Codalogic AXE™
'Annotated XML Example' Specification

Version 0.5

Document Revision 4

Table of Contents

Introduction

Annotated XML Example (AXE™) is a simple method for specifying the format of XML data. In its simplest use an example of your XML data can be used to specify the XML format. The XML example can then be annotated with additional characters to more accurately describe your XML data.

AXE™'s main goal is to be easy to use. Therefore it does not support some of the more advanced features supported by other XML data specification languages such as W3C XML Schema.

1 - Quick Overview

This section briefly introduces the AXE™ format. The concepts presented here are discussed further later in the document so it doesn't matter if you don't fully understand the example.

The following is a brief example of an AXE specification:

```
<MyElement a1="12" a2="?int">
    <Element1>This is a string</Element1>
    * <Element2>string</Element2>
    ? <Element3>AComplexType</Element3>
</MyElement>

AComplexType =
    <_ a3="AnInt">MyInt</_>

AnInt = int

MyInt = int( min=0, max=100 )
```

As you can see, the format follows that of the actual XML data that it represents, but has some additional characters added to it to describe the format in more detail.

For example, an AXE parser will look at the value of attribute `a1` ("12") and infer that it is an integer. If you prefer, you can explicitly state that a type is an integer by putting the name of the type where the value normally appears. This is shown for attribute `a2`.

The set of built-in types supported by AXE are described below. They are the set of types specified by W3C XML Schema Part 2.

The example shows a `?` character as the first non-whitespace character of attribute `a2`'s value. This indicates that the attribute is optional. Without the ? the attribute is assumed to be mandatory.

Similar to above, the type of the `Element1` element is inferred to be a string, whereas the type of `Element2` is explicitly stated to be a string.

The `*` before `Element2` indicates that it can occur 0 or more times. A `?` in that location would indicate that the element can appear 0 or 1 times, and a + would indicate 1 or more times.

`Element3`'s type is defined by the user defined `AComplexType` complex type. Effectively the name of the element is substituted for the _ in the complex type definition in the places where the type is used by an element.

The value of the `a3` attribute in the `AComplexType` type uses the user defined type `AnInt`, which is effectively an alias of the `int` built-in type. The body of any element that is associated with the `AComplexType` type uses the user defined `MyInt` type. `MyInt` is defined to be an `int` with a minimum value of 0 and a maximum value of 100.

2 - Identifying AXE Files

The preferred file extension for an AXE file is `.axe`. If you add AXE annotations to an XML file it is recommended that you rename the file to include the `.axe` extension.

3 - Structure of an AXE Specification File

An AXE specification file consists of zero or more example elements followed by zero or more User-Defined Type definitions. The example elements are the set of elements that the document element of an XML instance document is validated against. A simple AXE specification with one example element and no User-Defined Types is:

```
<MyElement a1="12" a2="?int">
    <Element1>This is a string</Element1>
</MyElement>
```

The above example indicates that a valid XML instance document must have a document element called `MyElement` with a mandatory attribute called `a1` of integer type, an optional attribute called `a2` of integer type and a child element called `Element1` of string type.

The following is an example containing two example elements, permitting a valid XML instance to match either the `MyElement` example element or the `YourElement` example element:

```
<MyElement a1="12" a2="?int">
    <Element1>This is a string</Element1>
</MyElement>

<YourElement a="?int">
    <Child1>15.2</Child1>
</YourElement>
```

User-Defined Types are described below (See 9 - Specifying User-Defined Simple Types and 10 - Specifying User-Defined Complex Types).

An AXE specification file may optionally also have an AXE wrapper as described in 12 - Making an AXE Definition with User-Defined Types a Valid XML Document.

4 - Specifying Types

The type associated with an attribute or element can be inferred from an example of the type, or be explicitly stated using a built-in type or a user defined type.

The type of an attribute is specified in the attribute's value field, for example:

```
a1="12"
```

Or:

```
a2="int"
```

The type of an element is specified in the body of the element, for example:

```
<Element1>string</Element1>
```

The built-in types are those of W3C XML Schema Part 2.

