Date: June 2013

OMG MOF 2 XMI Mapping Specification

Version 2.4.1
with change bars

OMG Document Number: formal/2013-06-04
Standard document URL: http://www.omg.org/spec/XMI/2.4.1
Associated Normative Machine-Readable Files*:
http://www.omg.org/spec/XMI/20110701/XMI.xsd
http://www.omg.org/spec/XMI/20110701/XMI-model.mdxml
http://www.omg.org/spec/XMI/20110701/XMI-model.xmi

---

* original file: ptc/2011-09-12

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- CORBA/IIOP
- Data Distribution Services
- Specialized CORBA

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- UML, MOF, CWM, XMI
- UML Profile

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- CORBAServices
- CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/report_issue.htm*.

1 Scope

XMI is a widely used interchange format for sharing models using XML. XMI defines many of the important aspects involved in describing objects in XML:

- The representation of objects in terms of XML elements and attributes is the foundation.
- Since objects are typically interconnected, XMI includes standard mechanisms to link objects within the same file or across files.
- Object identity allows objects to be referenced from other objects in terms of IDs and UUIDs.
- Validation of XMI documents using XML Schemas.

XMI describes solutions to the above issues by specifying EBNF production rules to create XML documents and Schemas that share objects consistently.

MOF is the foundation technology for describing metamodels, which cover the wide range of domains: this in turn is based on (a constrained subset) of UML.

2 Conformance

2.1 Introduction

This sub clause describes the required and optional points of compliance with the XMI specification. “XMI Document” and “XMI Schema” are defined as documents and schemas produced by the XMI production (document and XML schema) rules defined in this specification.

2.2 Required Compliance

2.2.1 XMI Schema Compliance

XMI Schemas must be equivalent to those generated by the XMI Schema production rules specified in this document. Equivalence means that XMI documents that are valid under a schema produced by the XMI Schema production rules would be valid in a conforming XMI Schema and that those XMI documents that are not valid under a schema produced by the XMI Schema production rules are not valid in a conforming XMI Schema.

2.2.2 XMI Document Compliance

XMI Documents are required to conform to the following points:

- The XMI document must be “valid” and “well formed” as defined by the XML recommendation, whether used with or without the document’s corresponding XMI Schema(s). Although it is optional not to transmit and/or validate a document with its XMI Schema(s), the document must still conform as if the check had been made.

- The XMI document must be equivalent to those generated by the XMI Document production rules specified in this document. Equivalence for two documents requires a one to one correspondence between the elements in each document, each correspondence identical in terms of element name, element attributes (name and value), and contained elements. Elements declared within the XMI documentation and extension elements are excepted.

2.2.3 Software Compliance

Software is XMI schema compliant when it produces XML schemas that are XMI schema compliant. Software is XMI document compliant when it produces or consumes XML documents that are XMI document compliant.

2.3 Optional Compliance Points

2.3.1 XMI Extension and Differences Compliance

XMI Documents optionally conform to the following points:

- The guidelines for using the extension elements suggested in “XMI Model” on page 8 are found there and in “Tailoring Schema Production” on page 26. Tools should place their extended information within elements that are not in the XMI namespace or within elements that have the XMI namespace and a tag name of “Extension.” They should also declare the nature of the extension using the standard XMI elements where applicable, and preserve the extensions of other tools that fall within the XMI namespace.
- Processing of XMI differencing elements (“Effects on Document Production” on page 31) is an optional compliance point.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- Meta Object facility (MOF) Core Specification, Version 2.4.1
  http://www.omg.org/spec/MOF
- Extensible Markup Language (XML) 1.0 (Fifth Edition)
  W3C Recommendation 26 November 2008
- XML Schema Part 1: Structures Second Edition
  W3C Recommendation 28 October 2004
- XML Schema Part 2: Datatypes Second Edition
  W3C Recommendation 28 October 2004
- XML Linking Language (XLink) Version 1.1
  W3C Recommendation 06 May 2010
- XPointer Framework
  W3C Recommendation 25 March 2003

- XPointer element() Scheme
  W3C Recommendation 25 March 2003
- XPointer xmlns() Scheme
  W3C Recommendation 25 March 2003

4 Terms and Definitions

There are no formal definitions in this specification that are taken from other documents.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Adaptive
- Ceira Technologies, Inc.
- Compuware Corporation
- DSTC
- Hewlett-Packard
- International Business Machines
- IONA
- MetaMatrix
- Softeam
- Sun Microsystems
- Telelogic, AB
- Unisys
- University of Kent

7 XMI Document and Schema Design Principles

7.1 Purpose

This clause contains a description of the XML documents produced from instances of MOF models, and XML schemas that may be used to allow some XML validation of these documents. The use of schemas in XMI is described first, followed by a brief description of some basic principles, which includes a short description of each XML attribute and XML element defined by XMI. Those descriptions are followed by more complete descriptions that provide examples illustrating the motivation for the XMI schema design in the areas of model class specification, transmitting incomplete metadata, linking, tailoring schema production, transmitting metadata differences, and exchanging documents between tools.

It is possible to define how to automatically generate a schema from the MOF model to represent any MOF-compliant model. That definition is presented in Clause 7.

You may specify tag value pairs as part of the MOF model to tailor the schemas that are generated, but you are not required to do so. Using these tag value pairs requires some knowledge of XML schemas, but the schemas that are produced might perform more validation than the default schemas. Section 7.11, “Tailoring Schema Production,” on page 26 describes the tag values, their affect on schema production, and their impact on document serialization.

7.2 Use of XML Schemas

An XML schema provides a means by which an XML processor can validate the syntax and some of the semantics of an XML document. This specification provides rules by which a schema can be generated for any valid XMI-transmissible MOF-based model. However, the use of schemas is optional; an XML document need not reference a schema, even if one exists. The resulting document can be processed more quickly, at the cost of some loss of confidence in the quality of the document.

Although XML schemas are optional in general terms, it is incumbent on standards bodies that define MOF2 instances to produce corresponding XMI 2 Schemas for them.

It can be advantageous to perform XML validation on the XML document containing MOF model data. If XML validation is performed, any XML processor can perform some verification, relieving import/export programs of the burden of performing these checks. It is expected that the software program that performs verification will not be able to rely solely on XML validation for all of the verification since XML validation does not perform all of the verification that could be done.

Each XML document that contains model data conforming to this specification contains: XML elements that are required by this specification, XML elements that contain data that conform to a model, and, optionally, XML elements that contain metadata that represent extensions of the model. Models are explicitly identified in XML elements required by this specification. Some model information can also be encoded in an XML schema. Performing XML validation provides useful checking of the XML elements that contain metadata about the information transferred, the transfer information itself, and any extensions to the model.

The XML Namespace specification has been adopted by the W3C, allowing XMI to use multiple models at the same time. XML schema validation works with XML namespaces, so you can choose your own namespace prefixes in an XML document and use a schema to validate it. The namespace URIs, not the namespace prefixes, are used to identify which schemas to use to validate an XML document.

7.2.1 XML Validation of XMI documents

XML validation can determine whether the XML elements required by this specification are present in the XML document containing model data, whether XML attributes that are required in these XML elements have values for them, and whether some of the values are correct.

XML validation can also perform some verification that the model data conforms to a model. Although some checking can be done, it is impossible to rely solely on XML validation to verify that the information transferred satisfies all of a model's semantic constraints. Complete verification cannot be done through XML validation because it is not currently possible to specify all of the semantic constraints for a model in an XML schema, and the rules for automatic generation of a schema preclude the use of semantic constraints that could be encoded in a schema manually, but cannot be automatically encoded.

Finally, XML validation can be used to validate extensions to the model, because extensions must be represented as elements; if those elements are defined in a schema, the schema can be used to verify the elements.

7.2.2 Requirements for XMI Schemas

Each schema used by XMI must satisfy the following requirements:

- All XML elements and attributes defined by the XMI specification must be imported in the schema. They cannot be put directly in the schema itself, since there is only one target namespace per schema.
- Model constructs have corresponding element declarations, and may have an XML attribute declaration, as described below. In addition, some constructs also have a complexType declaration. The declarations may utilize groups, attribute groups, and types, as described below.
- Any XML elements that represent extensions to the model may be declared in a schema, although it is not necessary to do so.

By default, XMI schemas allow incomplete metadata to be transmitted, but you can enforce the lower bound of multiplicities if you wish (see 7.9 for further details).

7.3 Basic Principles

This sub clause discusses the basic organization of an XML schema for XMI. Detailed information about each of these topics is included later in this clause.

7.3.1 Required XML Declarations

This specification requires that XML element declarations, types, attributes, and attribute groups be included in schemas to enable XML validation of metadata that conforms to this specification.

All XML elements defined by this specification are in the namespace "http://www.omg.org/spec/XMI/*version-namespace*," where *version-namespace* is the version of the XMI specification being used. The XML namespace mechanism can be used to avoid name conflicts between the XMI elements and the XML elements from your MOF models.

In addition to required XML element declarations, there are some attributes that must be defined according to this specification. Every XML element that corresponds to a model class must have XML attributes that enable the XML element to act as a proxy for a local or remote XML element. These attributes are used to associate an XML element with another XML element. There are also other required attributes to let you put data in XML attributes rather than XML elements. You may customize the declarations using MOF tag values.

7.3.2 Model Class Representation

Every model class is represented in the schema by an XML element whose name is the class name, as well as a complexType whose name is the class name. The declaration of the type lists the properties of the class. By default, the content models of XML elements corresponding to model classes do not impose an order on the properties.

By default, XMI allows you to serialize features using either XML elements or XML attributes; however, XMI allows you to specify how to serialize them if you wish. Composite and multivalued properties are always serialized using XML elements.

7.3.3 Model Extension Mechanism

Every XMI schema contains a mechanism for extending a model class. Zero or more **extension** elements are included in the content model of each class. These extension elements have a content model of ANY, allowing considerable freedom in the nature of the extensions. The processContents attribute is **lax**, which means that processors will validate the elements in the extension if a schema is available for them, but will not report an error if there is no schema for them. In addition, the top level XMI element may contain zero or more **extension** elements, which provides for the inclusion of any new information. One use of the extension mechanism might be to transmit data that represents extensions to a model.

Tools that rely on XMI are expected to store the extension information and export it again to enable round trip engineering, even though it is unlikely they will be able to process it further. XML elements that are put in the **extension** elements may be declared in schemas, but are not required to be.

7.4 XMI Schema and Document Structure

Every XMI schema consists of the following declarations:

- An XML version processing instruction. Example: <?XML version=”1.0”?>
- An optional encoding declaration that specifies the character set, which follows the ISO-10646 (also called extended Unicode) standard. Example: <?XML version=”1.0” ENCODING=”UCS-2”?>
- Any other valid XML processing instructions.
- A schema XML element.
- An import XML element for the XMI namespace.
- Declarations for a specific model.

Every XMI document consists of the following declarations, unless the XMI is embedded in another XML document:

- An XML version processing instruction.
- An optional encoding declaration that specifies the character set.
- Any other valid XML processing instructions.

XMI imposes no ordering requirements beyond those defined by XML. XML Namespaces may also be declared in the XMI element as described below.

The top element of the XMI information structure is either the XMI element, or an XML element corresponding to an instance of a class in the MOF model. An XML document containing only XMI information will have XMI as the root element of the document. It is possible for future XML exchange formats to be developed that extend XMI and embed XMI elements within their XML elements.

7.5 XMI Model

This sub clause describes the model for XMI document structure, called the XMI model. The XMI model is an instance of MOF for describing the XMI-specific information in an XMI document, such as the version, documentation, extensions, and differences.

Using an XMI model enables XMI document metadata to be treated in the same fashion as other MOF metadata, allowing use of standard MOF APIs for access to and construction of XMI-specific information in the same manner as other MOF objects. A valid XMI document may contain XMI metadata but is not required to.

7.5.1 XML Schema for the XMI Model

When the XMI model is generated as an XML Schema following the XMI schema production rules, the result is a set of XML element and attribute declarations. These declarations are shown in Clause 7 and given the XML namespace name of the form “http://www.omg.org/spec/XMI/*version-namespace*,” where *version-namespace* is the XML namespace for the version of the XMI specification being used. Every XMI-compliant schema must include the declarations of the following XML elements by importing the declarations in the XMI namespace “http://www.omg.org/spec/XMI/*version-namespace*.” The version of this XMI specification is 2.4.1, and its XMI namespace is “http://www.omg.org/spec/XMI/20110701,” and the XSD file can be found at “http://www.omg.org/spec/XMI/20110701/XMI.xsd.”

