XML Schema to Ecore Mapping

June 28, 2004 (draft)

EMF support for XML Schema has been significantly enhanced in version 2.0. EMF 2.0 users should therefore refer to this material in place of Chapter 7 of Eclipse Modeling Framework: A Developer’s Guide (Frank Budinsky et al., Addison-Wesley, August 2003).

This document describes the mapping from XML Schema to Ecore. This mapping is used by the EMF importer when creating an EMF model from XML Schema. For each schema component, the corresponding Ecore representation is described along with any attributes and nested content which affect the resulting model.

Annotations on schema components can be used to customize the mapping for a particular schema. The following set of attributes from the Ecore namespace (http://www.eclipse.org/emf/2002/Ecore), are used for this purpose:

- **ecore:instanceClass** on a simple type definition is used to specify the Ecore instanceClassName (that is Java class) of the corresponding EDataType. For example, specifying ecore:instanceClass="byte[]" produces a data type, features of which will return byte[].
- **ecore:name** on any named component or on a wildcard can be used to override the name of the corresponding ENamedElement.
- **ecore:documentRoot** on a schema component is used to change the name of the document root EClass from the default ("DocumentRoot").
- **ecore:package** on a schema component is used to specify the fully qualified Java package name.
- **ecore:nsPrefix** on a schema component specifies the nsPrefix attribute of the corresponding EPackage.
- **ecore:reference** on an attribute or element declaration of type IDREF, IDREFS, or anyURI can be used to specify the target of the corresponding EReference. The value of "ecore:reference" must be a QName that resolves to a complex type within the schema.
- **ecore:opposite** on an element or attribute declaration, that maps to an EReference, can be used to specify the element or attribute, in the target complex type, corresponding to the reference’s eOpposite.
- **ecore:mixed** on a complex type definition will make it behave as if it had the mixed="true" attribute declared.
- **ecore:featureMap** on a model group or reference, an element declaration or reference, or a complex type, can be used to produce or block the use of a feature map in the corresponding Ecore representation. By default, a feature map is used to implement mixed complex types (section 3.4), substitution groups (section 5.9), abstract elements (section 5.10), repeating model groups (section 6.2), and wildcards (section 7).
- **ecore:ignore** on facets, annotations, documentation or appinfo components, instructs the importer to ignore them during XML Schema to Ecore conversion.

These attributes are described in more detail in the sections, below, which correspond to the components to which they apply. An overview of feature maps can be found in the paper titled: EMF FeatureMaps.

Many of the Ecore elements created from XML Schema components require extended meta data (that is, data above and beyond what’s representable in Ecore itself) to retain all of the information provided by the schema. An EAnnotation with a source attribute set to “http://org/eclipse/emf/ecore/util/ExtendedMetaData” is used for this purpose. In the following sections, the term “extended meta data EAnnotation” will be used to refer to this type of EAnnotation.

One important use of such extended meta data EAnnotations is to record the original name of an XML Schema component corresponding to an Ecore element whose name is adjusted while mapping to Ecore. Because XML Schema naming conventions are less restrictive than Java’s (and consequently Ecore’s), names sometimes need to be converted to conform to the naming conventions outlined in the Java Language Specification (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307).
In some situations, the mapping rules, described below, might result in Ecore elements with conflicting names (for example, two EAttributes that are in the same EClass and have the same name). In such situations, the second and subsequent elements will be made unique by appending a number to the end of their names (for example, “foo1”).

1 Schema

A schema element maps to an EPackage. The name, nsURI, and nsPrefix of the EPackage depend on whether or not the schema has a targetNamespace.

1.1 Schema without targetNamespace

A schema with no targetNamespace maps to an EPackage initialized as follows:

\[
\begin{align*}
\text{nsURI} &= \text{the URI of the schema document} \\
\text{nsPrefix} &= \text{last segment of the URI (short file name), excluding the file extension} \\
\text{name} &= \text{same as nsPrefix} \\
\text{eAnnotations} &= \text{an extended meta data EAnnotation}
\end{align*}
\]

The details map of the extended meta data EAnnotation contains the following entry:

\[
\text{key} = "\text{qualified}'', \quad \text{value} = "\text{false}"
\]

1.2 Schema with targetNamespace

If a schema has a targetNamespace attribute, then it is used to both initialize the corresponding EPackage as well as to specify the fully qualified Java package name, via the basePackage property of class GenPackage in the generator model.

In this case, the EPackage attributes are set as follows:

\[
\begin{align*}
\text{nsURI} &= \text{the targetNamespace value} \\
\text{nsPrefix} &= \text{a last segment of the Java package name (derived from the targetNamespace)} \\
\text{name} &= \text{same as nsPrefix}
\end{align*}
\]

There is no extended meta data EAnnotation in this case.

The Java package name, and consequently the nsPrefix, is derived from the targetNamespace using the following algorithm:

1. Strip the URI protocol and leading slash (“/”) characters (for example, http://www.example.com/library -> www.example.com/library)
2. Remove “www” and then reverse the components of the URI authority, if present (for example, www.example.com/library -> com.example/library)
3. Replace slash (“/”) characters with dot (“.”) characters
4. Split mixed-case names into dot-separated lower case names

The nsPrefix is then set to the last component of the Java package name while the basePackage property in the GenPackage is set to the rest of the name:

```xml
<xsd:schema targetNamespace="http://www.example.com/library">
  ...
</xsd:schema>
```

EPackage

- name="library"
- nsPrefix="library"
- nsURI="http://www.example.com/library"

GenPackage

- basePackage="com.example"

---

**1.3 Schema with ecore:nsPrefix annotation**

The ecore:nsPrefix attribute can be used to explicitly set the nsPrefix attribute of the corresponding EPackage:

```xml
<xsd:schema ecore:nsPrefix="myprefix" … >
  ...
</xsd:schema>
```

EPackage

- nsPrefix="myprefix"

---

**1.4 Schema with ecore:package annotation**

The ecore:package attribute can be used to specify the fully qualified Java package name corresponding to the schema. It sets both the name of the corresponding EPackage as well as the basePackage of the GenPackage (in the generator model) based on the Java package name, as described in section 1.2.

```xml
<xsd:schema ecore:package="org.basepackage.mypackage" … >
  ...
</xsd:schema>
```

EPackage

- name="mypackage"

GenPackage

- basePackage="org.basepackage"

---

**1.5 Schema with global element or attribute declarations**

If there is one or more global element or attribute declaration in the schema, then an EClass, representing the document root, is created in the schema’s EPackage. The name of the document root class is “DocumentRoot” by default.

```xml
<xsd:schema … >
  <xsd:element … >
    ...
  </xsd:schema>
```

EPackage

- EClass
  - name="DocumentRoot"

A document root class will contain one feature for every global attribute or element declaration in the schema (see sections 4.7 and 5.8, below). A single instance of this class is used as the root object of an XML resource (that is, a conforming XML document). This instance will have exactly one of its element features set; the one corresponding to the global element at the root of the XML document. The features corresponding to global attribute declarations will never be set, but are used for setting values in attribute wildcard feature maps.
The document root EClass looks like one corresponding to a mixed complex type (see section 3.4) including a “mixed” feature, and derived implementations for the other features in the class. This allows it to maintain comments and white space that appears in the document, before the root element. A document root class contains two more EMap features, both String to String, to record the namespace to prefix mappings (xMLNSPrefixMap) and xsi:schemaLocation mappings (xSISchemaLocation) of an XML instance document.