The common types include:

`string`, `long`, `unsignedLong`, `int`, `unsignedInt`, `short`, `unsignedShort`, `byte`, `unsignedByte`, `float`, `double`, `boolean`

Date and time types are:

`date`, `time`, `dateTime`, `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `gMonth`, `duration`

Variations on the string type are:

`normalizedString`, `token`, `NMTOKENS`, `Name`, `NCName`, `NMTOKEN`

Arbitrary precision numbers are:

`integer`, `nonPositiveInteger`, `negativeInteger`, `nonNegativeInteger`, `positiveInteger`, `decimal`

Binary data types are:

`hexBinary`, `base64Binary`

Other types are:

`anyURI`, `language`, `QName`, `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES`, `anySimpleType`, `anyAtomicType`, `anyType`, `NOTATION`

To specify a built-in type, put the name of the type in the position described above.

The following types can be inferred by example:

`int`, `long`, `double`, `boolean`, `date`, `time`, `dateTime`, `gYearMonth`, `gMonthDay`, `gDay`, `gMonth`, `duration`

Any unrecognised value is assumed to be a `string` type.

Note that an AXE parser may infer an incorrect type when the example of the type is ambiguous. In this case you should explicitly specify the type using a built-in type or a user-defined type.

The above built-in types are all simple types. Simple types may have parameters associated with them to further specify the type. The parameters are placed in brackets after the type name. The parameters are specified using a comma separated list of either the parameter name on its own, or `x=y` pairs in which the `x` value of the pair is the name of the parameter, and the `y` value of the pair is the parameter's value. For example:

int(min=0, max=100)

This specifies an integer type with a minimum value of 0 and a maximum value of 100.

When a parameter value contains spaces it should be placed in quotes, as in:

string(pattern = "\w{3} \d{3}")

The allowed type parameters mirror the W3C XML Schema Part 2 facets. The following parameters are allowed:

min	An AXE alias of minInclusive
minInclusive	The minimum inclusive value
minExclusive	The minimum exclusive value
max	An AXE alias of maxInclusive
maxInclusive	The maximum inclusive value
maxExclusive	The maximum inclusive value
minLength	The minimum length of a string or binary type
maxLength	The maximum length of a string or binary type
length	The fixed length of a string or binary type
enumeration	An enumeration. Multiple occurences of this parameter are allowed
enum	An AXE alias of enumeration. Multiple occurences of this parameter are allowed
pattern	A pattern. Multiple occurences of this parameter are allowed
fractionDigits	The number of digits to the right of a decimal point in decimal types
totalDigits	The maximum number of digits an integer or decimal type can have
whiteSpace	Indicates whitespace handling. Can be preserve ,replace or collapse

In addition to the above W3C XML Schema Part 2 facets, AXE also supports the following parameters:

anyEnumeration	Specifies that the set of specified enumerations is open and extensible. While parsing an XML instance file values that do not correspond to the specified enumerations are not treated as validation errors.
anyEnum	An alias of anyEnumeration

AlternativeEnumeration	See description below
AltEnum	An alias of AlternativeEnumeration
id	Indicates that the type is an identifier. See description below
idRef	Indicates that the type is a reference to an identifier. See description below

The `AlternativeEnumeration` parameter (and its alias `AltEnum`) allows additional enumerated values to be associated with a type. The set of valid values of the type becomes the union of the specified base type (as modified by the other specified parameters), plus the enumerated values specified by any `AlternateEnumeration` parameters. For example, to specify that a `range` type must have an unsigned integer value or the enumerated value `unbounded`, you could specify:

```
range = unsignedInt( altEnum=unbounded )
```

The `id` and `idRef` parameters are used to allow parts of an XML instance document to reference other parts of an XML instance document. The value part of the `id` and `idRef` parameters specify the name of an id set. For example to specify that the `authorId` and `authorRef` types are associated with the `author` id set you could do:

```
authorId = unsignedInt( id=author )

authorRef = unsignedInt( idRef=author )
```

Multiple id sets may be specified, with each id set being given a unique name, for example:

```
authorId = unsignedInt( id=author )

authorRef = unsignedInt( idRef=author )

bookId = unsignedInt( id=book )

bookRef = unsignedInt( idRef=book )
```

During parsing of an XML instance document a parser records an implementation specific reference to the parent of an attribute or element whose type has an `id` parameter and ensures that the value of the attribute or element is unique within the applicable id set.