In addition, there are attribute declarations and attributeGroup declarations that must be imported. These include the id attribute, and the IdentityAttribs, LinkAttribs, and ObjectAttribs attribute groups. These constructs are not defined in the XMI model.

In the declarations that follow, the XML Schema namespace, whose URI is “http://www.w3.org/2001/XMLSchema,” has the namespace prefix “xsd.” The XMI namespace is the default namespace.

7.5.2 XMI Model classes

There are three diagrams that describe the XMI model. The details of the classes are described in the sections below. This sub clause gives an overview of the model.

Figure 7.1 shows the XMI element, documentation, and extension elements. The XMI class is an overall default container for XMI document metadata and contents. The attributes of the XMI class are the documentation, differences (add, replace, delete in Figure 7.2), and extensions. The Documentation class contains many fields to describe the document for non-computational purposes. The Extension class contains the metadata for external information. The String datatype and the Integer datatype come from the PrimitiveTypes package used by MOF Core and UML Infastructure. The PrimitiveTypes package also contains UnlimitedNatural and Boolean. The DateTime primitive type has XML Schema data type of “http://www.w3.org/2001/XMLSchema#dateTime.”

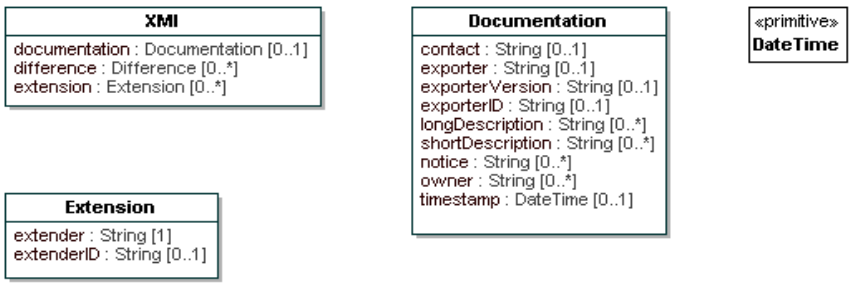


Figure 7.1 - The XMI Model for the XMI element, documentation, and extension

The differences information (Figure 7.2) is described as additions, deletions, and replacements to target objects. The objects referenced by the differences may be in the same or different documents. The differences information consists of the Add, Delete, and Replace classes, which specify a set of differences and refer to MOF objects that are added or removed. Note that the Element class is a placeholder for specifying that a Difference has a target that can refer to any objects. The Element class is not included in the required element declarations.

The XML Schema declarations for each element of the XML model are given in the following sub clauses. They may be generated by following the XMI production of XML Schema rules defined in Clause 7, except for the XMI class and the XMI attributes described in “XMI Attributes” on page 13.

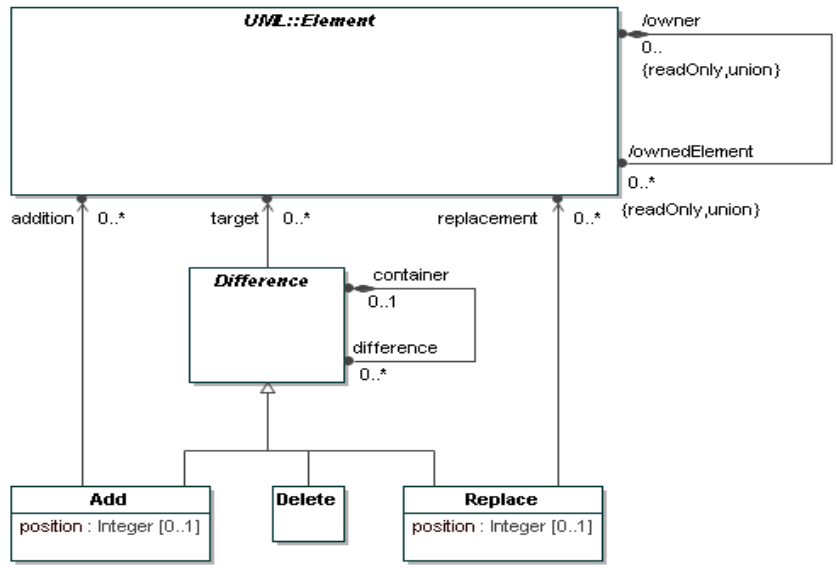


Figure 7.2 - The XMI Model for differences

7.5.3 XMI

The root level XML element for XMI documents containing only XMI data may be the XMI element, but it must be the XMI element if there are multiple elements. Its declaration is:

```
<xsd:complexType name="XMI">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:any processContents="strict"/>
  </xsd:choice>
  <xsd:attribute ref="id"/>
  <xsd:attributeGroup ref="IdentityAttribs"/>
  <xsd:attributeGroup ref="LinkAttribs"/>
  <xsd:attribute name="type" type="xsd:QName" use="optional"
                 form="qualified"/>
</xsd:complexType>
<xsd:element name="documentation" type="Documentation"/>
<xsd:element name="difference" type="Difference"/>
<xsd:element name="extension" type="Extension"/>
<xsd:element name="XMI" type="XMI"/>
```

Note that in the schema that the elements for documentation, difference and extension may not validly be included in the xsd:choice for XMI since that already has xsd:any. However are the elements that must be used within the XMI elements. The Documentation, Difference, and Extension elements (starting with uppercase), defined in the following sub clauses, may only be used if they are root elements, not nested underneath XMI, and qualified with the XMI namespace: for example xmi:Documentation.

Each version of XMI is unambiguously identified by its unique namespace URI of the form “http://www.omg.org/spec/XMI/version-namespace.”

The XMI element need not be the root element of an XML document; you can include it inside any XML element that was not serialized according to this specification. If a document contains only XMI information, the XMI element may not be present when there is only a single top-level object, but is often useful for consistency and for elements such as Documentation. The start of XMI information and identification of the XMI version is indicated by the presence of the XMI namespace declaration, regardless of whether the XMI element itself is present. Clause 8 contains examples of the use of the XMI element.

The XMI class has the XMI tag org.omg.xmi.contentType set to “any” to indicate that any XMI element may be present in the XMI stream.

See “Overall Document Structure” on page 57” for details on how the XMI class is serialized.

The XMI model package has the following tag settings:

- tag org.omg.xmi.nsURI set to “http://www.omg.org/spec/XMI/*version-namespace*”
- tag org.omg.xmi.nsPrefix set to “xmi”
- tag org.omg.xmi.superClassFirst set to “true”
- tag org.omg.xmi.useSchemaExtension set to “true”
- tag org.omg.xmi.element set to “true”
- tag org.omg.xmi.attribute set to “false”

7.5.4 Extension

The Extension class is designed to contain extended information outside the scope of the user model. Extensions are a multivalued attribute of the XMI class and may also be embedded in specific locations in an XMI document. The Schema for extension is:

```
<xsd:complexType name="Extension">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:any processContents="lax"/>
  </xsd:choice>
  <xsd:attribute ref="id"/>
  <xsd:attributeGroup ref="ObjectAttribs"/>
  <xsd:attribute name="extender" type="xsd:string" use="optional"/>
  <xsd:attribute name="extenderID" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Extension" type="Extension"/>
```

The **extender** attribute should indicate which tool made the extension. It is provided so that tools may ignore the extensions made by other tools before the content of the extensions element is processed. The **extenderID** is an optional internal ID from the extending tool that allows the element to be uniquely located within the tool. The other attributes allow individual extensions to be identified and to act as proxies for local or remote extensions.

The Extension class in the MOF model has the tag org.omg.xmi.contentType set to “any” and the org.omg.xmi.processContents tag set to “lax.” The extender and extenderID attributes have the tag attribute set to “true.”

7.5.5 Documentation

The Documentation class contains information about the XMI document or stream being transmitted, for instance the owner of the document, a contact person for the document, long and short descriptions of the document, the exporter tool which created the document, the version of the tool, the date and time the document was created, and copyright or other legal notices regarding the document. The data type of all the attributes of Documentation is string except for the timestamp which is DateTime. The XML Schema generated for Documentation is:

```
<xsd:complexType name="Documentation">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="contact" type="xsd:string"/>
    <xsd:element name="exporter" type="xsd:string"/>
    <xsd:element name="exporterVersion" type="xsd:string"/>
    <xsd:element name="longDescription" type="xsd:string"/>
    <xsd:element name="shortDescription" type="xsd:string"/>
    <xsd:element name="notice" type="xsd:string"/>
    <xsd:element name="owner" type="xsd:string"/>
    <xsd:element name=timestamp” type=”xsd:datetime”/>
    <xsd:element ref="Extension"/>
  </xsd:choice>
  <xsd:attribute ref="id"/>
  <xsd:attributeGroup ref="ObjectAttribs"/>
  <xsd:attribute name="contact" type="xsd:string" use="optional"/>
  <xsd:attribute name="exporter" type="xsd:string" use="optional"/>
  <xsd:attribute name="exporterVersion" type="xsd:string" use="optional"/>
```

```
  <xsd:attribute name="longDescription" type="xsd:string" use="optional"/>
  <xsd:attribute name="shortDescription" type="xsd:string" use="optional"/>
  <xsd:attribute name="notice" type="xsd:string" use="optional"/>
  <xsd:attribute name="owner" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Documentation" type="Documentation"/>
```

7.5.6 Add, Replace, and Delete

The Add class represents an addition to a target object in this document or other documents. The *target* is constrained to reference only one object. The *position* attribute indicates where to place the addition relative to other XML elements of that type within the target. The default, -1, indicates to add the new elements at the end of those elements for the target element. The *addition* attribute refers to the set of objects to be added. Both of these attributes have the tag org.omg.xmi.attribute set to “true.”

The Replace class represents the removal of a target set of objects and the addition of the objects referred to in the *replacement* attribute. The *position* attribute indicates where to place the replacements relative to other XML elements of that type within their container (they should all be of the same XML type). The default, -1, indicates to add the new elements at the end of those elements for the target element. The *replacement* attribute refers to the objects that will replace the target elements. Both of these attributes have the tag org.omg.xmi.attribute set to “true.” Note that, unlike Delete, the replaced elements are only removed from the container not deleted.

The Delete class represents a deletion of the target set of objects in this document or other documents.

The Difference class is the superclass for the Add, Replace, and Delete classes (see Figure 7.2 and sub clause 7.12).

The declarations for these classes are:

```
<xsd:complexType name="Difference">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="target">
      <xsd:complexType>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:any processContents="skip"/>
        </xsd:choice>
        <xsd:anyAttribute processContents="skip"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="difference" type="Difference"/>
    <xsd:element name="container" type="Difference"/>
    <xsd:element ref="Extension"/>
  </xsd:choice>
  <xsd:attribute ref="id"/>
  <xsd:attributeGroup ref="ObjectAttribs"/>
  <xsd:attribute name="target" type="xsd:IDREFS" use="optional"/>
  <xsd:attribute name="container" type="xsd:IDREFS" use="optional"/>
</xsd:complexType>

<xsd:element name="Difference" type="Difference"/>
```

```
<xsd:complexType name="Add">
 <xsd:complexContent>
  <xsd:extension base="Difference">
   <xsd:attribute name="position" type="xsd:integer" use="optional"/>
   <xsd:attribute name="addition" type="xsd:IDREFS" use="optional"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Add" type="Add"/>

<xsd:complexType name="Replace">
 <xsd:complexContent>
  <xsd:extension base="Difference">
   <xsd:attribute name="position" type="xsd:string" use="optional"/>
   <xsd:attribute name="replacement" type="xsd:IDREFS" use="optional"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Replace" type="Replace"/>

<xsd:complexType name="Delete">
 <xsd:complexContent>
  <xsd:extension base="Difference"/>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Delete" type="Delete"/>
```

7.6 XMI Attributes

This sub clause describes the fixed XML attributes that are used in the XMI production of XML documents and Schemas. By defining a consistent set of XML attributes, XMI provides a consistent architectural structure enabling consistent object identity and linking across all assets.

7.6.1 Element Identification Attributes

Three XML attributes are defined by this specification to identify XML elements so that XML elements can be associated with each other. The purpose of these attributes is to allow XML elements to reference other XML elements using XML IDREFs, XLinks, and XPointers.

