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 **EAnnotation**s 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 then 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 **EAttribute**s 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:

nsURI = the URI of the schema document
nsPrefix = last segment of the URI (short file name), excluding the file extension
name = same as **nsPrefix**
eAnnotations = an extended meta data **EAnnotation**

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

key = "qualified", **value** = "false"

in resource: file:/c:/myexample/library.xsd <xsd:schema> … </xsd:schema>	**EPackage** **name**="library" **nsPrefix**="library" **nsURI**="file:/c:/myexample/library.xsd" **EAnnotation** **source**="…/ExtendedMetaData" **details**="qualified->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:

nsURI = the targetNamespace value
nsPrefix = a last segment of the Java package name (derived from the targetNamespace)
name = same as **nsPrefix**

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:

<xsd:schema targetNamespace="http://www.example.com/libary"> … </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**:

<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.

<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.

<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>	**EPackage** **EClass** **name**="LibraryRoot" …

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>	**EPackage** …

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

<xsd:simpleType name="zipCodes"> <xsd:restriction base="xsd:integer"> <xsd:minInclusive value="10000"/> <xsd:maxInclusive value="99999"/> </xsd:restriction> </xsd:simpleType>	**EDataType** **name**="ZipCodes" **instanceClass**="int" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->zipCodes, baseType->…/XMLType#integer, minInclusive->10000, maxInclusive->99999"

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

<xsd:minInclusive value="10000" ecore:ignore="true"/>	*No minInclusive entry in **details** map*

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)
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

<xsd:simpleType name="USState"> <xsd:restriction base="xsd:string"> <xsd:enumeration value="AK"/> <xsd:enumeration value="AL"/> <!-- and so on ... --> </xsd:restriction> </xsd:simpleType>	**EEnum** **name**="USState" **EEnumLiteral name**="AK" **value**=0 **EEnumLiteral name**="AL" **value**=1 **EAnnotation** **source**="…/ExtendedMetaData" **details**="name-> USState"

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), **EAttribute**s 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 itemType of the list

<xsd:simpleType name="actorsList"> <xsd:list itemType="xsd:NCName"/> </xsd:simpleType>	**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

<xsd:simpleType name="zipUnion"> <xsd:union memberTypes="zipCodes USState"/> </xsd:simpleType>	**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 **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 **EAnnotation** will have the following value in this case:

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

<xsd:element name="myElement"> <xsd:simpleType> … </xsd:simpleType> *</element>*	**EDataType** **name**="MyElementType" **EAnnotation** **details**="name->myElement_._type, …" …

If an anonymous simple type is used as the base type of a restriction, then the corresponding **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 **EAnnotation** will have the suffix "_._base" in this case:

<xsd:simpleType name= "myType"> *<xsd:restriction>* <xsd:simpleType> … </xsd:simpleType> *</xsd:restriction>* *</xsd:simpleType>*	**EDataType** **name**="MyTypeBase" **EAnnotation** **details**="name->myType_._base, …" …

Similarly, if an anonymous simple type is used as the item type of a list, then the corresponding **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 **EAnnotation** will have the suffix "_._item":

<xsd:simpleType name= "myType"> *<xsd:list>* <xsd:simpleType> … </xsd:simpleType> *</xsd:list>* *</xsd:simpleType>*	**EDataType** **name**="MyTypeItem" **EAnnotation** **details**="name->myType_._item, …" …

Finally, if an anonymous simple type is used as a member type of a union, then the corresponding **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 **EAnnotation** will have the suffix "_._member" also qualified with the position number.

<xsd:simpleType name= "myType"> *<xsd:union>* <xsd:simpleType> … </xsd:simpleType> … *</xsd:union>*	**EDataType** **name**="MyTypeMember0" **EAnnotation** **details**="name->myType_._member0, …" …

</xsd:simpleType>	

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.

<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**.

<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).