An attribute value or element body may contain multiple instances of a simple type, for example:

```
<MyElement>10 15 82</MyElement>
```

This is called a simple type array. The number of instances of the simple type in such an array is specified in square brackets after the type name, for example:

int[1..10]

Or:

int[1..*](min=0, max=100)

The first number in the array specification is the minimum number of times the type should appear and the second number is the maximum number of times the type should appear. A `*` in the place of the second number indicates that the upper limit is unbounded. The array specification min and max values are separated by two dots (`..`).

Simple type parameters and array specifications can also be applied to user-defined simple types.

The contents of an element's body can be specified to be empty using an empty element, for example:

```
<MyElement></MyElement>
```

Or:

```
<MyElement/>
```

4.1 - Specifying Attributes from the XML Namespace

An exception to the above is specifying the use of attributes from the XML namespace, such as `xml:lang` and `xml:id`. Such definitions use the name of the attribute to indicate the type, and ignore the attribute value. Therefore, to indicate the use of the `xml:lang` attribute, do:

```
<MyElement xml:lang="en">
    ...
</MyElement>
```

5 - Specifying How Many Times an Attribute Can Occur

By default an attribute is considered to be mandatory. If the attribute is optional place a `?` character as the first non-whitespace character of the attribute's value, for example:

```
<MyElement attr="? int"></MyElement>
```

6 - Specifying How Many Times an Element Can Occur

By default an element is considered to be required once and only once. If the element is optional (0 or 1 times), place a `?` character in front of the element specification, for example:

```
?<MyElement attr="int">string</MyElement>
```

If an element can appear 0 or more times, place a `*` character in front of the element specification, for example:

```
*<MyElement attr="int">string</MyElement>
```

If an element can appear 1 or more times, place a + character in front of the element specification, for example:

```
+<MyElement attr="int">string</MyElement>
```

White space may appear between the annotation character and the start of the element specification, for example:

```
+ <MyElement attr="int">string</MyElement>
```

A specific range of occurrences can be specified within a pair of braces, for example:

```
{5,10}<MyElement attr="int">string</MyElement>
```

The minimum number of times the element can occur is specified by the first number within the braces. If the maximum number of times the element can appear is the same as the minimum number, then the braces contain no further content. For example, if the element must appear exactly 5 times, the following can be used:

```
{5}<MyElement attr="int">string</MyElement>
```

If the maximum number of times the element can appear is a finite number, then a comma is placed after the first number, and then the maximum number of times the element can appear is specified, for example:

```
{5,10} <MyElement attr="int">string</MyElement>
```

If the maximum number of times the element can appear is unbounded then a `*` instead of a number for the maximum number of times the element can appear, for example:

```
{5,*} <MyElement attr="int">string</MyElement>
```

7 - Specifying Complex Element Bodies

If the body of an element contains multiple child elements, then by default it is assumed that the elements occur in the sequence they are specified in in the specification. For example:

```
<MyElement a1="12" ?a2="int">
    <Element1>This is a string</Element1>
    * <Element2>string</Element2>
    ? <Element3 a1="MyInt">MyInt</Element3>
</MyElement>
```

would mean that `MyElement` contains 1 instance of `Element1`, followed by 0 or more instances of `Element2`, optionally followed by an instance of `Element3`. (This mirrors W3C XML Schema `xs:sequence`.)

If only one of the child elements should appear, then include the `|` character between the element specifications. For example:

```
<MyElement>
    <Element1>This is a string</Element1>
    | {1,6} <Element2>date</Element2>
    | + <Element3>int</Element3>
</MyElement>
```

means that the body of `MyElement` can contain either a single occurrence of `Element1`, or between 1 and 6 occurrences of `Element2`, or 1 or more occurrences of `Element3`. (This mirrors W3C XML Schema `xs:choice`.)

If multiple child elements can appear, but in any order, then include the `^` between the child element specifications. For example:

```
<MyElement>
    <Element1>This is a string</Element1>
    ^ {1,6} <Element2>date</Element2>
    ^ + <Element3>int</Element3>
</MyElement>
```

(This mirrors W3C XML Schema `xs:all` or Relax-NG's interleave.)

