace stripped and sequences of whitespace characters replaced by spaces (`'collapse'`, the default).

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

Implementations may specify extension preprocessing methods with additional attributes. These must be ignored by implementations that do not support them.

```
preprocess |= extension-preprocess-attribute
extension-preprocess-attribute = extension-attribute
```

7.2 Parsing Methods

There are two core methods of parsing: via a regular expression, and by specifying a list. This set of methods can be supplemented by extension parsing elements.

```
parsing-method |= regex
parsing-method |= \list
parsing-method |= extension-parsing-element
```

7.2.1 Regex Parsing

The <regex> element specifies parsing via an extended regular expression. To be a legal value, the entire string value must be matched by the regular expression. (Although it's legal to use ^ and $ to mark the beginning and end of the matched string, it's not necessary.)

The tree value generated by parsing consists of a root (document) node with text node and element children. The string value of the root (document) node is the string value itself. There is one element for each named subexpression. The element's name being the name of the subexpression with the namespace indicated by the prefix indicated in the name. If no prefix is used, the element is in no namespace. The string value of each of these elements is the matched part of the string value as a whole.

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

7.2.1.1 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 tree:

```
(root)
           +- year
           |   +- "2003"
           +- "-"
           +- month
           |   +- "12"
           +- "-"
           +- day
               +- "19"
```

We need to explain that the root element name is that of the parse spec. in effect; but when it's not named?

7.2.1.2 Regular Expression Flags

Four attributes modify the way in which regular expressions are applied. These are equivalent to the flags available within XPath 2.0.

By default, the "." meta-character matches all characters except the newline (#xA) character. If dot-all="true" then "." matches all characters, including the newline character.

```
regex-flags |= attribute dot-all { boolean }
```

By default, ^ matches the beginning of the entire string and $ the end of the entire string. If multi-line="true" then ^ matches the beginning of each line as well as the beginning of the string, and $ matches the end of each line as well as the end of the string. Lines are delimited by newline (#xA) characters.

```
regex-flags |= attribute multi-line { boolean }
```

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-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-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?

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

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

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

7.2.1.2.1 Example: Ignoring Whitespace in Regular Expressions

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

7.2.2 Lists

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

The result of parsing the string value based on the <list> element is a node-set of sibling elements. The names of the item elements are implementation-defined.

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*
        }
```

7.2.2.1 Example: Parsing Lists

For example, if you have:

```
<list separator="\s*,\s*" />
```

and the string value:

```
1, 2, 3, 45
```

then the variable is set to the elements in the tree:

```
(root)
           +- item
           |   +- "1"
           +- item
           |   +- "2"
           +- item
           |   +- "3"
           +- item
               +- "45"
```

7.2.3 Extension Parsing Elements

Extension parsing elements can be used to parse elements using methods other than the core methods explained above. Extension parsing elements can be used, for example, to parse a value using EBNF (Extended Backus-Naur Form) or PEGs (Parsing Expression Grammars).

If the extension parsing element is not recognised, the value is considered to fail the parse. If the extension parsing element occurs in a `<parse>` element without any alternative parsing methods, this means no value can match the datatype, and the implementation must issue a warning. Usually, an extension parsing element will be used alongside a built-in parsing method.

```
extension-parsing-element = extension-element
```

7.2.3.1 Example: Using Extension Parsing Elements

```
<parse name="path">

            <ext:ebnf ref="http://www.w3.org/1999/xpath" />
            <regex dot-all="true">.*</regex>
            </parse>
```

8 Testing

Conditions define run-time tests that check values.

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

9 Variable Binding

Properties and variables declare variables for use in binding expressions (i.e. XPath expressions). Property variables are of the form `$this.`*name* where *name* is the name of the property; ordinary variables just use the name of the variable. The variable `$this` refers to the value itself (as does the XPath expression `.`).

Why do we need "$this." ? Seems more natural without it... (What about Namespaces?)
Variable binding is carried out in the order the variables are declared. It is an error if a variable is referenced without being declared. The scope of a variable binding is limited to the following siblings of the variable declaration and their descendants.

9.1 Properties

The `<property>` element specifies a property of the datatype. The values of properties are available via the `dt:property()` extension function within XPath expressions in DTLL (or via other implementation-defined APIs). The value of a property for a value can be referenced using `$this.`*name* where *name* is the value of the `name` attribute on the `<property>` element.

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

9.1.1 Example: Datatype Properties

For example, consider:

```
<datatype name="RRGGBB">
  <parse name="colour">
    <regex ignore-whitespace="true">
     #(?[red][0-9A-F]{2}) (?[green][0-9A-F]{2}) (?[blue][0-9A-F]{2})
    </regex>
  </parse>
  <property name="red" type="hexByte" select="$colour/red"/>
  <property name="green" type="hexByte" select="$colour/green"/>
  <property name="blue" type="hexByte" select="$colour/blue"/>
  <property name="is-greyscale"
            select="$this.red = $this.green and $this.green = $this.blue"/>
</datatype>
```

9.2 Variables

The `<variable>` element binds a value to a variable. Variables are similar to properties except that their values are not accessible via APIs. The value of a variable is accessed through $*name* , where *name* is the name of the variable. It is an error if the name of a variable starts with (or is) `'this'`. For future use, it is also an error if the name of a variable starts with (or is) `'type'`. Variables are used for intermediate calculations.

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

9.3 Type Specifiers

There are two ways to specify a type: via a `type` attribute or via an anonymous `<datatype>` element.

```
type |= attribute type { datatype-reference }
        type |= anonymous-datatype
```

If there is a mapping specified from the type of the provided value to the required type, then that mapping is used to convert the value to the required type. If the value is a standard XPath 1.0 type (string, number,

boolean or node-set), then that value is converted to a string using the `string()` function and interpreted as the string value of the required type. Otherwise (there's no mapping and the value is not a standard XPath type), it's an error.

If no type is specified for a variable or property, then the supplied value is used directly. Note that this value can be a standard XPath type (string, number, boolean or node-set) as well as a value of a datatype defined in the datatype library.

9.4 Value Specifiers

There are two built-in ways to bind a value to a property or variable: 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.

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

If a `value` attribute is specified, its value is the string value of the value of the variable or property; the type of the variable or property is used to interpret that value.

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

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

If a `select` attribute is specified, the XPath expression it contains is evaluated to give the value of the property or variable.

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

Extension binding elements are used where more power is needed to specify the value of a parameter, property or variable. This can be used to provide values using methods such as XSLT or MathML. If an implementation does not support any of the extension binding elements specified, then it must assign to the variable the value specified by the `value` or `select` attribute instead. If an implementation supports one or more of the extension binding elements, then it must use the first extension binding element it understands to calculate the value of the variable.

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

10 Common Constructs

10.1 Common Types

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

```
XPath = text
```

Variable and property values are available within an XPath expression if the variable or property is declared prior to the XPath expression.

Within a datatype library, each datatype has a corresponding extension function named after the name of the datatype. This function takes a single argument, which can be of any type, and returns a typed value of the type specified by the name of the function. The supplied value is converted to the required type using the same rules as for type conversions for variables. Note that this works for all datatypes, including lists.

10.2 Extended Regular Expressions

A regular expression as defined in XPath 2.0

```
regular-expression = text
```

Extended regular expressions 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
```

10.3 DTLL Extension Functions

10.3.1 `dt:item(list-value, number)`

returns the item in the list-value at the index given by the number (counting starts from 1); returns an empty string if the number is greater than the number of items in the list-value. Values that are not of a list type are treated like list-type values with a single item.

10.3.2 `dt:property(value, prop-name)`

returns the value of the named property for the value

10.3.3 `dt:if(test, true, false)`

returns the true value if the test is true and the false value if the test is false. Note that both the true and false arguments are evaluated (unlike the `if` expression in XPath 2.0.

10.3.4 `dt:default(value, default)`

returns the first argument if the effective boolean value of the first argument is true, and the second argument otherwise

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:

[4] Root element

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

[5] Top-level elements

Note maps have been removed as per discussions in Atlanta.

[5] Top-level elements

Can we leverage XInclude here instead? What happens when included content has different encoding, e.g.? If we use XInclude do we need to allow xml:base liberally?

[6.2] Except

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

We need to explain that the root element name is that of the parse spec. in effect; but when it's not named?
[7.2.1.2] Regular Expression Flags

Does it matter that ignore-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?
[8] Testing

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?

[9] Variable Binding

Why do we need "$this." ? Seems more natural without it... (What about Namespaces?)
[9.4] Value Specifiers

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