<xsd:complexType name="customerReviewType"> <xsd:complexContent> … </xsd:complexContent> </xsd:complexType>	**EClass** **name**="CustomerReviewType" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->customerReviewType, kind->elementOnly"

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**.

<xsd:complexType name="customerReviewType"> <xsd:complexContent> <xsd:extension base="criticsReviewType"> … </xsd:extension> </xsd:complexContent> </xsd:complexType>	**EClass** **eSuperTypes**="#//criticsReviewType" …

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.

<xsd:complexType … > <xsd:simpleContent> <xsd:extension base="xsd:int"> … </xsd:extension> </xsd:simpleContent> </xsd:complexType>	**EClass** **EAttribute** **name**="value" **eType**="…/XMLType#//Int" …

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"

<xsd:element name="myElement"> <xsd:complexType> … </xsd:complexType> *</element>*	**EClass** **name**="MyElementType" **EAnnotation** **details**="name->myElement_._type, … …

3.3 Abstract complex type

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

<xsd:complexType abstract="true" … > … </xsd:complexType>	**EClass** **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>	**EClass** **name**="MixedType" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->MixedType, kind->mixed" **EAttribute** **name**="mixed" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->:mixed, kind->elementWildcard" …

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 (**EReference**s and **EAttribute**s) 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>	*EClass name="CustomersType" …* **EAttribute** **name**="customer" **volatile**=true **transient**=true **derived**=true *(derived from "mixed")* …

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" … </xsd:complexType>	**EClass** **name**="MyName" …

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

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"

<xsd:complexType ecore:featureMap="myMap" … > <xsd:sequence> … </xsd:sequence> </xsd:complexType>	**EClass** … **EAttribute** **name**="myMap" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->myMap:group, kind->group" …

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**.

<xsd:complexType ecore:featureMap="myMap" mixed="true" … > … </xsd:complexType>	**EClass** … **EAttribute** **name**="myMap" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->:mixed, kind->elementWildcard" …

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.

<xsd:complexType ecore:mixed="true" … > … </xsd:complexType>	**EClass** … **EAttribute** **name**="mixed" **eType**="…/Ecore#//EFeatureMapEntry" …

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)
> **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 **EDataType**s 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"

<xsd:attribute name="title" type="xsd:string"/>	**EAttribute** **name**="title" **eType**="…/XMLType#//String" **lowerBound**=0 **upperBound**=1 **EAnnotation** **details**="name->title, kind->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`:

<xsd:attribute name="ID" type="xsd:ID"/>	**EAttribute** **eType**="…/XMLType#//ID" **iD**=true …

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 **EAttribute**s with **eType** set to the corresponding **EDataType** in the XMLTypePackage (see section 9).

<xsd:attribute name="author" type="xsd:IDREF"/>	**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`:

<xsd:attribute name="author" type="xsd:IDREF" ecore:reference="Writer"/>	**EReference** **name**="author" **eType**="//Writer" **upperBound**=1 **containment**=false **resolveProxies**=false …
<xsd:attribute name="authors" type="xsd:IDREFS" ecore:reference="Writer"/>	**EReference** **name**="authors" **eType**="//Writer" **upperBound**=-1 *(unbounded)* **containment**=false **resolveProxies**=false …
<xsd:attribute name="author" type="xsd:anyURI" ecore:reference="Writer"/>	**EReference** **name**="author" **eType**="//Writer" **upperBound**=1 **containment**=false **resolveProxies**=true …

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**:

<xsd:attribute name="author" type="xsd:anyURI" ecore:reference="Writer" ecore:opposite="books"/>	**EReference** **name**="author" **eType**="//Writer" **upperBound**=1 **containment**=false **resolveProxies**=true …

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:

<xsd:attribute use="required" … />	**EAttribute** **lowerBound**=1 …

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" … />	**EAttribute** **eType**="…/XMLType#//String" **defaultValueLiteral**="hello world" **unsettable**=true …

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"/>	**EAttribute** **eType**="…/XMLType#//Int" **unsettable**=true …

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>	**EAttribute** **eType**="…/XMLType#//Int" **defaultValueLiteral**="1" **unsettable**=true …

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"

<xsd:attribute form="qualified" … />	**EAttribute** **EAnnotation** **details**="namespace->##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:

<xsd:schema xmlns:some="http://someSchema" … > <xsd:attribute ref="some:globalAttribute"/> *</xsd:schema>*	**EAttribute** **EAnnotation** **details**="namespace->http://someSchema", …" …

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.