8 - Specifying Groups of Elements

Child elements can be specified to appear in groups. This is indicated by placing (round) brackets around the elements forming the group. For example:

```
<MyElement a1="12" ?a2="int">
    <Element1>This is a string</Element1>
    ?(
        + <Element2>string</Element2>
        + <Element3 a1="MyInt">MyInt</Element3>
     )
</MyElement>
```

A group may contain body structure characters (i.e. `|` characters), for example:

```
<MyElement a1="12" ?a2="int">
    <Element1>This is a string</Element1>
    ?(
        + <Element2>string</Element2>
        | + <Element3 a1="MyInt">MyInt</Element3>
     )
</MyElement>
```

The number of times the group is allowed to appear is indicated using the same method to specify the number of times an element can appear. For example:

```
<MyElement a1="12" ?a2="int">
    <Element1>This is a string</Element1>
    +(
        + <Element2>string</Element2>
        | + <Element3 a1="MyInt">MyInt</Element3>
     )
</MyElement>
```

indicates that the group can appear 1 or more times, and:

```
<MyElement a1="12" ?a2="int">
    <Element1>This is a string</Element1>
    (
        + <Element2>string</Element2>
        | + <Element3 a1="MyInt">MyInt</Element3>
     )
</MyElement>
```

indicates that the group can appear once and only once.

9 - Specifying User-Defined Simple Types

User-defined types are defined after the example elements, if present. They follow a `Name = Type` format. The `Name` must be an XML Name without any colons. The `Type` of the user-defined simple type follows the format as described in 4 - Specifying Types.

For example, given:

```
<MyElement>
    <Element1 a2="AnInt" a3="MyInt">MyOtherInt</Element1>
</MyElement>

AnInt = int

MyInt = int( min=0, max=100 )

MyOtherInt = MyInt( min=0, max=50 )
```

`AnInt` becomes an alternative name for the built-in `int`, `MyInt` defines an integer limited to the range 0 to 100, and `MyOtherInt` further restricts `MyInt` to the number range 0 to 50.

To use a user-defined type in an XML document, place the `Name` of the type where a built-in type name would appear (see example above). See also 17 - Modularity of User-Defined Types for how User-Defined Types in other AXE modules can be referenced.

10 - Specifying User-Defined Complex Types

User-defined complex types provide a useful way to decompose your XML document description into more manageable chunks in much the same way as you would use methods and functions to decompose a program's structure. Their use is highly recommended.

Specifying a User-Defined Complex Type is similar to defining a User-Defined Simple Type in that it has the form `Name = Type`, and are defined after the example elements, if present.

In this case the `Type` looks like an element declaration, except that the name of the element is replaced with a _ character; for example:

```
MyComplex =
        <_ attr="?int">
            * <Child1 a3="string">time</Child1>
            | * <Child2>date</Child2>
        </_>
```

To use the type, place the name of the type in the body of the element to which the type is being assigned in the same way you would for a user-defined simple type, for example:

```
<MyElement>MyComplex</MyElement>
```

This effectively gives a definition for `MyElement` of:

```
<MyElement attr="?int">
    * <Child1 a3="string">time</Child1>
    | * <Child2>date</Child2>
</MyElement>
```

11 - How a User-Defined Complex Type Affects a Referencing Element

A User-Defined Complex Type can contribute attributes and/or element body content to an element that references the User-Defined Complex Type.

If a User-Defined Complex Type includes attribute definitions then these attributes become part of the referencing element's definition.

If the content of a User-Defined Complex Type is empty then the User-Defined Complex Type does not affect the content of the referencing element.

If the content of a User-Defined Complex Type is a simple type, then this becomes the simple type content of the referencing element. An element that references a User-Defined Complex Type whose content is a simple type must not define content locally, and any other User-Defined Complex Types referenced by the referencing element must have empty content.

If the content of a User-Defined Complex Type is one or more child elements then these child elements are conceptually pasted into the referencing elements definition as a group (See 8 - Specifying Groups of Elements) in place of the reference to the User-Defined Complex Type.