1.6 Schema with.ecore:documentRoot annotation

The name of a document root class, if there is one, can be changed from the default (“DocumentRoot”) by including an Ecore:documentRoot attribute on the schema:

```
<xsd:schema ecore:documentRoot="LibraryRoot" … >
  <xsd:element … >
  …
</xsd:schema>
```

1.7 Schema with elementFormDefault or attributeFormDefault

Qualification of local elements and attributes can be globally specified by a pair of attributes, elementFormDefault and attributeFormDefault, on the schema element, or can be specified separately for each local declaration using the form attribute. Any of these attributes may be set to “qualified” or “unqualified”, to indicate whether or not locally declared elements and attributes must be qualified or not.

Both elementFormDefault and attributeFormDefault have no effect on the corresponding EPackage or DocumentRoot EClass (if it exists), however the Ecore model for any corresponding local declarations may include additional information. For details see sections 4.5 (attributes) and 5.7 (elements).

```
<xsd:schema elementFormDefault="qualified" … >
  …
</xsd:schema>
```

2 Simple Type Definition

Each simple type definition of a schema maps to an EDataType in the eClassifiers list of the EPackage corresponding to its target namespace. The name, instanceClass, and eAnnotations of the EDataType depend on the contents of the type.

Some simple types map to Java types that cannot support nillable elements. In these cases, a second (wrapper) EDataType will be created for the type as described in section 5.4 (Nillable element). Here we just describe the primary EDataType that corresponds to the simple type.

2.1 Simple type with restriction

The attributes of an EDataType corresponding to a restricted simple type are set as follows:

- **name** = the name of the simple type converted, if necessary, to a proper Java class name (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)
- **instanceClass** = the instanceClass of the EDataType corresponding to the base type
- **eAnnotations** = an extended meta data EAnnotation
The **details** map of the extended meta data **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered name of the simple type
- **key** = "baseType", **value** = the **EDataType** corresponding to the restriction’s base type

Each simple type constraint in the restriction will produce an additional **details** entry as follows:

- **key** = the name of constraint, **value** = the constraint’s value

<table>
<thead>
<tr>
<th>XSD</th>
<th>ECore</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:simpleType name=&quot;zipCodes&quot;&gt;</code></td>
<td><strong>EDataType</strong></td>
</tr>
<tr>
<td><code>&lt;xsd:restriction base=&quot;xsd:integer&quot;&gt;</code></td>
<td><code>name=&quot;ZipCodes&quot;</code></td>
</tr>
<tr>
<td><code>&lt;xsd:minInclusive value=&quot;10000&quot;/&gt;</code></td>
<td><code>instanceClass=&quot;int&quot;</code></td>
</tr>
<tr>
<td><code>&lt;xsd:maxInclusive value=&quot;99999&quot;/&gt;</code></td>
<td><strong>EAnnotation</strong></td>
</tr>
<tr>
<td><code>&lt;/xsd:restriction&gt;</code></td>
<td><code>source=&quot;.../ExtendedMetaData&quot;</code></td>
</tr>
<tr>
<td><code>&lt;/xsd:simpleType&gt;</code></td>
<td><code>details=&quot;name-&gt;zipCodes, baseType-&gt;.../XMLType#integer, minInclusive-&gt;10000, maxInclusive-&gt;99999&quot;</code></td>
</tr>
</tbody>
</table>

The **ecore:ignore** attribute can be specified on a constraint to suppress it in the corresponding **EDataType**:

<table>
<thead>
<tr>
<th>XSD</th>
<th>ECore</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:minInclusive value=&quot;10000&quot; ecore:ignore=&quot;true&quot;/&gt;</code></td>
<td><strong>No minInclusive entry in details map</strong></td>
</tr>
</tbody>
</table>

### 2.2 Simple type with enumeration facets

An enumeration restriction of a base type whose corresponding **EDataType**’s **instanceClass** is **java.lang.String** (for example, xsd:string, xsd:NCName, etc.), will map to an **EEnum**, but only if all of the enumeration values are valid Java identifiers. The **EEnum** is initialized as follows:

- **name** = the name of the simple type converted, if necessary, to a proper Java class name (see [http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307](http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307))
- **eLiterals** = one **EEnumLiteral** for each enumeration in the restriction
- **eAnnotations** = an extended meta data **EAnnotation**

Each **EEnumLiteral** has the following attributes:

- **name** = the value of the schema enumeration
- **value** = an integer value sequentially assigned, starting at 0

The **details** map of the extended meta data **EAnnotation** contains the following entry:

- **key** = "name", **value** = the unaltered name of the simple type

<table>
<thead>
<tr>
<th>XSD</th>
<th>ECore</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:simpleType name=&quot;USState&quot;&gt;</code></td>
<td><strong>EEnum</strong></td>
</tr>
<tr>
<td><code>&lt;xsd:restriction base=&quot;xsd:string&quot;&gt;</code></td>
<td><code>name=&quot;USState&quot;</code></td>
</tr>
<tr>
<td><code>&lt;xsd:enumeration value=&quot;AK&quot;/&gt;</code></td>
<td><strong>EEnumLiteral name=&quot;AK&quot; value=0</strong></td>
</tr>
<tr>
<td><code>&lt;xsd:enumeration value=&quot;AL&quot;/&gt;</code></td>
<td><strong>EEnumLiteral name=&quot;AL&quot; value=1</strong></td>
</tr>
<tr>
<td><code>&lt;/xsd:restriction&gt;</code></td>
<td><strong>EAnnotation</strong></td>
</tr>
<tr>
<td><code>&lt;/xsd:simpleType&gt;</code></td>
<td><code>source=&quot;.../ExtendedMetaData&quot;</code></td>
</tr>
<tr>
<td><code>&lt;/xsd:simpleType&gt;</code></td>
<td><code>details=&quot;name-&gt; USState&quot;</code></td>
</tr>
</tbody>
</table>
If the base type maps to something other than `java.lang.String` (e.g., `xsd:int`), or any of the enumeration values are invalid Java identifiers (e.g., `value="a:b:c"`), then the type will instead map to an ordinary `EDataType` as described in the previous section (2.1). If the Java instance class of such an `EDataType` is primitive (for example, `int`), `EAttributes` of the type, will have a default value set (see section 4.4).