<xsd:schema ... > <xsd:attribute name="globalAttribute" type="xsd:string"/> ... *</xsd:schema>*	*DocumentRoot EClass* **EAttribute** **name**="globalAttribute" **eType**="…/XMLType#//String" … **EAnnotation** **details**="namespace->##targetNamespace, …" …

4.8 Attribute with ecore:name

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

<xsd:attribute name="…" ecore:name="MyName" … />	**EAttribute** **name**="MyName" …

In the case of an attribute reference, a local ecore:name attribute takes precedence over an ecore: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 **EDataType**s 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" />	**EAttribute** **name**="mySimple" **eType**="…/XMLType#//String" **lowerBound**=1 **upperBound**=-1 *(unbounded)* **EAnnotation** **details**="name->mySimple, kind->element"
<xsd:element name="myComplex"> <xsd:complexType … > … </xsd:complexType> </xsd:element>	**EReference** **name**="myComplex" **eType**="//MyComplexType" **lowerBound**=1 **upperBound**=1 **containment**=true **EAnnotation** **details**="name->myComplex, kind->element"

5.1 Element of type xsd:anyType

In addition to the **EDataType**s 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"/>	**EReference** **eType**="…/Ecore#//EObject" …

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`:

<xsd:element name="…" type="xsd:ID"/>	**EAttribute** **eType**="…/XMLType#//ID" **iD**=true …

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 **EReference**s instead of being treated as ordinary elements of simple type (which always map to **EAttribute**s). 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:

<xsd:element name="author" type="xsd:anyURI" maxOccurs=="10" ecore:reference="Writer"/>	**EReference** **name**="author" **eType**="//Writer" **upperBound**=10 **containment**=false **resolveProxies**=true …

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:

<xsd:element name="authors" type="xsd:IDREFS" ecore:reference="Writer"/>	**EReference** **name**="authors" **eType**="//AuthorsHolder" **containment**=true …

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" **ERererence** will contain the following:

key = "name", **value** = ":0"
key = "kind", **value** = "simple"

<xsd:element name="authors" type="xsd:IDREFS" ecore:reference="Writer"/>	**EClass** **name**="AuthorsHolder" **EAnnotation** **details**="name->authors:holder, kind->simple" … **EReference** **name**="value" **eType**="//Writer" **upperBound**=-1 *(unbounded)* **containment**=false **resolveProxies**=false **EAnnotation** **details**="name->:0, kind->simple" …

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).

<xsd:element nillable="true" type="xsd:int" … />	**EAttribute** **unsettable**=true **eType**="…/XMLType#//IntObject" …

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.

<xsd:element nillable="true" type="USState" … />	**EClass** **EAttribute** **unsettable**=true **eType**="//USStateObject" …

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:

name = the name of the original **EDataType** with the string “Object” appended
instanceClassName = a Java wrapper class or “org.eclipse.emf.common.util.AbstractEnumerator”

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

key = "name", **value** = the name of the enumeration type with the suffix ":Object" appended
key = "baseType", **value** = the enumeration type

<xsd:simpleType name="USState"> <xsd:restriction base="xsd:string"> <xsd:enumeration value="AK"/> <xsd:enumeration value="AL"/> <!-- and so on ... --> </xsd:restriction> </xsd:simpleType>	*EEnum name="USState"* … **EDataType** **name**="USStateObject" **instanceClassName**="…util.AbstractEnumerator" EAnnotation **details**="name->USState:Object, baseType->USState" …