Two of these attributes are declared in an attribute group called **IdentityAttribs**; the id attribute is declared globally. Placing these attributes in an attribute group prevents errors in the declarations of these attributes in schemas. Its declaration is as follows:

```
<xsd:attribute name="id"  type="xsd:ID" use="optional"/>

<xsd:attributeGroup name="IdentityAttribs">
  <xsd:attribute name="label" type="xsd:string" use="optional"
```

```
                    form="qualified"/>
  <xsd:attribute name="uuid" type="xsd:string" use="optional"
                    form="qualified"/>
</xsd:attributeGroup>
```

id

XML semantics require the values of this attribute to be unique within an XML document; however, the value is not required to be globally unique. This attribute may be used as the value of the **idref** attribute defined in the next sub clause. It may also be included as part of the value of the **href** attribute in XLinks. An example of the use of this attribute and the other attributes in this sub clause can be found in Section 7.10.3, “Example for UML,” on page 25.

If the metaclass has (or inherits) a Property with isId = ‘true,’ then the value of that property may be used as the basis of the xmi:id and/or xmi:uuid attributes.

This is not mandatory, and the exact algorithm to be used is not specified in this specification. However it is important, to be a valid XML document, that the value for xmi:id is unique across all elements within the file. The xmi:uuid is not so constrained, but if the same value is used in multiple XML elements, then they are all deemed to reference the same MOF element (e.g., they may represent different aspects).

label

This attribute may be used to provide a string label identifying a particular XML element. Users may put any value in this attribute.

The value of the label attribute is ignored on import.

uuid

The purpose of this attribute is to provide a globally unique identifier for an XML element. The values of this attribute should be globally unique strings prefixed by the type of identifier. If you have access to the UUID assigned in MOF, you may put the MOF UUID in the uuid XML attribute when encoding the MOF data in XMI.

UUIDs should use URIs as the unique string. Refer to sub clause 6.4.1.1 of the MOF Facility and Object Lifecycle Specification for an example of a scheme for detailed URI production rules.

An example URI for the metaclass UseCase in the UML2 metamodel looks like this:

http://www.omg.org/spec/UML//20200901/uml.xml#UseCase

7.6.2 Linking Attributes

XMI allows the use of several XML attributes to enable XML elements to refer to other XML elements using the values of the attributes defined in the previous sub clause. The purpose of these attributes is to allow XML elements to act as simple XLinks or to hold a reference to an XML element in the same document using the XML IDREF mechanism.

The attributes described in this sub clause are included in an attribute group called **LinkAttribs**. The attribute group declaration is:

```
<xsd:attributeGroup name="LinkAttribs">
  <xsd:attribute name="href" type="xsd:anyURI" use="optional"/>
  <xsd:attribute name="idref" type="xsd:IDREF" use="optional"
                    form="qualified"/>
</xsd:attributeGroup>
```

The link attributes act as a union of two linking mechanisms, any one of which may be used at one time. The mechanisms are the XLink **href** for advanced linking across or within a document, or the **idref** for linking within a document.

XMI offers another mechanism for linking, using the name of the property involved in the reference instead of href or idref. See sub clause 7.10 for more information.

Simple XLink Attributes

The href attribute declared in the above entity enables an XML element to act in a fashion compatible with the simple XLink according to the XLink and XPointer W3C recommendations. The declaration and use of href is defined in the XLink and XPointer specifications. XMI enables the use of simple XLinks. XMI does not preclude the use of extended XLinks, although it is not anticipated that many XMI tools will support them. The XLink specification defines many additional XML attributes, and it is permissible to use them in addition to the attributes defined in the LinkAttribs group.

To use simple XLinks, set **href** to the URI of the desired location. The **href** attribute can be used to reference XML elements whose **id** attributes are set to particular values. The **id** attribute value can be specified using a special URI form for XPointers defined in the XLink and XPointer recommendations.

idref

This attribute allows an XML element to refer to another XML element within the same document using the XML IDREF mechanism. In XMI documents, the value of this attribute should be the value of the **id** attribute of the XML element being referenced.

7.6.3 Type Attribute

The type attribute is used to specify the type of object being serialized, when the type is not known from the model. This can occur if the type of a reference has subclasses, for instance. The declaration of the attribute is:

```
<xsd:attribute name="type" type="xsd:QName" form="qualified"/>
```

Rather than including the IdentityAttribs, and LinkAttribs attribute groups, and the version and type attributes in the declarations for each MOF class, the XMI namespace includes the following declaration of the **ObjectAttribs** attribute group for the attribute declarations that pertain to objects:

```
<xsd:attributeGroup name="ObjectAttribs">
  <xsd:attributeGroup ref="IdentityAttribs"/>
  <xsd:attributeGroup ref="LinkAttribs"/>
  <xsd:attribute name="type" type="xsd:QName"
                 form="qualified"/>
</xsd:attributeGroup>
```

7.7 XMI Types

The XMI namespace contains a type called Any. It is used in the XMI schema production rules for class attributes, class references, and class compositions. The declaration of this type is part of the fixed declarations for XMI. The Any type allows any content and any attributes to appear in elements of that type, skipping XML validation for the element's content and attributes. The declaration of the type is as follows:

```
<xsd:complexType name="Any">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:any processContents="skip"/>
```

```
    </xsd:choice>
    <xsd:anyAttribute processContents="skip"/>
</xsd:complexType>
```

By using this type, the XMI schema production rules generate smaller schemas than if this type was declared multiple times in a schema. Also, using the Any type enables some changes to be made to the Any type declaration without affecting generated XMI schemas.

7.8 Model Representation

This sub clause describes how to represent information using XMI:

- How classes, properties, composites, multiple elements, datatypes, and inheritance are represented in XMI compliant XML schemas.
- How instances of classes are represented in XMI compliant XML documents.

The production rules for these representations are given in EBNF form in the “XML Schema Production” and “XML Document Production” clauses.

7.8.1 Namespace Qualified XML Element Names

When the official schema for a model is produced, the schema generator must use the namespace URI specified by the Package::URI property on the package representing the metamodel, which may be overridden by the org.omg.xmi.nsURI tag to identify uniquely the XML namespace in the model. XML processors will use namespace URIs to identify the schemas to be used for XML validation, as described in the XML schema specification.

The XML element name for each model Class, and Association in a document is its short name. The name for XML tags corresponding to model Properties is the short name of the property. The name of XML attributes corresponding to model properties (DataType-typed or Class-typed) is the short name of the property, since each tag in XML has its own naming context.

Each namespace is assigned a logical URI. The logical URI is placed in the namespace declaration of the top level element in XML documents that contain instances of the model. The XML namespace specification assigns logical names to namespaces that are expected to remain fixed throughout the life of all uses of the namespace since it provides a permanent global name for the resource. An example is “http://www.omg.org/spec/UML/20110701.” There is no requirement or expectation by the XML Namespace specification that the logical URI be resolved or dereferenced during processing of XML documents.

The following is an example of a UML model in an XMI document using namespaces.

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
            xmlns:xmi="http://www.omg.org/spec/XMI/20110701">
  <uml:Class name="C1" xmi:type=”uml:Class” xmi:id=”_1”>
    <ownedAttribute xmi:type="uml:Property" xmi:id=”_2” name="a1"
        visibility="private"/>
  </uml:Class>
</xmi:XMI>
```

The model has a single class named C1 that contains a single attribute named a1 with visibility private. The XMI element declares the version of XMI and the namespace for UML with the logical URI.

7.8.2 Multiplicities

In XMI 1, the multiplicities from the model were ignored, since DTDs were not able to validate multiplicities without ordering the content of XML elements. By default, XMI 2 produces schemas that ignore multiplicities also.

You may tailor the schemas produced by XMI by specifying tag values in the model. Two of the tags, “org.omg.xmi.enforceMaximumMultiplicity” and “org.omg.xmi.enforceMinimumMultiplicity” allow you to specify that multiplicities are to be used in a schema rather than being ignored.

Model multiplicities map directly from the EMOF definition of multiplicity, which is a lower bound and an upper bound, to schema XML attributes called “minOccurs” and “maxOccurs.” The minOccurs XML attribute corresponds to MultiplicityElement’s lower property, and the maxOccurs XML attribute corresponds to its upper property. If the lower bound for a property is null, the org.omg.xmi.enforceMinimumMultiplicity tag is ignored, and minimum multiplicity is not enforced in the Schema (minimum multiplicity is effectively “0”). Similarly, if the upper bound for a property is null, the org.omg.xmi.enforceMaximumMultiplicity tag is ignored, and maximum multiplicity is not enforced in the Schema (the multiplicity is effectively unbounded).

7.8.3 Class Representation

A class is represented by an XML element, with an XML element or attribute for each property. The XML element for the class includes the inherited properties.

In the examples that follow in this sub clause, “xsd” is the namespace prefix for the XML schema namespace (“http://www.w3.org/2001/XMLSchema”) and “xmi” is the namespace prefix for the XMI namespace.

The representation of a class named “c” is shown below for the simplest case where “c” does not have any Properties:

```
<xsd:element name="c" type="c"/>

<xsd:complexType name="c">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>
```

If the class has properties, the XML elements for them are put in the all group of the content model, as explained below.

7.8.4 DataType-typed Property Representation

The representation of properties of class “c” uses XML elements and XML attributes. If the property types are primitives or enumerations, then by default XML attributes and XML elements are declared for these types. The reasons for the XML element choice are several, including: the values to be exchanged may be very large values and unsuitable for XML attributes, and may have poor control of whitespace processing with options that apply only to element contents. The default encoding can be changed using the XMI “org.omg.xmi.attribute” and “org.omg.xmi.element” tags. See sub clause 7.11.3 for information on how these tags affect encoding. See sub clause 7.11.1 for a complete list of XMI tags.

The declaration of a property named “a” is as follows:

```
<xsd:element name="a" type="type specification"/>
```

The XML element corresponding to the property is declared in the content of the complexType corresponding to the class that owns the attribute. The type specification is either an XML schema data type, an enumeration data type, or a class from the model.

For properties whose types are primitive types (for example, String) and whose upper bound multiplicity is 1, an XML attribute must also be declared in the XML element corresponding to model class “c,” and the XML element must be put in the content model of the XML element for class “c.” The declaration of “c” appears as follows without multiplicity enforcement:

```
<xsd:element name="c" type="c"/>

<xsd:complexType name="c">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="a" type="xsd:string" nillable="true"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="a" type="xsd:string" use="optional"/>
  </xsd:complexType>
</xsd:element>
```

An element is also declared to be of XML type string if the class contains a Tag org.omg.xmi.schemaType with value “string.”

For multi-valued DataType-typed Properties, no XML attributes are declared; each value is encoded as an XML element.

When “a” is a property with enumerated values, the type used for the declaration of the XML element and XML attribute corresponding to the model attribute is as follows:

```
<xsd:simpleType base="enumName" >
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="v1"/>
    <xsd:enumeration value="v2"/>
  </xsd:restriction>
</xsd:simpleType>
```

where enumName is the name of the enumeration type, and v1 and v2 are the names of the EnumerationLiterals.

If a property has enumerated values, an XML element and an XML attribute is put in the complexType for the class “c,” their declaration is as follows:

```
<xsd:element name="a" type="enumName"/>

<xsd:attribute name="a" type="enumName" use="optional"/>
```

If a property is a multi-valued enumeration, the declaration of the XML attribute is omitted.

The semantics of default values differs between MOF/UML and XML Schema, so the XML Schema will never contain default values for Properties.

7.8.5 Class-typed Property Representation

A Class-typed property references another model element. Each such reference is represented as an XML element and/or an XML attribute. The XML element declaration for a property named “r” for a class “c” is:

```
<xsd:element name="r" minOccurs="0" maxOccurs=”unbounded”>
    <xsd:attributeGroup ref="LinkAttribs"/>
</xsd:element>
```

This element is declared in the content of the complexType for the class that owns the property. This declaration enables any object to be serialized, enhancing the extensibility of models.

The attribute declaration for the property, which also is included in the complexType declaration for the class that owns the property, is as follows:

```
<xsd:attribute name="r" type="xsd:IDREFS" use="optional"/>
```

7.8.6 Composite Representation

Each property that is a composite is represented by an XML element, but not by an XML attribute.

The XML element declaration for a composite property named “r” for a class “c” of type “ClassType” is:

```
<xsd:element name="r" type="ClassType" minOccurs=”0” maxOccurs=”unbounded”/>
```

This element is declared in the content of the complexType for the class that owns the property.

If the org.omg.xmi.allowMetamodelExtension tag is set to true, then the name of the type is replaced by “xmi:Any”: this declaration enables any object to be serialized, enhancing the extensibility of models.

If org.omg.xmi.useSchemaExtension is false (the default), the names of all non-abstract subtypes must also be included (in alphabetical order of immediate children with depth first expansion): if ClassType has subclasses CTS1 and CTS2, and CTS1 has subclass CTS1S1, then the declaration needs to make use of an anonymous complex type:

```
<xsd:element name="r" minOccurs=”0” maxOccurs=”unbounded”>
      <xsd:complexType>
       <xsd:choice>
          <ref=”ClassType”/>
          <ref=”CTS1”/>
          <ref=”CTS1S1”/>
          <ref=”CTS2”/>
       </xsd:choice>
    </xsd:complexType>
 </xsd:element>
```

7.8.7 Datatype representation

Like classes, datatypes are classifiers and can have instances that are represented by XML elements. Unlike classes, datatypes do not have object identity, so there are no identification attributes in their representation.

The representation of a datatype named “dt” is shown below:

```
<xsd:element name="dt" type="dt"/>
```

```
<xsd:complexType name="dt">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attributeGroup ref="xmi:LinkAttribs"/>
    <xsd:attribute name="version" type="xsd:string" use="optional"
                   form="qualified"/>
    <xsd:attribute name="type" type="xsd:QName" use="optional"
                   form="qualified"/>
</xsd:complexType>
```

In the instance document, the value of a simple datatype appears as an attribute value or as character content.

In CMOF datatypes, other than Primitive and Enumeration Types, can have properties, which in effect allows them to be structured datatypes. During serialization, structured datatypes are treated like classes with properties, reusing the document production rules starting with rule 2a:XMIObjectElement (see sub clause 9.5.2) with the following adaption:

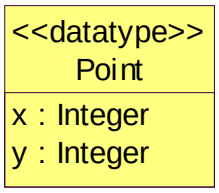
- The name of the structured datatype is used instead of the class name.

Serializing structured datatypes analogous to classes is the default. The org.omg.xmi.valueSeparator tag has no effect on this form of serialization.

Primarily for backward compatibility, flattening of structured datatypes may be performed if all of the following conditions hold:

- The structured datatypes are not nested (i.e., do not contain structured datatypes as one or more fields).
- The fields have multiplicity [1..1].
- The tag org.omg.xmi.flattenStructuredDataTypes (which defaults to false) is set to true.

As an example, here is a datatype called Point with two properties representing the X and Y coordinates of the point:



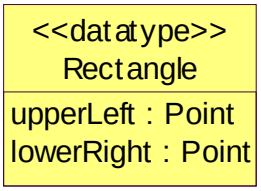
Using the class-like default serialization, an example of a graph with two points would serialize as:

```
<g:Graph xmi:type=”g:Graph”>
    <points xmi:type=”g:Point” x="0" y="0"/>
    <points xmi:type=”g:Point” x="1" y="5"/>
</g:Graph>
```

But using the special case flattened serialization (with org.omg.xmi.flattenStructuredDataTypes=true), the point coordinates would serialize as strings. The separator between the coordinate values is controlled by org.omg.xmi.valueSeparator:

```
<TwoPointsOnAGraph point1=”0,0” point2=”1,5” />
```

A structured datatype may be more than one level deep - its properties can in turn be structured datatypes. For example:



This example shows the nesting of two structured datatypes. The only valid serialization for a property called area of type Rectangle is:

```
<display xmi:type="g:Viewport">
   <area xmi:type="g:Rectangle">
      <upperLeft xmi:type="g:Point" x="0" y="5"/>
      <lowerRight xmi:type="g:Point" x="4" y="0"/>
   </area>
</display>
```

7.8.8 Inheritance representation

XML schemas have a mechanism for extending types, but it does not support extending from more than one type, and using that mechanism imposes an order on the content models of the types that are derived from other types. Since XMI attempts to minimize order dependencies, XMI by default does not use schema extension to represent inheritance. In its place, XMI specifies that inheritance will be copy-down inheritance and therefore uses xmi:type instead of xsi:type.

Multiple inheritance is treated in such a way that the Properties that occur more than once in the inheritance hierarchy are only included once in their subclasses. For associations (Class-typed Properties), the actual class referenced is used, and subclasses may be used on the other end of the reference.

7.8.9 Association Representation

Associations are classifiers whose instances are Links. There are cases where it makes sense to serialize Links: for example where the Association owns all of its ends, to link existing elements, or to add a new element to a composition without replacing the existing contents (e.g., to add a Property to a Class where including a new value for package::packagedElement would lose, or require repeating, the complete current list).

7.8.10 Derived Information

Whether or not information is derived information is orthogonal to whether or not that information is serialized. The org.omg.xmi.serialize tag is provided optionally to include derived data. This capability provides more control to the metamodelers, allowing them to customize exactly which information is present in their files. In some cases, derived information may be more condensed than the information it is derived from. In these cases, serialization of only the derived information may be desirable to keep the size of the XMI file as small as possible.

7.9 Transmitting Incomplete Metadata

Starting with XMI version 2.0, an implementation can decide whether to support the exchange of model fragments.

7.9.1 Interchange of model fragments

In practice, most information is related. The ability to transfer a subset of known information is essential for practical information interchange. In addition, as information models are developed, they will frequently need to be interchanged before they are complete.

The following guidelines apply for interchanging incomplete models via XMI:

- Information may be missing from a model. The transmission format should not require the addition or invention of new information.
- Model fragments may be disjoint sets. Each set may be transmitted in the same XMI file or in different XMI files.
- “Incomplete” indicates a quantity of information less than or equal to “complete.” Additional information beyond that which the model prescribes may be transmitted only via the extension mechanism.
- Semantic verification is performed on the metadata that is actually present as if it was included in complete metadata.

7.9.2 XMI encoding

The interchange of model fragments is accomplished by lowering the lower bound of multiplicities whose lower bound is greater than 0.

7.9.3 Example

The following is an example of an incomplete UML 1.4 model:

```
<UML:Model name="model1" xmi:id="id1">
    <ownedElement xmi:type="UML:Class" name="class1" xmi:id="id2">
            <feature xmi:type="UML:Attribute" name="attribute1"
                    type="type1"/>
    </ownedElement>
     <ownedElement xmi:type="UML:Datatype" name="Integer" xmi:id="type1"/>
</UML:Model>
```

7.10 Linking

The goal is to provide a mechanism for specifying references within and across documents. Although based on the XLinks standard, it is downwards compatible and does not require XLinks as a prerequisite.

7.10.1 Design principles

- Links are based on XLinks to navigate to the document (which may be the current document) and XPointers to navigate to the element within the document.
- Link definitions are encapsulated in the attribute group LinkAttribs defined in sub clause 7.6.2.
- Elements act as a union, where they are either a definition or a proxy. Proxies use the LinkAttribs attribute group to define the link, and contain no nested elements.

- LinkAttribs supports external links through the XLink attributes, and internal links through the xmi:idref and xmi:id attributes.
- Links are always to elements of the same type or subclasses of that type. Restricting proxies to reference the same element type reduces complexity, enhances reliability and type safety, and promotes caching.
- When acting as a proxy, XML attributes may be defined, but not contents. The XML attributes act as a cache or guide that gives an indication if the link should be followed: however there is no guarantee that these cached values accurately represent the current values of the linked element.
- Proxies may be chained.
- When following the link from a proxy, the definition of the proxy is replaced by the referenced element.
- It is efficient practice to use local proxies of the same element within a document to link to a single proxy that holds an external reference. For example: there could be local proxies defined for references to the predefined DataTypes such as Integer, UnlimitedNatural, String, and Boolean.

7.10.2 Linking

For XMI, the most common linking requirements are:

- Linking to an XML element in the same document using the element's id.
- Linking to an XML element in a different document using the element's id.
- Linking to an XML element using the element's uuid or label, in the same or a different document.

The following sections describe how XMI supports these requirements.

7.10.2.1 Linking within a Document

Every construct that can be referred to has a local XML ID, a string that is locally unique within a single XML file. Attributes representing Class-typed properties in the metamodel, or XML idref attributes can refer to other XML elements within the same XML file by specifying the target element's XML ID.

7.10.2.2 Linking across Documents

1. Using the XMI href attribute to locate an XMI id

This is the simplest form of cross document linking. With help from the XMI org.omg.xmi.idName tag, it can be backward compatible with XMI 1.2 and later.

Here, the XMI href attribute is used to locate an XML element in another XML document by its XMI id. The value of href must be a URI reference, as defined by IETF RFC 2396: Uniform Resource Identifiers. The URI reference must be of the form URI#id_value, where URI locates the XML file containing the XML element to link to, and id_value is the value of the XML element's XMI id attribute.

As an example:

```
<mgr xmi:id="mgr_1" href="Co.xml#emp_2"/>
```

locates XML element <**Employee xmi:id="emp_2" …** /> in file Co.xml.

2. Using an XLink simple link and XPointer bare name to locate an XMI id

This is a little more complicated than using the XMI href attribute, and does not provide any more function. It does have the advantage that standard XLink and XPointer software can follow the link.

Here, an xlink:href attribute is used, where XLink is the prefix for the XLink namespace. The XLink prefix must be declared in the document that contains the Xlink:href attribute. For example:

```
<xmi:XMI version="2.1" xmlns:xlink="http://www.w3.org/1999/XLink"
    xmlns:xmi=" http://schema.omg.org/spec/XMI/2.1">
```

The value of xlink:href must again be a URI reference of the form URI#id_value. In this case, id_value is technically an XPointer bare name, but it looks just like the id_value for the XMI href attribute.

The XML element with the xlink:href must also have an xlink:type=”simple” attribute, to identify it as a simple link.

As an example:

```
<mgr xmi:id="mgr_1" xlink:href="Co.xml#emp_2" xlink:type="simple"/>
```

locates XML element <**Employee xmi:id="emp_2" …** /> in file Co.xml.

3. Using an XLink simple link and full XPointer to locate an XMI uuid or label

An XLink simple link and a form of full XPointer can be used to locate an XML element in an XML document by its XMI uuid or label. This describes the form for uuid; the form for label is strictly analogous. Again:

- An xlink:href attribute is used, where XLink is the prefix for the XLink namespace. The xlink prefix must be declared in the document containing the xlink:href attribute.
- The value of xlink:href must be a URI reference.

However this time, the URI reference has a more complicated form:

URI#xpointer((//*[@xmi:uuid='value'])[1])

The xpointer expression is a series of instructions for finding the first element in the target file whose xmi:uuid has that value.

As an example:

```
<mgr xmi:id="mgr_1"
    xlink:href="Co.xml#xpointer((//*[@xmi:uuid='emp_2'])[1])"
    xlink:type="simple"/>
```

locates XML element <**Employee xmi:uuid="emp_2"…**/> in file Co.xml, as long as it is the first element with that uuid in the file.

Since a URI can identify the same file that contains the href, this also supports locating XML elements by XMI uuid in the same document.

4. Using full XLink and XPointer to locate almost anything

XLink and XPointer provide rich and complex capabilities for locating XML elements, far beyond what XMI requires. Consequently it is not expected that XMI implementations supporting linking across documents provide this level of support. The W3C XLink and XPointer specifications define what is possible and how it works.

5. Using the MOF2 facility Basic Encoding Scheme

The MOF2 Facility specification provides a means of encoding URIs to refer to elements in facilities: these may be realized through XMI files, database-backed repositories or other mechanisms. Hence it is usually not appropriate to make use of xmi:id values that are in general transient and limited in scope: rather names and unique ids are made use of. Full details are contained in that specification.

As an example here is a link to an activity called CalculateHoursWorked which is within ProcessModel within PayrollModels; PayrollModels is located via facility http://mof.adaptive.com:8083/ModelsFacility.