For example, if the following AXE definition occurs:

```
<MyElement a1="12">
    <Element1>This is a string</Element1>
    * MyType1
    ? MyType2
</MyElement>

MyType1 =
    <_ a3="AnInt">
        <T11>int</T11>
        <T12>string</T12>
    </_>

MyType2 =
    <_ a4="string">
        <T21>int</T21>
        <T22>string</T22>
    </_>
```

the effective definition of `MyElement` is:

```
<MyElement a1="12" a3="AnInt" a4="string">
    <Element1>This is a string</Element1>
    * (
        <T11>int</T11>
        <T12>string</T12> )
    ? (
        <T21>int</T21>
        <T22>string</T22> )
</MyElement>
```

The process by which this effective definition is realised is implementation dependent. For example, in an AXE to W3C XML XSD Schema converter it is recommended that if a User-Defined Complex Type in an element definition is the first specified item of an element's content and it may occur once and only once, then the element is modelled as an XML Schema `xs:extension` of the User-Defined Complex Type. If the referenced User-Defined Complex Type is not the first specified item of content, or is not specified to occur only once, then the User-Defined Complex Type definition should be treated as if it attributes defined an XML Schema attribute group, and it's content defined an XML Schema model group.

12 - Making an AXE Definition with User-Defined Types a Valid XML Document

Because the user defined types appear after the main example XML, without special consideration an AXE document may not be a valid XML document. To address this, the AXE specification can be optionally wrapped in an `axe` element, which has a namespace prefix associated with it that is associated with the `http://codalogic.com/axe` namespace. For example:

```
<axe:axe xmlns:axe="http://codalogic.com/axe">

    <MyElement a1="12" a2="?int">
        <Element1>This is a string</Element1>
        * <Element2>string</Element2>
        ? <Element3>AComplexType</Element3>
    </MyElement>

    AComplexType =
        <_ a3="AnInt">MyInt</_>

    AnInt = int

    MyInt = int( min=0, max=100 )

</axe:axe>
```

This more readily allows the AXE specification to be edited in an XML editor.

13 - Specifying an XML Namespace

An XML document can be associated with an XML namespace by including an XML namespace declaration in the first element. For example, to associate the default namespace with your document, do:

```
<MyElement xmlns="http://mynamespace.com">
    ...
</MyElement>
```

To associate a namespace prefix with your document, do:

```
<myns:MyElement xmlns:myns="http://mynamespace.com">
    ...
</myns:MyElement>
```

In the latter case, elements with names including the namespace prefix will be associated with the specified namespace, and elements with names without a prefix will not be associated with a namespace.

14 - Marking the Content of an Element as 'Mixed'

There are two ways to mark the content of an element as 'Mixed'. Firstly you can add the `mixed` attribute from the AXE namespace to the element's definition and set its value to `true`. For example, assuming the namespace prefix 'axe' is mapped to the AXE namespace then you can do:

```
<MyElement axe:mixed="true" ...>
    ...
</MyElement>
```

Alternatively you can include the string `##mixed` in the body of the element definition, for example:

```
<MyElement>
    ##mixed
    ...
</MyElement>
```

Any user-defined complex type that has child elements that is referenced as an immediate child in an element marked as 'mixed' must also be marked as 'mixed'. Similarly, if a user-defined complex type is marked as 'mixed' then any element that references it as an immediate child must also be marked as 'mixed'.

15 - Extensibility and Versioning

An AXE specification has two mechanisms for specifying extensibility and versioning.

You can mark the whole AXE definition as 'open' by including the `open` attribute from the AXE namespace in the definition's outer most element and setting its value to `true`. For example, assuming the namespace prefix 'axe' is mapped to the AXE namespace then you can do:

```
<axe:axe xmlns:axe="http://codalogic.com/axe" axe:open="true">
    ...
</axe>
```

If an AXE specification is marked as 'open' then all unknown attributes are ignored, and unknown child elements that appear before any element's end tag are ignored.

Alternatively you can specify specific places where unknown elements and attributes are permitted using the `any` attribute and element in the AXE namespace. If an elements's definition includes AXE's `any` attribute then any unknown attribute is ignored. If an element's content definition includes one or more `any` elements, then in an XML instance document any unknown element appearing at that position within the element's content is ignored.