### 2.3 List simple type

The attributes of an `EDataType` corresponding to a list simple type are set as follows:

- **name** = the name of the simple type converted, if necessary, to a proper Java class name (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)
- **instanceClass** = "java.util.List"
- **eAnnotations** = an extended meta data `EAnnotation`

The `details` map of the extended meta data `EAnnotation` contains the following entries:

- **key** = "name", **value** = the unaltered name of the simple type
- **key** = "itemType", **value** = the type of the list

```xml
<xsd:simpleType name="actorsList">
    <xsd:list itemTypetx="xsd:NCName"/>
</xsd:simpleType>
```

```java
EDataType
    name="ActorsList"
    instanceClass="java.util.List"
    EAnnotation
        source="…/ExtendedMetaData"
        details="name->actorsList, itemType->…/XMLType#NCName"
```

### 2.4 Union simple type

The attributes of an `EDataType` corresponding to a union simple type are set as follows:

- **name** = the name of the simple type converted, if necessary, to a proper Java class name (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)
- **instanceClass** = a common instance class of the members (if there is one) or "java.lang.Object"
- **eAnnotations** = an extended meta data `EAnnotation`

If the `EDataType`s corresponding to the union members share a common `instanceClass`, then the `instanceClass` of the union’s `EDataType` is set to this common value. If they are not all the same, then "java.lang.Object" is used instead.

The `details` map of the extended meta data `EAnnotation` contains the following entries:

- **key** = "name", **value** = the unaltered name of the simple type
- **key** = "memberTypes", **value** = the space-separated list of memberTypes in the union

```xml
<xsd:simpleType name="zipUnion">
    <xsd:union memberTypes="zipCodes USState"/>
</xsd:simpleType>
```

```java
EDataType
    name="ZipUnion"
    instanceClass="java.lang.Object"
    EAnnotation
        source="…/ExtendedMetaData"
        details="name->zipUnion,
        memberTypes->zipCodes, USState"
```
2.5 Anonymous simple type

If an anonymous simple type is used for an element (or attribute) declaration, then the corresponding \textbf{EDataType name} will be set to the converted name of the enclosing element (or attribute) with the suffix "Type" appended. The "name" entry in the details map of the extended meta data \textbf{EAnnotation} will have the following value in this case:

\textbf{EDataType name} = \textbf{MyElementType}
\textbf{EAnnotation details} = "name->myElement_.type, …"

If an anonymous simple type is used as the base type of a restriction, then the corresponding \textbf{EDataType name} will be set to the enclosing type’s converted name with the suffix "Base", instead of "Type". The "name" entry in the details map of the extended meta data \textbf{EAnnotation} will have the suffix "_.base" in this case:

\textbf{EDataType name} = \textbf{MyTypeBase}
\textbf{EAnnotation details} = "name->myType_.base, …"

Similarly, if an anonymous simple type is used as the item type of a list, then the corresponding \textbf{EDataType name} will be set to the enclosing type’s converted name with the suffix "Item" and the "name" entry in the details map of the extended meta data \textbf{EAnnotation} will have the suffix "_.item":

\textbf{EDataType name} = \textbf{MyTypeItem}
\textbf{EAnnotation details} = "name->myType_.item, …"

Finally, if an anonymous simple type is used as a member type of a union, then the corresponding \textbf{EDataType name} will be set to the enclosing type’s converted name with the suffix "Member", but in this case it will be followed by a number representing the position (starting from 0) of the member in the union. The "name" entry in the details map of the extended meta data \textbf{EAnnotation} will have the suffix "_.member" also qualified with the position number.

\textbf{EDataType name} = \textbf{MyTypeMember0}
\textbf{EAnnotation details} = "name->myType_.member0, …"
2.6 Simple type with ecore:name

The ecore:name attribute can be used to set the name of the EDataType, for example, if the corresponding simple type is anonymous or if the default name conversion is unacceptable.

```xml
<xsd:simpleType name="stName"
    ecore:name="MyName">
    ...
</xsd:simpleType>
```

```
EDataType
    name="MyName"
    ...
```

2.7 Simple type with ecore:instanceClass

The ecore:instanceClass attribute can be used to set the instanceClass attribute of the corresponding EDataType.

```xml
<xsd:simpleType name="date"
    ecore:instanceClass="java.util.Date">
    ...
</xsd:simpleType>
```

```
EDataType
    name="Date"
    instanceClass="java.util.Date"
    ...
```

The "baseType" (see section 2.1) is not recorded in the details map of the extended meta data EAnnotation in this case.

3 Complex Type Definition

Each complex type definition of a schema maps to an EClass in the eClassifiers list of the EPackage corresponding to its target namespace.

The attributes of the EClass corresponding to a complex type are set as follows:

- name = the name of the complex type converted, if necessary, to a proper Java class name (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)
- eAnnotations = an extended meta data EAnnotation

The details map of the extended meta data EAnnotation contains the following entries:

- key = "name", value = the unaltered name of the simple type
- key = "kind", value = one of “empty”, “simple”, “elementOnly”, or “mixed”.

The value of the "kind" details entry depends on the “content type” of the complex type definition (see http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/#content_type).
3.1 Complex type with extension or restriction

If a complex type is an extension or restriction of another complex type, then the base type’s corresponding EClass is added to the eSuperTypes of the EClass.

```xml
<xsd:complexType name="customerReviewType">
  <xsd:complexContent>
    <xsd:extension base="criticsReviewType">
      ...
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

In the case of extension, attribute and element declarations within the body of the extension will also produce features in the EClass as described in the following sections. If the type is a restriction, however, anything in the body will be ignored and the corresponding EClass will contain no new features. The subclass is simply provided to restrict the existing features, for example, to constrain their multiplicity or to make their types narrower.

If the base type of an extension or restriction is a simple type, instead of adding an eSuperType, a single EAttribute with name equal to “value” will be added to the eAttributes of the EClass. The eType of this EAttribute will be the EDataType corresponding to the base of the simpleContent extension.

```xml
<xsd:complexType "...">
  <xsd:simpleContent>
    <xsd:extension base="xsd:int">
      ...
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

3.2 Anonymous complex type

If an anonymous complex type is used for the type of an element declaration, then the corresponding EClass’ name will be set to the enclosing element’s converted name with the suffix “Type” appended. The value of "name" entry in the details map of the extended meta data EAnnotation will have the following value in this case:

```
key = "name", value = the name of enclosing simple type appended with the suffix "_.type"
```

```xml
<xsd:element name="myElement">
  <xsd:complexType>
    ...
  </xsd:complexType>
</xsd:element>
```

3.3 Abstract complex type

An abstract attribute in a complex type is used to set the abstract attribute of the corresponding EClass.

```xml
<xsd:complexType abstract="true" "...
  ...
</xsd:complexType>
```

EClass

```xml
abstract=true
```
3.4 Mixed complex type

A complex type with mixed content will produce a feature map EAttribute named “mixed” in the corresponding EClass. This EAttribute will include the following entries in the details map of its extended meta data EAnnotation:

key = "name", value = ":mixed"
key = "kind", value = "elementWildcard"

```
<xsd:complexType name="MixedType" mixed="true">
    ...
</xsd:complexType>
```

A feature EAnnotation with the special name ":mixed" identifies it as the “mixed” feature for the class, of which there can only be one.