```
<activity
href="http://mof.adaptive.com:8083/ModelsFacility/PayrollModels?ProcessModel/MonthlyProcess/
CalculateHoursWorked"/>
```

7.10.3 Example for UML

There is an association between ModelElements and Constraints in UML. Operation is a subclass of Element. This example shows an association between Operations and four Constraints with roles ownedRule and constrainedElement. Each of the methods of linking is shown. The Constraints are shown in both definition and proxy form. Note that one of the constrainedElement elements contains *href="#xpointer(descendent(1,Operation,xmi:label,op1))*." This is an example of case 4. Using full XLink and XPointer to locate almost anything.

Document 1, doc1.xml (omitting root and namespace declarations)::

```
<uml:Operation xmi:id="idO1" xmi:type="uml:Operation" xmi:label="op1"
      xmi:uuid="DCE:1234">
  <ownedRule xmi:id="idC1" xmi:type="uml:Constraint" xmi:label="co1"
      xmi:uuid="DCE:abcd">
    <specification xmi:type="uml:OpaqueExpression">
      <body>First Constraint definition</body>
    </specification>
    <constrainedElement xmi:idref="idO1"/>
  </ownedRule>
  <ownedRule xmi:idref="idC2" />
  <ownedRule xmi:idref="idC3" />
  <ownedRule href="doc2.xml#idC4" />
</uml:Operation>
<uml:Constraint xmi:id="idC2" xmi:type="uml:Constraint" xmi:label="co2"
    xmi:uuid="DCE:efgh">
  <specification xmi:type="uml:OpaqueExpression">
    <body>Second Constraint definition</body>
  </specification>
  <constrainedElement xmi:idref="idO1" />
</uml:Constraint>
<uml:Constraint xmi:id="idC3" xmi:type="uml:Constraint" xmi:label="co3"
    xmi:uuid="DCE:ijkl">
  <specification xmi:type="uml:OpaqueExpression">
     body>Third Constraint definition</body>
  </specification>
  <constrainedElement href="#xpointer(descendent(1,Operation,xmi:label,op1))"/>
</uml:Constraint>
```

Document 2, doc2.xml (omitting root and namespace declarations):

```
<uml:Constraint xmi:id="idC4" xmi:type="uml:Constraint" xmi:label="co4"
  xmi:uuid="DCE:mnop">
 <specification xmi:type="uml:OpaqueExpression">
  <body>Fourth Constraint definition</body>
 </specification>
 <constrainedElement href="doc1.xml#idO1"/>
</uml:Constraint>
```

The first constraint is a definition. The constrainedElement role contains an Operation proxy that has a local reference to the initial Operation definition using **xmi:idref**. The second constraint is a proxy referencing a constraint definition using the **xmi:idref** of “idC2.” The third constraint is a proxy reference to the definition using **xmi:idref** to the constraint “idC3.” The fourth constraint is an XPointer reference proxy to the definition of the constraint using the **href** to the file doc2.xml with id “idC4.”

Following the definition of the operation and its 3 constraint proxies are the definitions of two of the constraints. The second document contains the third constraint definition.

The use and placement of references is freely determined by the document creator. It is likely that most documents will make internal and external references for a number of reasons: to minimize the amount of duplicate declarations, to compartmentalize the size of the document streams, or to refer to useful information outside the scope of transmission. For example, the **href** of an XLink could contain a query to a repository that will recall additional related information. Or there may be a set of XMI documents created, one file per package to be transferred, where there are relationships between the packages.

7.11 Tailoring Schema Production

This sub clause describes how to tailor schema production by specifying particular MOF tags to augment a MOF model. It also explains the impact the tailored schemas have on document production.

Note that the MOF definition of the association between ModelElement and Tag is not a composition and does not have a reference as part of ModelElement. This allows Tags to be contained in separate Packages and ‘remotely’ reference the tagged elements. For XMI purposes this means that the following tags can be incrementally added to an existing model without needing to be embedded in it - and thus changing it. Typically, the Tags could be in a separate Package and a ‘super’ package could import (via PackageImport) this Tags package and the model package to drive the Schema generation. This conveniently allows different Tag sets to be used with the same model (there would be a separate ‘super’ package for each). And the ‘super’ package extent allows runtime model access to the Tags package for introspection of the tags that were used for the generation.

7.11.1 XMI Tag Values

The following table specifies the XMI tags that allow you to tailor the schemas that are produced and the documents that are produced using XMI. Each of the names has a prefix of “org.omg.xmi.” The prefix is not included in the names to make the table easier to read.

Table 7.1 - XMI Tag Values Summary

Tag Name	Value Type	Default value	Description
Naming tags			
xmiName	string	nil	Provides an alternate name from the MOF name for writing to XMI. Useful in cases where the MOF name has characters that conflict with XML. This value is used rather than the MOF name.
nsURI	string	nil	The namespace URI of the MOF package.
nsPrefix	string	Package::name	The namespace prefix of the MOF package; this is used in schemas. (Any legal XML prefix may be used in documents.)
XML Syntax tags			
serialize	string	non-derived	If non-derived, then the MOF construct is serialized unless it is derived. ‘true’ forces the construct to be serialized regardless of whether it is derived; and ‘false’ suppresses it regardless.
attribute	boolean	false	If false, do not serialize the MOF construct as an XML attribute unless element is also false.
element	boolean	false	If false, do not serialize the MOF construct as an XML element unless attribute is also false.
remoteOnly	boolean	false	If set on one end of a bidirectional relationship, only serializes that end if it is remote.
href	boolean	false	If true, use the href attribute rather than the idref attribute for links within a document. This also prohibits the use of XML attributes for class-typed properties.
valueSeparator	string	","	The value of a structured datatype (i.e., a datatype that has properties) is represented as the values of the properties separated (by default) by a comma. This tag allows the specification of a different separator.
Ordering			
superClassFirst	boolean	false	If true, serialize the super class content first.
ordered	boolean	false	If true, serialize object content in the order it is defined in a MOF model. Where properties have isOrdered=false then the order used is alphabetic order of the string rendition of that property value.

Table 7.1 - XMI Tag Values Summary

Tag Name	Value Type	Default value	Description
Content			
includeNils	boolean	true	If false, do not serialize nil values.
XML Schema Production			
enforceMaximumMultiplicity	boolean	false	If true, enforce maximum multiplicities; otherwise, they are “unbounded.”
enforceMinimumMultiplicity	boolean	false	If true, enforce minimum multiplicities; otherwise, they are “0.”
useSchemaExtensions	boolean	false	If true, use schema extensions to represent inheritance in the MOF model.
schemaType	string	nil	The name of a datatype defined in the XML Schema Datatype specification.
contentType	string	complex	Defines the schema content type. Other valid values are: any, mixed, empty, and simple.
processContents	string	strict	If the contentType is any, this tag is used to specify the value of the processContents attribute of the any element. Other valid values are: lax, skip.
form	string	nil	Specifies the value of the form attribute for attributes. Other valid values are qualifed and unqualified.
allowMetamodelExtension	boolean	false	Whether the XML Schema generated should allow for the original metamodel to be extended – allowing subclasses outside the metamodel to be substituted.
flattenStructuredDataTypes	boolean	false	If set to true, instances of non-nested structured datatypes with field multiplicities [1..1] may be serialized as a string of values separated by the separator defined by the valueSeparator tag.

7.11.2 Tag Value Constraints

There are constraints on the values of the XMI tags in addition to the ones specified in the above table. Here is a list of them:

- If org.omg.xmi.includeNils is true (the default), and the value of a property is empty, the value must be represented by an XML element regardless of the value of the org.omg.xmi.attribute tag.
- For class scope or multi-valued construct scope, if org.omg.xmi.enforceMinimumMultiplicity or org.omg.xmi.enforceMaximumMultiplicity is true, the org.omg.xmi.ordered tag must be true as well (to validate multiplicities, schemas require element content to be serialized in a particular order). The multiplicity tags require the use of serializing in elements. For singlevalued construct scope, when org.omg.xmi.enforceMinimumMultiplicity is true and lower bound of the multiplicity = 0 or 1, then there is no need to enforce the use of the ordered tag or the use of elements for document serialization.

- If the lower bound for a property is null, the org.omg.xmi.enforceMinimumMultiplicity tag is ignored, and minimum multiplicity is not enforced in the Schema (minimum multiplicity is effectively “0”). Similarly, if the upper bound for a property is null, the org.omg.xmi.enforceMaximumMultiplicity tag is ignored, and maximum multiplicity is not enforced in the Schema (the multiplicity is effectively unbounded).
- If the MOF model has multiple inheritance, then org.omg.xmi.useSchemaExtensions must be false,
- If org.omg.xmi.useSchemaExtensions is true, org.omg.xmi.superClassFirst must be true also.
- If org.omg.xmi.href is true, org.omg.xmi.element must be true as well for every reference that is serialized.
- The org.omg.xmi.attribute tag may not be specified on containment references, multi-valued attributes, attributes without simple data types, or features with the following tags as true: org.omg.xmi.element, org.omg.xmi.includeNils, org.omg.xmi.enforceMinimumMultiplicity, org.omg.xmi.enforceMaximumMultiplicity, and org.omg.xmi.href.
- If org.omg.xmi.href is true, the org.omg.xmi.attribute must be false and org.omg.xmi.element must be true.

7.11.3 XML element vs XML attribute

You may choose features (DataType-typed or Class-typed properties) to appear as XML attributes, XML elements, or both, based on the model and tags in the model. The following is a list of the conditions for mapping a feature to an XML construct.

XML attribute only

- The feature has an attribute tag set to true.

XML element only

- The feature is a containment, or
- has an org.omg.xmi.element tag set to true, or
- has an org.omg.xmi.href tag set to true, or
- is a multi-valued property, or
- is a property whose type is not a simple data type.

Both XML attribute and element

- The default

7.11.4 Summary of XMI Tag Scope and Affect

The table below contains the following information:

- Affect: the second column identifies the MOF constructs that are affected by a given XMI tag.
- Scope: columns 3 through 5 identify the scope of each tag. If the scope is Package Scope, a tag set on the package applies to all the affected constructs within the package. If the scope is class Scope, a tag set on the class applies to all affected constructs within the class. If the scope is Construct Scope, the tag affects only the specific construct it is set on.

By setting a tag on a Package or Class, you avoid setting the same tags repeatedly for classes in the package, and for Properties belonging to the Class. For example, the org.omg.xmi.element tag applies to Properties. If the org.omg.xmi.element tag is set to *true* for a Class, the Class itself is not affected, but each Property belonging to the Class is treated as if the org.omg.xmi.element tag were set to *true* for all of them.

The org.omg.xmi.xmiName, org.omg.xmi.serialize, org.omg.xmi.contentType, org.omg.xmi.schemaType, and org.omg.xmi.remoteOnly tags apply only to the constructs for which they are specified. For example, setting the org.omg.xmi.xmiName of a MOF class to “c” means that the name “c” should be used in XMI schemas and documents for that class; it does not constrain the names of the features of the class.

Table 7.2 - XMI Tags, the MOF Constructs They Affect, and Their Scope

XMI Tag	MOF Constructs Affected	Package Scope	Class Scope	Construct Scope
xmiName	Class, Property			X
ordered	Class, Property	X	X	X
serialize	Property			X
element	Property	X	X	X
attribute	Property	X	X	X
enforceMaximumMultiplicity	Property	X	X	X
enforceMinimumMultiplicity	Property	X	X	X
form	Property	X	X	X
remoteOnly	Property			X
href	Property	X	X	X
includeNils	Property	X	X	X
schemaType	Property			X
valueSeparator	Property	X	X	X
allowMetamodelExtension	Property	X	X	X
nsURI	Package	X		X
flattenStructuredDataTypes	Property	X	X	X
nsPrefix	Package	X		X
processContents	Class, Property, Package	X	X	X
useSchemaExtensions	Class	X	X	X
contentType	Class			X
superClassFirst	Class	X	X	X

7.11.5 Effects on Document Production

The values of the XMI tags affect how documents are serialized. In general, the more validation a schema performs, the more restrictions there are on the XMI documents that validate using the schemas. There are two reasons for this. First, schemas cannot validate multiplicities without imposing an order on element content. Second, if the schema extension mechanism is used, superclass elements must be serialized in element content before subclass elements.

Here are some examples of how the XMI tags affect document production. Assume that there is a MOF model with class "Super" and class "Sub." Sub inherits from Super. Super has attribute a of type String, and Sub has attribute b of type String. If the namespace URI is "URI," and the prefix is "p," here is the default schema produced from the MOF model:

```
<xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    targetNamespace="URI"
    xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:p="URI">

<xsd:import
    namespace="http://www.omg.org/spec/XMI/20110701"
    schemaLocation="XMI.xsd"/>

<xsd:complexType name="Super">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="a" type="xsd:string"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="a" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Super" type="p:Super"/>

<xsd:complexType name="Sub">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="a" type="xsd:string"/>
    <xsd:element name="b" type="xsd:string"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="a" type="xsd:string" use="optional"/>
  <xsd:attribute name="b" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Sub" type="p:Sub"/>

</xsd:schema>
```

Note that the content model for Sub allows attribute a or attribute b to be serialized first if they are serialized as elements. For example, if p is the namespace prefix for a namespace whose uri is “URI” in an XML document, the following instance of Sub validates against the default schema:

```
<p:Sub>
  <b>Value1</b>
  <a>Value2</a>
</p:Sub>
```

The following is also legal:

```
<p:Sub>
  <a>Value2</a>
  <b>Value1</b>
</p:Sub>
```

If org.omg.xmi.useSchemaExtensions is true, the declaration of the Sub complexType uses the XML schema extension mechanism, as follows:

```
<xsd:complexType name="Sub">
  <xsd:complexContent>
    <xsd:extension base="p:Super">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="b" type="xsd:string"/>
      </xsd:choice>
      <xsd:attribute name="b" type="xsd:string" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

This declaration of the Sub type imposes an ordering on the content of Sub instances. With this declaration attribute a must be serialized before attribute b, so the first instance of Sub above does not validate with this schema, but the second does validate. Also, any xmi:extension elements must be serialized in Sub instances before elements corresponding to attribute b.