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:

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

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

<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>	**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>*	**EAttribute** **EAnnotation** **details**="namespace->##targetNamespace, …" …

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>*	**EAttribute** **EAnnotation** **details**="namespace->http://someSchema", …" …

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>*	*DocumentRoot EClass* **EAttribute** **name**="zip" **eType**="//ZipCode" **upperBound**=-2 *(unspecified)* … **EAnnotation** **details**="namespace->##targetNamespace, …" …

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

<xsd:element name="customerReview" substitutionGroup="criticsReview" type="customerReviewType"/>	**EReference** **name**="customerReivew" … **EAnnotation** **details**="affiliation->criticsReview, …" …

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 **EClass**es 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:

<xsd:element name="criticsReview" type="criticsReviewType"/> … <xsd:complexType name="reviewType"> <xsd:sequence> … <xsd:element ref="criticsReview"/> … </xsd:sequence> </xsd:complexType>	**EClass** **name**="CriticsReviewType" … **EAttribute** **name**="criticsReviewGroup" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->criticsReview:group, kind->group" … **EReference** **name**="criticsReview" **eType**="//CriticsReviewType" **upperBound**=1 **volatile**=true **transient**=true **derived**=true *(derived from "myElementGroup")* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->criticsReview, kind->element, group->criticsReview:group" …

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**.

<xsd:element name="address" abstract="true" type=”addressType” />	**EAttribute** **name**="addressGroup" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* … **EReference** **name**="address" **eType**="//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.

<xsd:element name="…” ecore:name="MyName" … />	**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**.

<xsd:element name="books" type="Book" maxOccurs="unbounded" ecore:opposite="library"/>	**EReference** **name**="books" **eType**="//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:

<xsd:element name="myElement" type="addressType" ecore:featureMap="MyMap" … />	**EAttribute** **name**="myMap" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* … **EReference** **name**="myElement" **eType**="//AddressType" **volatile**=true **transient**=true **derived**=true *(derived from "myMap")* …

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=""/>	**EReference** **name**="myElement" **eType**="//AddressType" …

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>	**EAttribute** **name**="element1" **eType**="…/XMLType#//String" **lowerBound**=0 **upperBound**=1 **EAnnotation** **details**="exclusion->element2 element3, …" …

6.2 Repeating model group

A model group (xsd:sequence, xsd:choice, or xsd:all) with 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"

<xsd:choice maxOccurs="unbounded"> … </xsd:choice>	**EAttribute** **name**="group" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name->group:0, kind->group"

All other **EReference**s and **EAttribute**s which are mapped from element declarations in the schema will have **derived** implementations which delegate to the feature map:

<xsd:choice> <xsd:element name="element1" type="xsd:float"/> … </xsd:sequence>	**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), if necessary.
.

<xsd:group name="aGroup"> <xsd:choice> …	**EClass** **name**="…" **EAttribute**

</xsd:choice> </xsd:group> … <xsd:complexType name="…"> <xsd:group ref="aGroup" maxOccurs="unbounded"/> </xsd:complexType>	**name**="aGroup" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* …

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

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

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.

<xsd:group name="aGroup" ecore:name="myName"> <xsd:choice> … </xsd:choice> </xsd:group> … <xsd:complexType name="…"> <xsd:group ref="aGroup" maxOccurs="unbounded"/> </xsd:complexType>	**EClass** **name**="…" **EAttribute** **name**="myName" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* …

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

<xsd:any namespace="##any" maxOccurs="unbounded"/>	**EAttribute** **name**="any" **eType**="…/Ecore#//EFeatureMapEntry" **lowerBound**=1 **upperBound**=-1 *(unbounded)* **EAnnotation** **source**="…/ExtendedMetaData" **details**="name->:0, kind->elementWildcard, wildcards->##any, processing->strict"
<xsd:anyAttribute namespace="##other"/>	**EAttribute** **name**="anyAttribute" **eType**="…/Ecore#//EFeatureMapEntry" **lowerBound**=0 **upperBound**=-1 *(unbounded)*