All other features (EReferences and EAttributes) which are mapped from element declarations in the schema will have derived implementations which delegate to the feature map:

```
<xsd:complexType name="customersType" mixed="true">
    <xsd:sequence>
        <xsd:element name="customer" … />
    </xsd:sequence>
</xsd:complexType>
```

3.5 Complex type with ecore:name

The ecore:name attribute can be used to set the name of the EClass, for example, if the corresponding complex type is anonymous or if the default name conversion is unacceptable.

```
<xsd:complexType ecore:name="MyName" mixed="true">
    ...
</xsd:complexType>
```

3.6 Complex type with ecore:featureMap

The ecore:featureMap attribute can be specified on a complex type that has complex content and is not an extension or restriction of another complex type. In this case, it will produce a feature map EAttribute in the corresponding
**XSD to Ecore Mapping**

**EClass**: This feature map is similar to one used when handling a repeating model group (see section 6.2); the implementations of all other features in the class will derive from it. The **name** of the **EAttribute** is set to the value of the **ecore:featureMap** attribute.

The extended meta data **EAnnotation** includes the following entries in its **details** map:

- **key** = "name", **value** = the **name** of the **EAttribute** followed by the string "::group"
- **key** = "kind", **value** = "group"

```xml
<xsd:complexType ecore:featureMap="myMap" … >
  <xsd:sequence>…</xsd:sequence>
</xsd:complexType>
```

If the complex type is mixed (section 3.4) or one that is being treated as if it is (section 3.7), it will already have a feature map based implementation. In this case, the only effect of the ecore:featureMap attribute is to override the name of the “mixed” **EAttribute**.

```xml
<xsd:complexType ecore:featureMap="myMap" mixed="true" … >
  …
</xsd:complexType>
```

### 3.7 Complex type with ecore:mixed

The **ecore:mixed** attribute can be used to produce a feature map based implementation as described in section 3.4, for a complex type that is not actually mixed. The complex type must have complex content and cannot be an extension or restriction of another complex type. This feature is typically used to provide support for adding and accessing comments in an XML document, as opposed to real “mixed text”. Adding “mixed text”, other than white space, to such instances would produce an invalid document, since the type is not really mixed.

```xml
<xsd:complexType ecore:mixed="true" … >
  …
</xsd:complexType>
```

### 4 Attribute Declaration
Each schema attribute declaration maps to an **EAttribute** or **EReference** in the **EClass** corresponding to the complex type definition containing the attribute if locally defined, or in the DocumentRoot **EClass** if the attribute is global.

An attribute declaration maps to an **EReference** in only a few special cases (see section 4.3). Otherwise it maps to an **EAttribute** which is initialized as follows:

- **name** = the name of the attribute converted, if necessary, to a proper Java field name (see [http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307](http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307))
- **eType** = an **EDataType** corresponding to the attribute’s simple type
- **lowerBound** = 0 if use="optional" (default) or 1 if use="required" (see section 4.3)
- **upperBound** = 1
- **eAnnotations** = an extended meta data **EAnnotation**

If the type of the attribute is one of the predefined schema types, then the **eType** of the corresponding **EAttribute** will be one of the **EDataTypes** from the XMLTypePackage as described in section 9. Otherwise, it will be a user defined **EDataType** created from a simple type as described in section 2.

The **details** map of the extended meta data **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered **name** of the attribute
- **key** = "kind", **value** = "attribute"

### 4.1 Attribute of type xsd:ID

An attribute of, or derived from, type xsd:ID maps to an **EAttribute** of the “ID” **EDataType** from the XMLTypePackage (see section 9) as expected, but has the added affect of setting the **iD** attribute of the **EAttribute** to **true**:

```xml
type="xsd:string" />
```

```java
EAttribute
name="title"
eType=".../XMLType#//String"
lowerBound=0
upperBound=1
EAnnotation
details="name->title, kind->attribute"
```

### 4.2 Attribute of type xsd:IDREF, xsd:IDREFS, or xsd:anyURI

Ordinarily, attributes of, or derived from, type IDREF, IDREFS, and anyURI, are handled no differently than those of other predefined schema simple types. They simply map to **EAttributes** with **eType** set to the corresponding **EDataType** in the XMLTypePackage (see section 9).

```xml
type="xsd:ID" />
```

```java
EAttribute
eType=".../XMLType#//ID"
iD=true
```

```xml
type="xsd:IDREF" />
```

```java
EAttribute
name="author"
eType=".../XMLType#//IDREF"
```
If, however, an attribute of one of these three types also includes the ecore:reference attribute, an EReference is created instead. The reference is non-containment (containment equals false) and its eType is set to the EClass corresponding to the complex type specified by the ecore:reference. The upperBound is set to -1 (unbounded) for IDREFS, 1 otherwise. Since IDREF and IDREFS cannot span documents, the resolveProxies property is set to false for them. For anyURI, which can span documents, the resolveProxies property will be set to true:

<table>
<thead>
<tr>
<th>XSD attribute</th>
<th>EReference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:attribute name=&quot;author&quot; type=&quot;xsd:IDREF&quot; ecore:reference=&quot;Writer&quot;/&gt;</code></td>
<td>EReference name=&quot;author&quot; eType=&quot;//Writer&quot; upperBound=1 containment=false resolveProxies=false</td>
</tr>
<tr>
<td><code>&lt;xsd:attribute name=&quot;authors&quot; type=&quot;xsd:IDREFS&quot; ecore:reference=&quot;Writer&quot;/&gt;</code></td>
<td>EReference name=&quot;authors&quot; eType=&quot;//Writer&quot; upperBound=-1 (unbounded) containment=false resolveProxies=false</td>
</tr>
<tr>
<td><code>&lt;xsd:attribute name=&quot;author&quot; type=&quot;xsd:anyURI&quot; ecore:reference=&quot;Writer&quot;/&gt;</code></td>
<td>EReference name=&quot;author&quot; eType=&quot;//Writer&quot; upperBound=1 containment=false resolveProxies=true</td>
</tr>
</tbody>
</table>

If the relationship is bidirectional, ecore:opposite can be used to specify the attribute or element declaration, of the target complex type, that corresponds to the reverse (eOpposite) EReference:

<table>
<thead>
<tr>
<th>XSD attribute</th>
<th>EReference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:attribute name=&quot;author&quot; type=&quot;xsd:anyURI&quot; ecore:reference=&quot;Writer&quot; ecore:opposite=&quot;books&quot;/&gt;</code></td>
<td>EReference name=&quot;author&quot; eType=&quot;//Writer&quot; upperBound=1 containment=false resolveProxies=true</td>
</tr>
</tbody>
</table>

The ecore:opposite attribute can be specified on either (or both) sides of the relationship.