7.11.6 Example: Customize the XML Schema for a GIS Model

This example uses a model from GIS. It shows the flexibility that XMI tags give the modeler in tailoring an XML schema for a metamodel: in this case an EMOF metamodel.

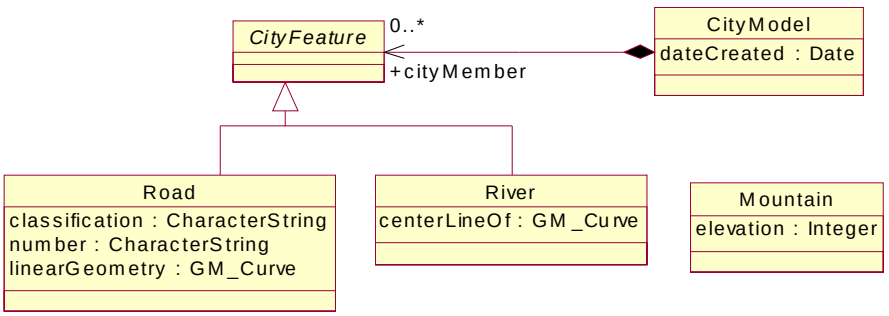


Figure 7.3 - GIS Cambridge model

Note: The definition of type “GM_Curve” is intentionally not shown to keep the example focused and simple.

The default XML schema for this model is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xmi="http://www.omg.org/spec/XMI/20110701">

<xsd:import namespace="http://www.omg.org/spec/XMI/20110701" schemaLocation="XMI.xsd"/>

<xsd:complexType name="Road">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="classification" type="xsd:string" nillable="true"/>
    <xsd:element name="number" type="xsd:string" nillable="true"/>
    <xsd:element name="linearGeometry" type="xsd:string" nillable="true"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="classification" type="xsd:string" use="optional"/>
  <xsd:attribute name="number" type="xsd:string" use="optional"/>
  <xsd:attribute name="linearGeometry" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Road" type="Road"/>

<xsd:complexType name="CityFeature">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>

<xsd:element name="CityFeature" type="CityFeature"/>

<xsd:complexType name="River">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="centerLineOf" type="xsd:string" nillable="true"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="centerLineOf" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="River" type="River"/>

<xsd:complexType name="CityModel">
```

```
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="dateCreated" type="xsd:string" nillable="true"/>
    <xsd:element name="cityMember" type="CityFeature" minOccurs=”0” maxOccurs=”unbounded”/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="dateCreated" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="CityModel" type="CityModel"/>

<xsd:complexType name="Mountain">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="elevation" type="xsd:int" nillable="true"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="elevation" type="xsd:int" use="optional"/>
</xsd:complexType>

<xsd:element name="Mountain" type="Mountain"/>

<xsd:element name="Cambridge">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="Road"/>
      <xsd:element ref="CityFeature"/>
      <xsd:element ref="River"/>
      <xsd:element ref="CityModel"/>
      <xsd:element ref="Mountain"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Some things to notice about the default schema include:

- It does not use the XML schema extension element to express inheritance. This is because XML schema does not support multiple inheritance. Instead, declarations of the superclass(es) are repeated in the subclass. For models that only use single inheritance, the ‘org.omg.xmi.useSchemaExtensions’ tag signals that the xsd:extension element should be generated rather than repeating the declarations.
- XMI allows attributes with primitive values (like String) to be serialized either as XML attributes or as XML elements. This makes the default schema verbose, since it needs to take into account both possibilities. The XMI ‘org.omg.xmi.element’ tag can be used to signal that attributes can be serialized only as XML elements. Similarly, the XMI ‘org.omg.xmi.attribute’ tag signals the other case.

- The dateCreated attribute of the CityModel class has type string. This is because Date is not in the set of MOF primitive datatypes. This could be addressed by including datatype ‘Date’ in the model and having XMI tag ‘org.omg.xmi.schemaType’ with value http://www.w3.org/2001/XMLSchema#date.
- xsd:choice is used to represent attributes, but does not constrain cardinality. This makes it possible to leave out an attribute or to repeat it a number of times without being caught when validating a document with the schema. You can set XMI tags “org.omg.xmi.enforceMaximumMultiplicity,” “org.omg.xmi.enforceMinimumMultiplicity,” and “org.omg.xmi.ordered” to “true.” Also, the XMI tag “org.omg.xmi.attribute” must be “false” (the default). There is a disadvantage to using these tags: in order to validate multiplicity, schemas require the XML elements be serialized in the same order as declared in the schema.
- The schema declaration for cityMember has type xmi:Any instead of type CityFeature:

```
<xsd:element name="cityMember" type="xmi:Any"/>
```

It would be useful to be able to constrain the attribute to the correct type - in this case CityFeature instead of Any. In the default case, where XMI tag org.omg.xmi.useSchemaExtensions=”false,” using xmi:Any instead of CityFeature allows the subclasses of CityFeature (Road or River) to be serialized and validated by the schema. If we used type=CityFeature, the validator would not recognize the additional attributes in Road and River, and the document would be considered invalid. However, with XMI tag org.omg.xmi.useSchemaExtensions=”true,” the correct type can safely be used.

By applying all the XMI tags described above, we can tailor the schema to look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xmi="http://www.omg.org/spec/XMI/20110701">

<xsd:import namespace="http://www.omg.org/spec/XMI/20110701"
       schemaLocation="XMI.xsd"/>
<xsd:annotation>
 <xsd:documentation>PACKAGE: Cambridge</xsd:documentation>
</xsd:annotation>

<xsd:annotation>
 <xsd:documentation>CLASS: Road</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="Road">
 <xsd:extension base="CityFeature">
  <xsd:sequence>
   <xsd:element name="classification" type="xsd:string" nillable="true"/>
   <xsd:element name="number" type="xsd:string" nillable="true"/>
   <xsd:element name="linearGeometry" type="xsd:string" nillable="true"/>
  </xsd:sequence>
 </xsd:extension>
</xsd:complexType>

<xsd:element name="Road" type="Road"/>

<xsd:annotation>
 <xsd:documentation>CLASS: CityFeature</xsd:documentation>
```

```
</xsd:annotation>

<xsd:complexType name="CityFeature">
  <xsd:sequence>
    <xsd:element ref="xmi:Extension" minOccurs="0"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>

<xsd:element name="CityFeature" type="CityFeature"/>

<xsd:annotation>
  <xsd:documentation>CLASS: River</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="River">
  <xsd:extension base="CityFeature">
    <xsd:sequence>
      <xsd:element name="centerLineOf" type="xsd:string" nillable="true"/>
    </xsd:sequence>
  </xsd:extension>
</xsd:complexType>

<xsd:element name="River" type="River"/>

<xsd:annotation>
  <xsd:documentation>CLASS: CityModel</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="CityModel">
  <xsd:sequence>
    <xsd:element name="dateCreated" type="xsd:date" nillable="true"/>
    <xsd:element name="cityMember" type="CityMember" minOccurs="0"
                 maxOccurs="unbounded"/>
    <xsd:element ref="xmi:Extension" minOccurs="0"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>

<xsd:element name="CityModel" type="CityModel"/>

<xsd:annotation>
  <xsd:documentation>CLASS: Mountain</xsd:documentation>
</xsd:annotation>
```

```
<xsd:complexType name="Mountain">
  <xsd:sequence>
    <xsd:element name="elevation" type="xsd:int" nillable="true"/>
    <xsd:element ref="xmi:Extension" minOccurs="0"
                 maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>

<xsd:element name="Mountain" type="Mountain"/>

<xsd:element name="Cambridge">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="Road"/>
      <xsd:element ref="CityFeature"/>
      <xsd:element ref="River"/>
      <xsd:element ref="CityModel"/>
      <xsd:element ref="Mountain"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

7.12 Transmitting Metadata Differences

The goal is to provide a mechanism for specifying the differences between documents so that an entire document does not need to be transmitted each time. This design does not specify an algorithm for computing the differences, just a form for transmitting them.

Up to now we have seen how to transmit an incomplete or full model. This way of working may not be adequate for all environments. More precisely, we could mention environments where there are many model changes that must be transmitted very quickly to other users. For these environments the full model transmission can be very resource consuming (time, network traffic, ...) making it very difficult or even not viable for finding solutions for cooperative work.

The most viable way to solve this problem is to transmit only the model changes that occur. In this way different instances of a model can be maintained and synchronized more easily and economically. Concurrent work of a group of users becomes possible with a simple mechanism to synchronize models. Transmitting less information allows synchronizing models more efficiently.

7.12.1 Definitions

The idea is to transmit only the changes made to the model (differences between new and old model) together with the necessary information to be able to apply the changes to the old model.

A. New - Old = Difference

Model differencing is the comparison of two models and identifying the differences between them in a reversible fashion. The difference is expressed in terms of changes made to the old document to arrive at the new document.

B. New = Old + Difference

Model merging is the ability to combine difference information plus a common reference model to construct the appropriate new model.

7.12.2 Differences

Differences must be applied in the order defined. A later difference may refer to information added by a previous difference by linking to its contents. Model integrity requires that all the differences transmitted are applied. The following are the types of differences recognized, the information transmitted, and the changes they represent:

- Delete (reference to deleted elements): The Delete element refers to particular elements and specifies a deep removal of the referenced elements and all of their contained elements (determined through composite associations).
- Add (reference to containing element, new elements, optional position): The Add element refers to a particular element of the old model and specifies a deep addition. The elements and their contents are added at the optional position specified relative to elements of that type within the target element (e.g., packagedElement), the default being at the end. The optional position form is based on XPointer's position form. 1 means the first position, -1 means the last position, and higher numbers count across the contents in the specified direction.
- Replace (reference to replaced elements, replacement elements, optional position): This operation removes the old elements from their container (they must all have the same container) but does not delete them. The new elements are added at the specified position within the same container (as per Add).

7.12.3 XMI encoding

The following are the elements used to encode the differences.

delete

The target element's link attributes contain a link to the element to be deleted.

add

The *addition* attribute of **add** references the elements to be added, which must all be of the same XML type. The *target* element's link attributes contain a link to the single container element for the new ones and an optional position. The numbering corresponds to XPointer numbering, where 1 is the first and -1 is the last element and it is used to count the elements of the same type as the ones to be added within the container. The type used is that of the XML element (which typically represents a composite property) as opposed to the xmi:type. The new elements are positioned after the element with the indicated position.

replace

The target of **replace** is the set of elements to be replaced that must all be of the same XML type and have the same container. The *replacement* attribute of **replace** references the elements to be added to that container, and must again all be of the same XML type. The optional *position* attribute uses numbering corresponding to XPointer numbering, where 1 is the first and -1 is the last element and it is used to count the elements of the same type as the ones to be added within the container (after removal of the target elements). The type used is that of the XML element (which typically represents a composite property) as opposed to the xmi:type. The new elements are positioned after the element with the indicated position.

7.12.4 Example

This example will delete a class and its attributes, add two classes, and replace a class within a package. The original document, called original.xml:

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
  xmlns:xmi=" http://www.omg.org/spec/UML/20110701">
 <uml:Package xmi:id="ppp" xmi:label="p1">
  <packagedElement xmi:type="uml:Class" xmi:id="ccc" name="c1">
   <ownedAttribute xmi:type="uml:Property" name="a1"/>
   <ownedAttribute xmi:type="uml:Property " name ="a2"/>
  </packagedElement >
 </uml:Package>
</xmi:XMI>
```

The differences document:

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
  xmlns:xmi=" http://www.omg.org/spec/UML/20110701">
 <difference xmi:type="xmi:Delete">
  <target href="original.xml#ccc"/>
 </difference/>
 <difference xmi:type="xmi:Add" addition="Class_1 Class_2">
  <target href="original.xml#ppp"/>
 </difference>
 <packagedElement xmi:type="uml:Class" xmi:id="Class_1" name="c2">
 <packagedElement xmi:type="uml:Class" xmi:id="Class_2" name="c3">
 <difference xmi:type="xmi:Replace" position="0" replacement="c4">
  <target href="original.xml#Class_2"/>
 </difference>
 <packagedElement xmi:type="uml:Class" xmi:id="Class_3" name="c4">
</xmi:XMI>
```

Here's how the 3 differences change the document as they're applied. The delete:

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
  xmlns:xmi=" http://www.omg.org/spec/UML/20110701">
 <uml:Package xmi:id="ppp" xmi:label="p1">
 </uml:Package>
</xmi:XMI>
```

Next, the add:

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
  xmlns:xmi=" http://www.omg.org/spec/UML/20110701">
 <uml:Package xmi:id="ppp" xmi:label="p1">
  <packagedElement xmi:type="uml:Class" xmi:id="Class_1" name="c2">
  <packagedElement xmi:type="uml:Class" xmi:id="Class_2" name="c3">
 </uml:Package>
</xmi:XMI>
```

Finally, the replace:

```
<xmi:XMI xmlns:uml="http://www.omg.org/spec/UML/20110701"
  xmlns:xmi=" http://www.omg.org/spec/UML/20110701">
 <uml:Package xmi:id="ppp" xmi:label="p1">
  <packagedElement xmi:type="uml:Class" xmi:id="Class_3" name="c4">
  <packagedElement xmi:type="uml:Class" xmi:id="Class_1" name="c2">
 </uml:Package>
 <uml:Class xmi:type="uml:Class" xmi:id="Class_2" name="c3">
</xmi:XMI>
```

Note that Class_2 is not deleted but merely removed from the package ppp.