	EAnnotation **source**="…/ExtendedMetaData" **details**="name->:1, kind->attributeWildcard, wildcards->##other, processing->strict"

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"/>	**EAttribute** **name**="extension" **eType**="…/Ecore#//EFeatureMapEntry" …

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"/>	**EAttribute** **name**="any" … **EAnnotation** **source**="…/ExtendedMetaData" **details**="processing->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 **EAnnotation**s 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

<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 "documentataion" entry in the **details** map will simply contain the concatenation of the individual documentation elements:

<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".

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

<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:

<xsd:annotation> <documentation ecore:ignore="true" …> doc1 </documentation> …	*No documentation entry for "doc1" in **details** map*

</xsd:annotation>	

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**.

XML Schema Simple Type	EDataType (in XMLTypePackage)	
	name	instanceClass
anySimpleType	AnySimpleType	**java.lang.Object**
anyURI	AnyURI	**java.lang.String**
base64Binary	Base64Binary	**byte[]**
boolean	Boolean	**java.lang.boolean**
boolean (nillable="true")	BooleanObject	**java.lang.Boolean**
byte	Byte	**byte**
byte (nillable="true")	ByteObject	**java.lang.Byte**
date	Date	**java.lang.Object**
dateTime	DateTime	**java.lang.Object**
decimal	Decimal	**java.math.BigDecimal**
double	Double	**double**
double (nillable="true")	DoubleObject	**java.lang.Double**
duration	Duration	**java.lang.Object**
ENTITIES	ENTITIES	**java.util.List**
ENTITY	ENTITY	**java.lang.String**
float	Float	**float**
float (nillable="true")	FloatObject	**java.lang.Float**
gDay	GDay	**java.lang.Object**
gMonth	GMonth	**java.lang.Object**
gMonthDay	GMonthDay	**java.lang.Object**
gYear	GYear	**java.lang.Object**
gYearMonth	GYearMonth	**java.lang.Object**
hexBinary	HexBinary	**byte[]**
ID	ID	**java.lang.String**
IDREF	IDREF	**java.lang.String**
IDREFS	IDREFS	**java.util.List**
int	Int	**int**
int (nillable="true")	IntObject	**java.lang.Integer**
integer	Integer	**java.math.BigInteger**
language	Language	**java.lang.String**
long	Long	**long**
long (nillable="true")	LongObject	**java.lang.Long**
Name	Name	**java.lang.String**
NCName	NCName	**java.lang.String**
negativeInteger	NegativeInteger	**java.math.BigInteger**
NMTOKEN	NMToken	**java.lang.String**
NMTOKENS	NMTOKENS	**java.util.List**
nonNegativeInteger	NonNegativeInteger	**java.math.BigInteger**
nonPositiveInteger	NonPositiveInteger	**java.math.BigInteger**

normalizedString	NormalizedString	**java.lang.String**
NOTATION	NOTATION	**java.lang.Object**
positiveInteger	PositiveInteger	**java.math.BigInteger**
QName	QName	**java.lang.Object**
short	Short	**short**
short (nillable="true")	ShortObject	**java.lang.Short**
string	String	**java.lang.String**
time	Time	**java.lang.Object**
token	Token	**java.lang.String**
unsignedByte	UnsignedByte	**short**
unsignedByte (nillable="true")	UnsignedByteObject	**java.lang.Short**
unsignedInt	UnsignedInt	**long**
unsignedInt (nillable="true")	UnsignedIntObject	**java.lang.Long**
unsignedLong	UnsignedLong	**java.math.BigInteger**
unsignedShort	UnsignedShort	**int**
unsignedShort (nillable="true")	**UnsignedShortObject**	**java.lang.Integer**