### 4.3 Required attribute

The lowerBound of an EAttribute or EReference corresponding to a required schema attribute is set to 1, instead of the usual 0:

<table>
<thead>
<tr>
<th>XSD attribute</th>
<th>EAttribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsd:attribute use=&quot;required&quot; … /&gt;</code></td>
<td>EAttribute lowerBound=1</td>
</tr>
</tbody>
</table>
4.4 Attribute with default

An attribute with a default value will set the `defaultValueLiteral` attribute of the corresponding `EAttribute`. The `EAttribute` will also be `unsettable` in this case:

```
<xsd:attribute name="message" type="xsd:string"
        default="hello world" … />
```

An attribute declaration without an explicit default value may also map to an `unsettable` `EAttribute` if the type has an *intrinsic* default value that is not equal to `null` (that is, the corresponding `eType` is an `EEnum` or an `EDataType` representing a primitive Java type):

```
<xsd:attribute name="pages" type="xsd:int"/>
```

An attribute declaration whose type is an enumeration restriction of a simple type that maps to a primitive Java type (for example, `int`) will have its default value set, even if no explicit default value is specified for the attribute. In this case, the `defaultValueLiteral` of the corresponding `EAttribute` will be set to the value corresponding to the first enumeration value of the type:

```
<xsd:attribute name="oneThreeFive">
    <xsd:simpleType>
        <xsd:restriction base="xsd:int">
            <xsd:enumeration value="1"/>
            <xsd:enumeration value="3"/>
            <xsd:enumeration value="5"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
```

4.5 Qualified attribute

If a local attribute declaration has qualified form, either explicitly declared with the `form` attribute set to “qualified” or inherited from a schema element with `attributeFormDefault` set to “qualified” (see section 1.7), then the `details` map of the extended meta data `EAnnotation` for the corresponding feature will contain an additional entry:

```
key = "namespace", value = "##targetNamespace"
```

4.6 Attribute reference

An attribute reference (that is one with a ref attribute) will produce a “namespace” entry in the `details` map of the extended meta data `EAnnotation` of the corresponding feature, exactly as described for qualified attributes (see...
section 4.6). If, however, the reference is to a global attribute in a different schema, then the value of the “namespace” entry will be set to the targetNamespace of the global attribute, instead of ##targetNamespace:

```xml
<xsd:schema xmlns:some="http://someSchema" ...
  xmlns:someGlobal="http://someSchema" ...
</xsd:schema>```

**4.7 Global attribute**

The EAttribute or EReference corresponding to a global attribute declaration is added to the package’s DocumentRoot class as described in section 1.5. The extended meta data EAnnotation corresponding to a global attribute declaration will also include exactly the same “namespace” details entry (with value “##targetNamespace”) as a qualified attribute described in section 4.5.

```xml
<xsd:schema ... 
  xmlns:global="http://globalSchema"
  xmlns:globalAttribute="http://globalSchema" ...
</xsd:schema>```

**4.8 Attribute with.ecore:name**

The .name attribute can be used to explicitly set (override) the name of the EAttribute, if the default name conversion is unacceptable.

```xml
<xsd:attribute name="..." ecore:name="MyName" ... />
```

In the case of an attribute reference, a local .name attribute takes precedence over an .name setting on the referenced global attribute, if there is one.

**5 Element Declaration**

Each schema element declaration maps to an EAttribute or EReference in the EClass corresponding to the complex type definition containing the element, or in the DocumentRoot EClass if the element is global.

An element declaration maps to an EAttribute if its type is simple (with the exception of the special cases described in section 5.3). Otherwise, if the type is complex, it maps to an EReference. In either case, the attributes of the feature are initialized as follows:

- **name** = the name of the element converted, if necessary, to a proper Java field name (see http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)
- **eType** = an EDataType or EClass corresponding to the element’s type
- **lowerBound** = the minOccurs value of the element declaration multiplied by the minOccurs of any containing model groups, or 0 if the element is nested in a choice or is not in a content model
- **upperBound** = the maxOccurs value of the element declaration multiplied by the maxOccurs of any containing model groups, or -2 if the element declaration is not in a content model (see section 5.8)
eAnnotations = an extended meta data EAnnotation

If the type of the element is one of the predefined schema types, then the eType of the corresponding EAttribute will be one of the EDataTypes from the XMLTypePackage as described in section 9. Otherwise, it will be a user defined EDataType created from a simple type as described in section 2, or if the element declaration maps to an EReference, then the eType will be the EClass corresponding to the element’s type. If an EReference, the containment property will be true, except for the cases described in section 5.3.

The details map of the extended meta data EAnnotation will contain the following entries:

key = "name", value = the unaltered name of the element
key = "kind", value = "element"

```
<xsd:element name="mySimple" type="xsd:string"
  maxOccurs="unbounded" />
```

```
<xsd:element name="myComplex">
  <xsd:complexType … >
    …
  </xsd:complexType>
</xsd:element>
```

5.1 Element of type xsd:anyType

In addition to the EDataTypes for all the XML Schema predefined simple types (see section 9), the XMLTypePackage also includes an EClass, "AnyType", corresponding to the xsd:anyType complex type. However, an element of type xsd:anyType does not map to an EReference of this type as you might expect. Instead, the eType of the corresponding EReference will be EObject, the base class of all EMF Objects:

```
<xsd:element name="…" type="xsd:anyType"/>
```

```
<xsd:element name="…" type="xsd:anyType"/>
```

Using EObject for the reference type allows an instance of any EMF object to be the value of the feature, which is the intended behavior. The purpose of the “AnyType” EClass is to handle situations where an instance contains arbitrary XML content. For example, when processing wildcard content in “lax mode” with no meta data available, an instance of the “AnyType” EClass, which like every other EClass is a subtype of EObject, will be used as the value of the feature. An instance of class “AnyType” can represent any arbitrary XML element content including any attributes and mixed text that it may have.

5.2 Element of type xsd:ID

Note: the XML Schema specification recommends avoiding the use of xsd:ID for the type of an element declaration.
An element of, or derived from, type xsd:ID maps to an EAttribute of the “ID” EDataType from the
XMLTypePackage (see section 9) as expected, but has the added affect of setting the iD attribute of the EAttribute
to true:

```xml
<xsd:element name="…" type="xsd:ID"/>
```

**5.3 Element of type xsd:IDREF, xsd:IDREFS, or xsd:anyURI**

*Note: the XML Schema specification recommends avoiding the use of xsd:IDREF or xsd:IDREFS for the type of an
element declaration.*

Elements of, or derived from, type IDREF or anyURI are given the same special treatment as described for attributes
of these types (see section 4.3); when an.ecore:reference is specified, they map to EReferences instead of being
treated as ordinary elements of simple type (which always map to EAttributes). Unlike attributes, however, elements
can be repeated, so the upperBound of the EReference is not always 1 (as described in section 4.3), but is instead
set according to the maxOccurs attribute of the element declaration, in the usual way:

```xml
<xsd:element name="author" type="xsd:anyURI"
           maxOccurs="10"
           ecore:reference="Writer"/>
```

The IDREFS element case is a little more complicated because the set of references represented by an IDREFS
element can themselves be repeated (that is, maxOccurs might be greater than 1). So, in this case the EReference’s
containment property is made true and the eType of the EReference is set to a special “Holder” EClass, instead
of to the type specified by the ecore:reference attribute:

```xml
<xsd:element name="authors" type="xsd:IDREFS"
             ecore:reference="Writer"/>
```

Such a “Holder” EClass is automatically created for every element declaration of type IDREFS with an
ecore:reference attribute specified. This EClass is initialized as follows:

- **name** = the name of the complex type converted, if necessary, to a proper Java class name (see
  http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307), and with the string
  “Holder” appended
- **eReferences** = a single multiplicity-many EReference
- **eAnnotations** = an extended meta data EAnnotation

The details map of the extended meta data EAnnotation for the EClass will have the following entries:

- **key** = "name", **value** = the name of the attribute with the string “:holder” appended
- **key** = "kind", **value** = “simple”
The **EReference** in the “Holder” **EClass** will have the following values:

- `name = "value"`
- `eType = the value of the.ecore:reference attribute`
- `upperBound = -1 (unbounded)`
- `containment = false`
- `resolveProxies = false`

The details map of the extended meta data **EAnnotation** for the “value” **EReference** will contain the following:

- `key = "name", value = ":0"
- `key = "kind", value = "simple"`

```xml
<xsd:element name="authors" type="xsd:IDREFS"
    ecore:reference="Writer"/>
```

### 5.4 Nillable element

A **nillable** element with **maxOccurs** equal to 1 will produce an **EAttribute** with **unsettable** set to **true**.

If the **EDataType** corresponding to a nillable element’s **type** has a Java primitive type as its **instanceClass** (for example, int), then an **EDataType** for a Java wrapper type (for example, java.lang.Integer) will be used as the **eType** instead of the usual one (see table in section 9).

```xml
<xsd:element nillable="true" type="xsd:int" … />
```

If the type of the element is an enumeration (see section 2.3) then the **eType** will be set to a wrapper **EDataType** for “org.eclipse.emf.common.util.AbstractEnumerator”, instead of the corresponding **EEnum** itself.

```xml
<xsd:element nillable="true" type="USState" … />
```

Such “Object” **EDataType**’s are automatically created for every user defined enumeration or primitive type in a schema. The attributes of a wrapper **EDataType** are initialized as follows:
**5.5 Element with default**

An element declaration with a default value will set the `defaultValueLiteral` attribute of the corresponding `EAttribute`. The `EAttribute` will also be `unsettable` in this case:

```xml
<xsd:element name="message" type="xsd:string"
    default="hello world" … />
```

```java
EAttribute
    eType="…/XMLType#//String"
    defaultValueLiteral="hello world"
    unsettable=true
```

An element declaration without an explicit default value may also map to an `unsettable EAttribute` if the type has an intrinsic default value that is not equal to `null` (that is, the corresponding `eType` is an `EEnum` or an `EDataType` representing a primitive Java type):

```xml
<xsd:element name="pages" type="xsd:int"/>
```

```java
EAttribute
    eType="…/XMLType#//Int"
    unsettable=true
```

An element declaration whose type is an enumeration restriction of a simple type that maps to a primitive Java type (for example, `int`) will have its default value set, even if no explicit default value is specified for the element. In this case, the `defaultValueLiteral` of the corresponding `EAttribute` will be set to the value corresponding to the first enumeration value of the type:

```xml
<xsd:element name="oneThreeFive">
    <xsd:simpleType>
        <xsd:restriction base="xsd:int">
            <xsd:enumeration value="1"/>
            <xsd:enumeration value="3"/>
            <xsd:enumeration value="5"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
```

```java
EAttribute
    eType="…/XMLType#//Int"
    defaultValueLiteral="1"
    unsettable=true
```

**5.6 Qualified element**
A local element declaration with qualified form, either explicitly declared with the form attribute set to "qualified" or inherited from a schema with elementFormDefault set to "qualified" (see section 1.7), then the details map of the extended meta data EAnnotation for the corresponding feature will contain an additional entry:

key = "namespace", value = "##targetNamespace"

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element form="qualified" … />
  </xsd:sequence>
</xsd:complexType>
```

### 5.7 Element reference

An element reference (that is one with a ref attribute) will produce a “namespace” entry in the details map of the extended meta data EAnnotation of the corresponding feature, exactly as described for qualified elements (see section 5.6). If, however, the reference is to a global element in a different schema, then the value of the namespace entry will be set to the targetNamespace of the global element, instead of ##targetNamespace:

```
<xsd:schema xmlns:some="http://someSchema" … >
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="some:globalElement"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

### 5.8 Global element

The EAttribute or EReference corresponding to a global element declaration is added to the package’s DocumentRoot class as described in section 1.5. The upperBound of the feature is set to -2 (unspecified). The extended meta data EAnnotation corresponding to a global element declaration will also include exactly the same “namespace” details entry (with value “##targetNamespace”) as a qualified element described in section 5.6:

```
<xsd:schema … >
  <xsd:element name="zip" type="zipCode"/>
  …
</xsd:schema>
```

### 5.9 Element with substitution group

An element declaration which includes a substitutionGroup attribute will produce an additional entry in the details map of the extended meta data EAnnotation of the corresponding EReference or EAttribute (for simple type):
key = "affiliation", value = the value of the substitutionGroup attribute

Any element declaration that is the head of a substitution group (from which other elements derive), like "criticsReview" in this example, will produce a feature map based implementation in the EClasses corresponding to any referencing elements. In addition to the normal EReference in the corresponding EClass, a feature map EAttribute will also be created. The name of the FeatureMap EAttribute will be the same as the name of the element’s corresponding EReference but with the string “Group” appended. The element’s corresponding EReference will be derived from the FeatureMap:

This feature map based implementation is required to allow instances of the substitution elements to be serialized in an XML document without using an xsi:type attribute. If this isn’t needed for the model in question, the feature map implementation can be suppressed using an.ecore:featureMap as described in section 5.13:

If the referencing element is nested within a schema component for which a feature map already exists (if the containing complex type is "mixed", for example), the feature map EAttribute (criticsReviewGroup in this example) will itself be derived from the containing feature map.

5.10 Abstract element
If an element declaration is `abstract`, then the same kind of feature map based implementation as described for the head of a substitution group element in the previous section (see section 5.9) will result. In this case, however, the corresponding feature will also be non `changeable`.

```xml
<xsd:element name="address" abstract="true"
    type="addressType" />
```

```xml
EAttribute
name="addressGroup"
<reference type="…/Ecore#/EFeatureMapEntry"
    upperBound=-1 (unbounded)

EReference
name="address"
<type="/AddressType"
    volatile=true
    transient=true
    derived=true (derived from "addressGroup")
    changeable=false
```

### 5.11 Element with ecore:name

The `ecore:name` attribute can be used to explicitly set (override) the `name` of the corresponding `EAttribute` or `EReference`, if the default name conversion is unacceptable.

```xml
<xsd:element name="…" ecore:name="MyName" … />
```

```xml
EAttribute
name="MyName"
```

In the case of an element reference, a local `ecore:name` attribute takes precedence over an `ecore:name` setting on the referenced global element, if there is one.