7.13 Document Exchange with Multiple Tools

This sub clause contains a recommendation for an optional methodology that can be used when multiple tools interchange documents. In this methodology, the **xmi:uuid** and extensions are used together to preserve tool-specific information. In particular, tools may have particular requirements on their IDs, which makes ID interchange difficult. Extensions are used to hold tool-specific information, including tool-specific IDs.

The basic policy is that the XML ID is assigned by the tool that initially creates a construct. The **UUID** will most likely be the same as the ID the tool would choose for its own use. Any other modifiers of the document must preserve the original **UUID**, but may add their own as part of their extensions.

In order to allow the use of such schemes as outlined here, XMI Extensions must be persistently maintained by the importing tool.

7.13.1 Definitions

General:

- MC - Model construct. An XML element that contains an xmi.uuid attribute**.**
- Extension - Extensions use the extension element. Extensions to MCs may be nested in MCs, linked to the extensions section(s) of the document, or linked outside the document. Each extension contains a tool-specific identifier in the extender attribute. Extensions are considered private to a particular tool. An MC may have zero or more extensions. Extensions may be nested.

IDs:

- **xmi:uuid** - The universally unique ID of an MC, expressed as the **xmi:uuid** attribute. Example: <Class **xmi:uuid**=”ABCDEFGH”>
- **extenderID** - The tool-specific ID of an MC. The extenderID is stored in an extension of the MC when it differs from the **xmi:uuid**.

Tool ID policies:

Every tool is either Open or Closed.

- Open tool - A tool that will accept any **xmi:uuid** as its own. Open tools do not need to add extensions to contain a tool-specific id.
- Closed tool - A tool that will not accept an **xmi:uuid** created by another tool. Closed tools store their ids in the **extenderID** attribute of an XMI.extension. The **extender** attribute of the XMI.extension is set to the name of the closed tool.

7.13.2 Procedures

Document Creation:

- The Creating Tool writes a new XMI document. Each MC is assigned an **xmi:uuid**. If the **xmi:uuid** differs from the **extenderID**, an **extension** for that tool is added containing the extenderID.

Document Import:

- The importing tool reads an existing XMI document. Extensions from other tools may be stored internally but not interpreted in the event a Modification will occur at a later time. One of the following cases occurs:

  1. If the importing tool is an Open tool, the **xmi:uuid**s are accepted internally and no conversion is needed.

  2. If the importing tool is a closed tool, the tool looks for a contained extension that it recognizes (identified by extender) with an extenderID. If one does not exist, the importing tool creates its own internal id.

Document Modification:

- The modifying tool writes the MCs and any extensions preserved from import.
- For new MCs, the MC is assigned an **xmi:uuid**.
- Closed tools add an **extension** including their internal id in the **extenderID**.

7.13.3 Example

This sub clause describes a scenario in which Tool1 creates an XMI document that is imported by Tool2, then exported to Tool1, and then a third tool imports the document. All the tools are closed tools.

1.A model is created in Tool1 with one class and written in XMI.

```
<UML:Class xmi:label="c1" xmi:uuid="abcdefgh"/>
```

2.The class is imported into Tool2. Tool2 assigns extenderID “JKLMNOPQRST.” A second class is added with name “c2” and uuid “X012345678.”

3.The model is merged back to XMI:

```
<UML:Class xmi:label="c1" xmi:uuid="abcdefgh">
      <xmi:Extension extender="Tool2" extenderID="JKLMNOPQRST"/>
</UML:Class>
<UML:Class xmi:label="c2" xmi:uuid="X012345678"/>
```

4.The model is imported into Tool1. Tool1 assigns extenderID “ijklmnop” to “c2” and a new class “c3” is created with uuid “qrstuvwxyz.”

5.The model is merged back to XMI:

```
<UML:Class xmi:label="c1" xmi:uuid="abcdefgh">
      <xmi:Extension extender="Tool2" extenderID="JKLMNOPQRST"/>
</UML:Class>
<UML:Class xmi:label="c2" xmi:uuid="X012345678">
     <xmi:Extension extender="Tool1" extenderID="ijklmnop"/>
</UML:Class>
<UML:Class xmi:label="c3" xmi:uuid="qrstuvwxyz"/>
```

6.A third closed tool, Tool3, adds its ids:

```
<UML:Class xmi:label="c1" xmi:uuid="abcdefgh">
      <xmi:Extension extender="Tool2" extenderID="JKLMNOPQRST"/>
      <xmi:Extension extender="Tool3" extenderID="s1234"/>
</UML:Class>
<UML:Class xmi:label="c2" xmi:uuid="X012345678">
       <xmi:Extension extender="Tool1" extenderID="ijklmnop"/>
       <xmi:Extension extender="Tool3" extenderID="s5678"/>
</UML:Class>
<UML:Class xmi:label="c3" xmi:uuid="qrstuvwxyz">
        <xmi:Extension extender="Tool3" extenderID="s90ab"/>
</UML:Class>
```

7.An open tool imports and modifies the file. There are no changes because the **xmi:uuids** are used by the tool.

7.14 General Datatype Mechanism

The ability to support general data types in XMI has significant benefits. The applicability of XMI is significantly expanded since domain models are likely to have a set of domain-specific data types. This general solution allows the user to provide a domain datatype model with a defined mapping to the XML data types.

Data types are defined in the model and the XML serialization of the datatypes is described in terms of the XML schema datatypes.

MOF complex data types are treated as MOF classes with each field treated as a MOF attribute with a primitive type mapped to XML schema.

The Tag org.omg.xmi.schemaType indicates that this class is a datatype with XML schema mapping. The value of the tag indicates the schema type. For example, http://www.w3.org/2001/XMLSchema#int is the int datatype.

7.15 Import Reconciliation

The following are cases where an element in an imported XMI file will resolve to an existing element in the importer: only one of the following need apply:

- both elements have uuids that are identical.
- the XMI element has extenderID and extender that are identical to those associated with the element in the importer.
- both elements have identical values of a Property with isID=true.
- both elements are in the same extent and would have identical values for the basic URI scheme.

Should elements match as above, then the element in the importer is updated as follows:

- If property P is explicitly included in the XMI file, then the value of that property is updated to the value(s) from the XMI file. If multivalued, then any existing values not in the new set are removed.
- If P is included but empty in the XMI file, the property is unset; if mandatory, it is instead set to its default value.
- If P is not explicitly included in the XMI file, then any existing value in the importer is unchanged.

Should a matching element be referenced from the Differences element then the actions are carried out in order prior to the main import.

8 XML Schema Production

8.1 Purpose

This sub clause describes the rules for creating a schema from a MOF-based metamodel. The conformance rules are stated in Clause 2.

8.1.1 Notation for EBNF

The rule sets are stated in EBNF notation. Each rule is numbered for reference. Rules are written as rule number, rule name, for example 1a. SchemaStart. Text within quotation marks are literal values, for example “<xsd:element.” Text enclosed in double slashes represents a placeholder to be filled in with the appropriate external value, for example //Name of Attribute//. Literals should be enclosed in single or double quotation marks when used as the values for XML attributes in XML documents. The suffix “*” is used to indicate repetition of an item 0 or more times. The suffix “?” is used to indicate repetition of an item 0 or 1 times. The suffix “+” is used to indicate repetition of an item 1 or more times. The vertical bar “|” indicates a choice between two items. Parentheses “()” are used for grouping items together.

EBNF ignores white space; hence these rules do not specify white space treatment. However, since white space in XML is significant, the actual schema generation process must insert white space at the appropriate points.

8.2 XMI Version 2 Schemas

8.2.1 EBNF

The EBNF for XMI Version 2 schemas is listed below with rule descriptions between sub clauses.

```
1.  Schema             ::= 1a:SchemaStart
                           1d:Imports?
                           1e:FixedDeclarations
                           2:PackageSchema+
                           1f:SchemaEnd

1a. SchemaStart        ::= "<xsd:schema
                                xmlns:xsd='http://www.w3.org/2001/XMLSchema'
                                xmlns:xmi='http://www.omg.org/spec/XMI/20100901'
                               1b:NamespaceDecl*
                               1c:TargetNamespace?
                              ">"
1b. NamespaceDecl      ::= "xmlns:" //Namespace prefix// "="
                              "'" //Namespace URI// "'"
1c. TargetNamespace    ::= "targetNamespace='" //Namespace URI// "'"
1d. Imports::= //Import statements for referenced metamodels//
```

```
1e. FixedDeclarations ::= "<xsd:import
                               namespace='http://www.omg.org/spec/XMI/20100901'/>"
1f. SchemaEnd         ::= "</xsd:schema>"
1g. XMIFixedAttribs   ::= "<xsd:attribute ref='xmi:id'/>"
                          "<xsd:attributeGroup ref='xmi:ObjectAttribs'/>"
1h. Namespace         ::= ( //Name of prefix// ":" )?
```

1.	A schema consists of a schema XML element that contains import statements, fixed declarations, plus declarations for the contents of the Packages in the metamodel.
1a.	The schema XML element consists of the schema namespace attribute, namespace attributes for the other namespaces used in the schema, if any, and an optional target namespace attribute. These rules are written as if the namespace name for the schema namespace is “xsd” and the namespace name for the XMI namespace is “xmi,” but you can substitute other names for these namespace names and still conform to this specification.
1b.	Each namespace used in the schema must have a namespace attribute that identifies the namespace prefix and the namespace URI. If the namespace name is "" the attribute name should be “xmlns.” The namespace is taken by default from the URI property on the Package representing the metamodel, which may be overridden by the org.omg.xmi.nsURI tag. The prefix is declared by the org.omg.xmi.nsPrefix in the metamodel.
1c.	The target namespace is set to the *URI* property of the Package representing the metamodel.
1d.	For each PackageImport in the metamodel there is a XML Schema import element. The namespace attribute will be set to the URI property of the Package defining the metamodel. The schemaLocation attribute is optional for XMI and may be set to the location of the generated XMI-complaint XML Schema for that metamodel.
1e.	The schema declarations that are in the XMI namespace are listed in sub clause 8.2.2.
1f.	The end of the schema XML element.
1g.	The fixed XMI attributes present on the major elements provide element identity and element linking. The identity attribute name is “xmi:id.”
1h.	A namespace is a namespace prefix followed by a “:”. If no namespace prefix is given, the rule is a blank.

```
2.  PackageSchema    ::= ( 2:PackageSchema
                         | 3:ClassSchema
                         | 6:StructuredDataTypeDef
                         | 7:AssociationDef
                         | 8:EnumSchema)*
```

2.	The schema contribution from a Package consists of the declarations for any contained Packages, Classes, Structured Data Types (those with properties), Associations and Enumerations. The order of definitions within the package is by element type (which includes their subtypes) as follows: Package, Class, Datatype, and alphabetically byname within each element type

```
3.  ClassSchema      ::= 4:ClassTypeDef
                         5:ClassElementDef
```