The normal rules for specifying the cardinality of elements and attributes apply to these constructs. Therefore, if the namespace prefix 'axe' is mapped to the AXE namespace, a typical example of using the `any` attribute and element is:

```
<MyElement a1="12" axe:any="?">
    <Element1>This is a string</Element1>
    * <axe:any/>
</MyElement>
```

16 - Modelling Polymorphism

Some XML documents model the type of polymorphism found in programming languages such as Java. To model this behaviour the `polymorphic`, `abstract`, `base`, and `selector` attributes are available in the AXE namespace.

To indicate that a User-Defined Complex Type is polymorphic, set AXE's `polymorphic` attribute to `true`, for example (if the namespace prefix 'axe' is mapped to the AXE namespace):

```
MyBase =
    <_ a4="string" axe:polymorphic="true">
        <E1>int</E1>
    </_>
```

If the polymorphic base type can not be used without being extended, then it can be marked as abstract by setting the AXE `abstract` attribute to `true`, for example:

```
MyBase =
    <_ a4="string" axe:polymorphic="true" axe:abstract="true">
        <E1>int</E1>
    </_>
```

Types that extend a polymorphic base use AXE's `base`, and `selector` attributes. The `base` attribute specifies the base that the type extends. The `selector` specifies an `x=y` pair where the `x` value is the name of an attribute, and the `y` value specifies the value that the attribute must be set to in order for the type to be selected, for example:

```
MyExtension1 =
    <_ aExtra="string" axe:base="MyBase" axe:selector="aExtra=Ext1">
        <E2>string</E2>
    </_>
```

This indicates that the `aExtra` attribute must be set to the value `Ext1` in order for the parsed type to be treated as `MyExtension1`. Thus an example of an element that should be validated against the `MyExtension1` code type might be:

```
<MyElement aExtra="Ext1" a4="This is the start">
    <E1>18</E1>
    <E2>Housing plan</E2>
</MyElement>
```

Note that the attribute specified in the `selector` attribute may be defined in either the base type or the extended type. The order in which possible candidate selectors are evaluated is undefined and it is the responsibility of the schema designer to ensure that determination of the applicable extension is deterministic independent of the order in which the selectors are evaluated.

For compatibility with W3C XML Schema, if the attribute name portion of the `selector` attribute value can be interpreted as a QName that corresponds to the `type` attribute in the `http://www.w3.org/2001/XMLSchema-instance` namespace, then the value portion of the `selector` attribute value should be treated as a QName; otherwise it should be treated as a token and any comparisons should be made on a character-by-character basis irrespective of the type of the attribute referenced in the `selector` attribute.

If during parsing an XML instance document none of the selectors found in the extensions of the base type are matched then an element is parsed assuming its type is that of the base type.

17 - Modularity of User-Defined Types

To aid modularity and reuse AXE specifications can be split into multiple modules. A module may be described using multiple files. A module is identified using the `module` attribute from the AXE namespace. Typically the value of the `module` attribute is a URI or reverse domain name notation. For example, if the namespace prefix 'axe' is mapped to the AXE namespace, the following specifies that the file is part of the `com.codalogic.schemas.libraryTypes` module:

```
<axe:axe xmlns:axe="http://codalogic.com/axe"
         axe:module="com.codalogic.schemas.libraryTypes">
    authorId = unsignedInt( id=author )
</axe:axe>
```

When a reference to a User-Defined Type is made, if it has no prefix then the type is searched for in the current file. If the reference has a prefix then a search is made amongst the known files that are specified to be part of the module that the namespace prefix is associated with. For example, to reference the above `authorId` from a separate file you can do:

```
<axe:axe xmlns:axe="http://codalogic.com/axe"
         xmlns:ltypes="com.codalogic.schemas.libraryTypes">

    <MyElement>ltypes:authorId</MyElement>
</axe:axe>
```

Note that specifying that a file is in a particular module does not indicate that all the elements and attributes that it defines are in the namespace of the module. Specifying the namespace of a module and the namespace that elements and attributes are associated with is orthogonal in AXE.

END