### 5.12 Element with ecore:opposite

Any element declaration that maps to an `EReference` can use the `ecore:opposite` attribute to specify the reverse (`eOpposite`) `EReference`, if the relationship is bidirectional. If the relationship is non-containment (see section 5.3), then the `ecore:opposite` specifies an attribute or element declaration in the target complex type, as described in section 4.2. Otherwise, it simply specifies the name of a type-safe container (`eContainer`) reference in the target `EClass`.

```xml
<xsd:element name="books" type="Book"
    maxOccurs="unbounded"
    ecore:opposite="library"/>
```

```xml
EReference
name="books"
<type="/Book"
    upperBound=-1 (unbounded)
    containment=true
    eOpposite="/Book/library"
```

### 5.13 Element with ecore:featureMap

The `ecore:featureMap` attribute can be used to introduce the same kind of feature map based implementation as described for the head of a substitution group element in section 5.9, or to rename a feature map feature that already exists:
Alternatively, an unwanted feature map implementation can be suppressed by setting an `ecore:featureMap` attribute to "" (empty string), which will revert to the ordinary (non derived) implementation pattern for the element.

```
<xsd:element name="myElement" type="addressType"
  ecore:featureMap=""
/>  
```

### 6 Model group

XML Schema model groups (xsd:sequence, xsd:choice, and xsd:all), with `maxOccurs` equal to 1 (the default) produce no corresponding elements in the Ecore model. These constructs simply serve to aggregate the elements under them. In Ecore, the `EClass` corresponding to the containing complex type already provides this aggregation function for the corresponding features. The only case requiring special treatment is a xsd:choice, which imposes certain exclusivity restrictions on the setting of the features corresponding to the elements within it, as described in the following section.

#### 6.1 Non repeating xsd:choice

**Note: implementation of this is TBD**

A non repeating xsd:choice (that is, with `maxOccurs = 1`) has no representation in the corresponding Ecore model, but it does have an effect on the extended meta data `EAnnotation` of the features corresponding to its nested elements. Each such feature will include the following additional entry in the `details` map:

```
key = "exclusion", value = a space separated list of features that can’t be set at the same time as this one
```

```
<xsd:choice>
  <xsd:element name="element1" type="xsd:string" />  
  <xsd:element name="element2" … />  
  <xsd:element name="element3" … />  
</xsd:choice>
```

#### 6.2 Repeating model group
A model group (xsd:sequence, xsd:choice, or xsd:all) with \( \text{maxOccurs} > 1 \) produces a feature map EAttribute in the EClass corresponding to the complex type definition containing the group. The EAttribute is initialized as follows:

- **name** = “group”
- **eType** = EFeatureMapEntry
- **lowerBound** = the minOccurs value of the model group multiplied by the minOccurs of any containing model groups, or 0 if the group is nested in a choice
- **upperBound** = the maxOccurs value of the model group multiplied by the maxOccurs of any containing model groups
- **eAnnotations** = an extended meta data EAnnotation

The details map of the extended meta data EAnnotation for the EAttribute will have the following entries:

- **key** = "name", **value** = the name of the EAttribute followed by ";" and the feature ID of the EAttribute
- **key** = "kind", **value** = "group"

```xml
<xsd:choice maxOccurs="unbounded">
  ...  
</xsd:choice>
```

```xml
EAttribute
  name="group"
  eType=".../Ecore#/EFeatureMapEntry"
  upperBound=-1 (unbounded)
  EAnnotation
    source=".../ExtendedMetaData"
    details="name->group:0,
             kind->group"
```

All other EReferences and EAttributes which are mapped from element declarations in the schema will have derived implementations which delegate to the feature map:

```xml
<xsd:choice>
  <xsd:element name="element1" type="xsd:float"/>
  ...
</xsd:sequence>
```

```xml
EAttribute
  name="element1"
  volatile=true
  transient=true
  derived=true (derived from "group")
  ...
  EAnnotation
    source=".../ExtendedMetaData"
    details="group->#group:0, ...
```

If the model group is nested within a schema component for which a feature map already exists (if the containing complex type is “mixed”, for example), the feature map EAttribute (group:0 in this example) will itself be derived from the containing feature map.

### 6.3 Repeating model group reference

The feature map EAttribute corresponding to a repeating reference to a model group definition (xsd:group) will have its name set to that of the model group definition, instead of “group” as described in section 6.2. The name will be converted to a proper Java field name (see [http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307](http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#73307)), if necessary.

```xml
<xsd:group name="aGroup">
  <xsd:choice>
    ...
  </xsd:choice>
</xsd:group>
```

```xml
EClass
  name="..."
EAttribute
  name="aGroup"
```
6.4 Model group with.ecore:featureMap attribute

The .ecore:featureMap attribute can be used to override the name of the feature map .EAttribute corresponding to a repeating model group:

```
<xsd:choice maxOccurs="unbounded"
    ecore:featureMap="choices">
    ...
</xsd:choice>
```

Alternatively, if order preservation among the elements in the group is not desired, the feature map implementation can be suppressed by setting an .ecore:featureMap attribute to "" (empty string), which will revert to the ordinary (non derived) implementation pattern for the elements in the group:

```
<xsd:choice maxOccurs="unbounded"
    ecore:featureMap="">
    ...
</xsd:choice>
```

No feature map attribute produced

Finally, the .ecore:featureMap attribute can be used on a non repeating model group to produce a feature map implementation, just like the one produced for a group that is repeating. One common use of this is to provide order preservation to an xsd:all group. By definition, an xsd:all is one where the group’s elements can appear in any order. By default EMF interprets this as meaning serialization order is irrelevant, so no feature map is provided. The other interpretation of an xsd:all is that the elements can be any order, but the order they’re in is important and cannot change. If this is the desired behavior, an .ecore:featureMap attribute can be used to override the simpler default mapping to produce a feature map for the group:

```
<xsd:all ecore:featureMap="allMap">
    ...
</xsd:all>
```

6.5 Model group definition with.ecore:name attribute

The .ecore:name attribute can be used to override the .name of a feature map .EAttribute for a model group reference.
Note that the same effect could be achieved using the ecore:featureMap attribute on the xsd:choice itself. If both attributes are provided, the ecore:featureMap attribute would take precedence.

7 Wildcards

Element wildcards (xsd:any) and attribute wildcards (xsd:anyAttribute) both map to a feature map EAttribute in the EClass corresponding to the complex type definition containing the wildcard. The EAttribute is initialized as follows:

- **name** = “any” for xsd:any or “anyAttribute” for xsd:anyAttribute
- **eType** = EFeatureMapEntry
- **lowerBound** = 0 for xsd:anyAttribute or an xsd:any that is nested in a choice, otherwise the minOccurs value of the xsd:any multiplied by the minOccurs of any containing model groups
- **upperBound** = -1 (unbounded) for xsd:anyAttribute or the maxOccurs value of the xsd:any multiplied by the maxOccurs of any containing model groups
- **eAnnotations** = an extended meta data EAnnotation

The case where **upperBound** is 1, is somewhat special; it is still implemented using a feature map, as opposed to just a feature map entry. In this case, the feature map will be restricted to contain only a single entry.

The **details** map of the extended meta data EAnnotation for the EAttribute will have the following entries:

- **key** = "name", **value** = ":\" followed by the feature ID of the EAttribute
- **key** = "kind", **value** = "elementWildcard" (for xsd:any) or “attributeWildcard” (for xsd:anyAttribute)
- **key** = "wildcards", **value** = the value of the namespace attribute of the wildcard
- **key** = "processing", **value** = the value of the processContents attribute of the wildcard
If the wildcard is nested within a schema component for which a feature map already exists (if the containing complex type is “mixed”, for example), the wildcard EAttribute (feature map) will be derived from the containing feature map.

7.1 Wildcard with ecore:name

The ecore:name attribute can be used to set the name of a wildcard EAttribute to something other than the default values of “any” or “anyAttribute”.

```
<xsd:any namespace="##any"
    ecore:name="myExtension"/>
```

7.2 Wildcard with processContents="lax"

A wildcard with processContents set to “lax”, has no special effect on the model, other than to set the value of the “processing” entry in the EAnnotation’s detail map:

```
<xsd:any namespace="##any"
    processContents="lax"/>
```

This can, however, have a significant effect on an instance. In this situation, instances of the “AnyType” EClass, from the XMLTypePackage (described in section 5.1), will be used as the values in the feature map to represent any arbitrary (unresolvable) XML element content within the wildcard.

8 Annotations

Documentation and appinfo elements of an annotation component both map to an EAnnotation in the eAnnotations list of the corresponding Ecore element.

8.1 Documentation annotation

The source attribute of an EAnnotation corresponding to a schema documentation element will be set to the value "http://www.eclipse.org/emf/2002/GenModel". This special URI is used by the EMF generator to identify EAnnotations containing documentation to be generated into the JavaDoc comments of the corresponding Java code.

The details map of the documentation EAnnotation will contain a single entry:
**key** = "documentation", **value** = the contents of the documentation element

```xml
<xsd:annotation>
    <xsd:documentation xml:lang="en">
        some information
    </xsd:documentation>
</xsd:annotation>
```

**EAnnotation**
- **source** = ".../emf/2002/GenModel"
- **details** = "documentation-> some information  "

A single EAnnotation instance is used to represent all the documentation elements of a given schema construct, should there be more than one. In this case, the value of the “documentation” entry in the details map will simply contain the concatenation of the individual documentation elements:

```xml
<xsd:annotation>
    <xsd:documentation xml:lang="en">
        some information
    </xsd:documentation>
    <xsd:documentation xml:lang="en">
        more information
    </xsd:documentation>
</xsd:annotation>
```

**EAnnotation**
- **source** = ".../emf/2002/GenModel"
- **details** = "documentation-> some information
more information  "

### 8.2 AppInfo annotation

The source attribute of an EAnnotation corresponding to schema appinfo will be set to the value of a source attribute, if provided or null otherwise. The handling of appinfo contents is essentially the same as for documentation as described in the previous section (section 8.1), only using "appinfo" for the details entry instead of “documentation”.

```xml
<xsd:annotation>
    <xsd:appinfo source="http://myURI">
        <junk>hello</junk>
    </xsd:appinfo>
</xsd:annotation>
```

**EAnnotation**
- **source** = "http://myURI"
- **details** = "appinfo-><junk>hello</junk>"

### 8.3 Annotation with ecore:ignore

The ecore:ignore attribute can be specified on an xsd:annotation to suppress the mapping of all its documentation and appinfo children:

```xml
<xsd:annotation ecore:ignore="true">
    ...
</xsd:annotation>
```

No documentation or appinfo entries in details map

Alternatively, the ecore:ignore attribute can be specified on individual documentation or appinfo elements to suppress only those entries:

```xml
<xsd:annotation>
    <documentation ecore:ignore="true" >
        doc1
    </documentation>
    ...
</xsd:annotation>
```

No documentation entry for “doc1” in details map
9 Predefined Schema Simple Types

Each predefined XML Schema simple type maps to a corresponding built-in `EDataType` in an EMF package named `XMLTypePackage` and with namespace URI "http://www.eclipse.org/emf/2003/XMLType". The following table lists the complete set of Schema simple types along with the values of the `name` and `instanceClass` attributes of their corresponding `EDataType`.

<table>
<thead>
<tr>
<th>XML Schema Simple Type</th>
<th>EDataType (in XMLTypePackage)</th>
<th>instanceClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>anySimpleType</td>
<td>AnySimpleType</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>anyURI</td>
<td>AnyURI</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>base64Binary</td>
<td>Base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>boolean (nillable=&quot;true&quot;)</td>
<td>BooleanObject</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
<td>byte</td>
</tr>
<tr>
<td>byte (nillable=&quot;true&quot;)</td>
<td>ByteObject</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>date</td>
<td>Date</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>dateTime</td>
<td>DateTime</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>decimal</td>
<td>Decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>double</td>
</tr>
<tr>
<td>double (nillable=&quot;true&quot;)</td>
<td>DoubleObject</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>duration</td>
<td>Duration</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>ENTITIES</td>
<td>java.util.List</td>
</tr>
<tr>
<td>ENTITY</td>
<td>ENTITY</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>float</td>
</tr>
<tr>
<td>float (nillable=&quot;true&quot;)</td>
<td>FloatObject</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>gDay</td>
<td>GDay</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>gMonth</td>
<td>GMonth</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>GMonthDay</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>gYear</td>
<td>GYear</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>GYearMonth</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>hexBinary</td>
<td>HexBinary</td>
<td>byte[]</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>IDREF</td>
<td>IDREF</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>IDREFS</td>
<td>IDREFS</td>
<td>java.util.List</td>
</tr>
<tr>
<td>int</td>
<td>Int</td>
<td>int</td>
</tr>
<tr>
<td>int (nillable=&quot;true&quot;)</td>
<td>IntObject</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>integer</td>
<td>Integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>language</td>
<td>Language</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>long</td>
</tr>
<tr>
<td>long (nillable=&quot;true&quot;)</td>
<td>LongObject</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NCName</td>
<td>NCName</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>NegativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>NMTOKEN</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>NMTOKENS</td>
<td>NMTOKENS</td>
<td>java.util.List</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>NonNegativeInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>NonPositiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>normalizedString</td>
<td>NormalizedString</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>NOTATION</td>
<td>NOTATION</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>PositiveInteger</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>QName</td>
<td>QName</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>short</td>
</tr>
<tr>
<td>short (nillable=&quot;true&quot;)</td>
<td>ShortObject</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>time</td>
<td>Time</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>token</td>
<td>Token</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>UnsignedByte</td>
<td>short</td>
</tr>
<tr>
<td>unsignedByte (nillable=&quot;true&quot;)</td>
<td>UnsignedByteObject</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>UnsignedInt</td>
<td>long</td>
</tr>
<tr>
<td>unsignedInt (nillable=&quot;true&quot;)</td>
<td>UnsignedIntObject</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>UnsignedLong</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>UnsignedShort</td>
<td>int</td>
</tr>
<tr>
<td>unsignedShort (nillable=&quot;true&quot;)</td>
<td>UnsignedShortObject</td>
<td>java.lang.Integer</td>
</tr>
</tbody>
</table>