3.	The class schema contribution consists of a type declaration based on the Properties of the Class, and an element declaration for the Class itself.

```
4.  ClassTypeDef   ::= "<xsd:complexType name='" //Name of Class//
                           ("mixed='true'")?
                           "'>"
                           ( "<xsd:complexContent>"
                             "<xsd:extension base='" 4a:ClassTypeName "'")?
                           (("<xsd:choice minOccurs='0'
                                        maxOccurs='unbounded'>" )
                            | "<xsd:sequence>")?
                           ( 4b:ClassContents
                           | ( "<xsd:any minOccurs='0' maxOccurs='unbounded' )
                           processContents='" // ProcessContents Value // "'/>")?
                           ("</xsd:choice>"
                           | "</xsd:sequence>")?
                           4g:ClassAttListItems
                           ( "</xsd:extension>"
                             "</xsd:complexContent>" )?
                           "</xsd:complexType>
4a. ClassTypeName     ::= 1h:Namespace //Name of Class//
4b. ClassContents     ::= 4d:ClassAttributes
                          4e:ClassReferences
                          4f:ClassCompositions
                          4c:Extension
4c. Extension         ::= ("<xsd:element ref='xmi:extension'/>")*
4d. ClassAttributes   ::= ("<xsd:element name='" //Name of DataType-typed Property// "'"
                             ("nillable='true'")?
                             ( 4m:MinOccursAttrib )?
                             ( 4n:MaxOccursAttrib )?
                             (("type='" //Name of type// "'/>")
                              | ("type='xmi:Any'/>")) )*
4e. ClassReferences   ::= ( "<xsd:element name='" //Name of Class-typed Property// "'"
                             ( 4m:MinOccursAttrib )?
                             ( 4n:MaxOccursAttrib )?
                             "xsd:attributeGroup ref='linkAttribs'")*
```

```
4f. ClassCompositions ::= ( "<xsd:element name='" //Name of Reference// "'"
                              ( 4m:MinOccursAttrib )?
                              ( 4n:MaxOccursAttrib )?
                              (("type='" 4a:ClassTypeName "'/>")
                               | ("type='xmi:Any'/>")/>")
                               | (" ><xsd:complexType><xsd:choice>"
                                    ("<ref='" 4a: ClassTypeName "'/>")*
                                 "></xsd:choice></xsd:complexType></xsd:element>" ) )*
4g. ClassAttListItems ::= 1g:XMIFixedAttribs 4h:ClassAttribAtts
4h. ClassAttribAtts   ::= ( 4i:ClassAttribRef
                          | 4j:ClassAttribData
                          | 4k:ClassAttribEnum )*
4i. ClassAttribRef    ::= "<xsd:attribute name='" //Name of attribute// "'"
                            ("type='xsd:IDREFS'" | "type='xsd:IDREF'" )
                            ("use='optional'" | use='required'" ) "/>"
                            | (QName ? "type='xsd:anyURI' use='optional'/>")}
4j. ClassAttribData   ::= "<xsd:attribute name='" //Name of attribute// "'"
                           "type='xsd:string' "
                           ("use='optional'" | "use='required'")
                           ("form='" // Form value // "'")?
                          "/>"
4k. ClassAttribEnum   ::= "<xsd:attribute name='" //Name of attribute// "'"
                              "type='" 8a:EnumTypeName "'"
                             ("use='optional'" | "use='required'"))
                              "/>"
4l. // rule deleted//
4m. MinOccursAttrib   ::= "minOccurs='" // Minimum // "'"
4n. MaxOccursAttrib   ::= "maxOccurs='" // Maximum // "'"
```

4.	These rules describe the declaration of a Class in the metamodel as an XML complex type with a content model and XML attributes. If either of the tags org.omg.xmi.enforceMaximumMultiplicity or org.omg.xmi.enforceMinimumMultiplicity is true, the contents of the class are put in a sequence; otherwise, they are put in a choice. If the org.omg.xmi.contentType tag is complex (the default), the class content declarations appear as defined by rule 4b; however, if the org.omg.xmi.contentType value is empty, they do not appear, and if the org.omg.xmi.contentType value is any, the "xsd:any" element declaration appears instead of the class content. If the org.omg.xmi.contentType value is mixed, then the mixed attribute is included. If .org.omg.xmi.useSchemaExtensions is true, the complex type for the class is derived by extension from the complex type for its superclass.
4a.	This rule is for a reference to the type for the class, which is the name of the Class prefixed by the namespace, if present and not the default namespace.

4b. 4c.	The complex type for the Class contains XML elements for the contained DataType-typed and Class-typed properties and Compositions of the Class, plus an extension element, regardless of whether they are marked as derived. The org.omg.xmi.serialize tag can be used to control whether these constructs are serialized. If org.omg.xmi.useSchemaExtensions is *false* or not present, inherited DataType-typed and Class-typed properties and Compositions are included; otherwise, only local DataType-typed and Class-typed properties, and Compositions are included.
4d.	The XML element name for each DataType-typed property of the Class is listed as part of the content model of the Class element. This includes the DataType-typed properties defined for the Class itself as well as all of the DataType-typed properties inherited from superclasses of the Class. The type is "xsd:string" for simple properties, the name of an enumeration for enumerated properties, or the value of the org.omg.xmi.schemaType tag, if present. For complex properties (possible in CMOF only), when org.omg.xmi.useSchemaExtensions is true, the name of the property type is used from rule 4, and when org.omg.xmi.useSchemaExtensions is false, xmi:Any is used. If the org.omg.xmi.includeNils tag is false, then the "nillable" attribute is not included in the declaration. If org.omg.xmi.enforceMinimumMultiplicity is true, the minOccurs attribute is included. If org.omg.xmi.enforceMaximumMultiplicity is true, the maxOccurs attribute is included.
4e.	This rule applies to Class-typed Properties that are not composite. The XML element name for each Class-typed Property of the Class is listed in the content model of the Class. The list includes the Class-typed Property defined for the Class itself, as well as all Class-typed Properties inherited from the superclasses of the Class. The type is the class name for the Property type if org.omg.xmi.useSchemaExtensions is "true" or if the org.omg.xmi.contentType is "complex;" otherwise, the type allows any object to be serialized. If org.omg.xmi.enforceMinimumMultiplicity is true, the minOccurs attribute is set to the lowerValue for the Property in the metamodel (unless it is 1 in which case minOccurs is omitted), otherwise it is set to "0" regardless. If org.omg.xmi.enforceMaximumMultiplicity is true, the maxOccurs attribute is set to the upperValue for the Property in the metamodel (unless it is 1 in which case maxOccurs is omitted), otherwise it is set to "unbounded" regardless For Class-typed Properties (following 7.8.5), when org.omg.xmi.useSchemaExtensions is true, the name of the property type is used from rule 4, and when org.omg.xmi.useSchemaExtensions is false, xmi:Any is used.
4f.	The XML element name for each Class-typed Property of the Class that is a composite is listed in the content model of the class. The list includes the Class-typed Properties defined for the Class itself, as well as all Class-typed Properties inherited from the superclasses of the Class. The type is the class name for the Class-typed Property type if org.omg.xmi.useSchemaExtensions is "true". If allowMetamodelExtensions is true, the type allows any object to be serialized and xmi:Any is used. Otherwise the list of candidate types is included: this is the name of the Property type (if not abstract) and the name of any non-abstract subclass. The list is in alphabetical order of immediate subclasses with depth first expansion. If org.omg.xmi.enforceMinimumMultiplicity is true, the minOccurs attribute is set to the lowerValue for the Property in the metamodel (unless it is 1 in which case minOccurs is omitted), otherwise it is set to "0" regardless. If org.omg.xmi.enforceMaximumMultiplicity is true, the maxOccurs attribute is set to the upperValue for the Property in the metamodel (unless it is 1 in which case maxOccurs is omitted), otherwise it is set to "unbounded" regardless. For Class-typed Properties (following 7.8.5 and/or 7.8.6), when org.omg.xmi.useSchemaExtensions is true, the name of the property type is used from rule 4, and when org.omg.xmi.useSchemaExtensions is false, xmi:Any is used.
4g. 4h.	In addition to the standard identification and linkage attributes, the attribute list of the Class element can contain XML attributes for the DataType-typed Properties and non-composite Class-typed Properties of the Class, when the limited facilities of the XML attribute syntax allow expression of the necessary values. Inherited properties are included unless the org.omg.xmi.useSchemaExtensions tag is true, in which case only local properties are included.
4i.	Class-typed Properties can be expressed as XML id reference XML attributes. If the upper bound of the multiplicity of the Property is 1, the type is IDREF otherwise it is IDREFS. If the lower bound is greater than 0 and org.omg.xmi.enforceMinimumMultiplicity is true, then use= required, otherwise use=optional..

4j.	Single-valued DataType-typed Properties of a Class that have a string representation for their data are mapped to XML attributes of type “xsd:string;” unless the org.omg.xmi.schemaType tag is present, in which case its value is used for the type. Multi-valued DataType-typed Properties of a Class cannot be so expressed, since the XML attribute syntax does not allow repetition of values. If the multiplicity of the attribute is exactly one, and org.omg.xmi.enforceMinimumMultiplicity is true, the attribute is required to be present.
4k.	Single-valued DataType-typed Properties that have enumerated values are mapped to XML attributes whose type is the enumerated type. If the multiplicity of the attribute is exactly one, and org.omg.xmi.enforceMinimumMultiplicity is true, the attribute is required to be present.
4m.	The value for minimum is the minimum multiplicity.
4n.	The value for maximum is the maximum multiplicity.

```
5.  ClassElementDef ::= "<xsd:element name='" //Name of class// "'"
                          "type=' 4a:ClassTypeName "'/>"
```

5.	This rule declares an XML element for a class in a metamodel.

```
6.  StructuredDataTypeDef::= "<xsd:complexType name='" //Name of DataType//
                              ("mixed='true'")?
                            "'>"
                              ( "<xsd:complexContent>"
                                "<xsd:extension base='" 6a:DataTypeName
                              ("<xsd:choice minOccurs='0'
                                           maxOccurs='unbounded'>" |
                                "<xsd:sequence>")?
                              ( 6b: DataTypeContents |
                                   "<xsd:any minOccurs='0' maxOccurs='u
                                      processContents='" // ProcessCont
                                  "'/>")?
                              ("</xsd:choice>" | "</xsd:sequence>")?
                              4g:ClassAttListItems
                              ( "</xsd:extension>"
                                "</xsd:complexContent>" )?
                              "</xsd:complexType>
6a. DataTypeName     ::= 1h:Namespace //Name of DataType//
6b. DataTypeContents     ::= 4d: ClassAttributes
                              4c:Extension
```

6.	These rules describe the declaration of a structured DataType in the metamodel as an XML complex type with a content model and XML attributes. The rules for declaring the Properties are the same as for Classes except that compositions and references do not apply to DataTypes
6a.	This rule is for a reference to the type for the class, which is the name of the DataType prefixed by the namespace, if present and not the default namespace.
6b.	The complex type for the DataType contains XML elements for the contained Properties, plus an extension element. The org.omg.xmi.serialize tag can be used to control whether these constructs are serialized. If org.omg.xmi.useSchemaExtensions is false or not present, inherited Properties are included; otherwise, only local Properties are include

```
7. AssociationDef    ::= "<xsd:element name='"' 7a:AssnElmtName '"'>"
                         "<xsd:complexType>
                         <xsd:choice minOccurs='0'
                                     maxOccurs='unbounded'>"
                         7b:AssnContents
                         "</xsd:choice>"
                         7d:AssnAtts
                         "</xsd:complexType>
                         </xsd:element>"
7a. AssnElmtName      ::= 1h:Namespace //Name of association//
7b. AssnContents      ::= 7c:AssnEndDef
                          7c:AssnEndDef
                          4c:Extension
7c. AssnEndDef        ::= "<xsd:element"
                          "name='" //Name of association end// "'>"
                          "<xsd:complexType>"
                          1g:XMIFixedAttribs
                          "</xsd:complexType>"
                          "</xsd:element>"
7d. AssnAtts          ::= 1g:XMIFixedAttribs
```

7.	The declaration of an Association consists of the names of its AssociationEnd XML elements (whether or not they are owned by the Association).
7a.	The use of the name of the XML element representing the Association.
7d.	The fixed identity and linking XML attributes are the Association XML attributes.

```
8.  EnumSchema      ::= "<xsd:simpleType name='" 8a:EnumType "'>"
                          "<xsd:restriction base='xsd:string'>"
                             8c:EnumLiterals
                           "</xsd:restriction>"
                            "</xsd:simpleType>"
8a. EnumTypeName    ::= 1h:Namespace 8b:EnumName
8b. EnumName        ::= // Name of enumeration //
8c. EnumLiterals    ::= ("<xsd:enumeration value='" 8d:EnumLiteral "'/>")+
8d. EnumLiteral     ::= // Name of enumeration literal //
```

8.	The enumeration schema contribution consists of a simple type derived from string whose legal values are the enumeration literals.
8a. 8b.	The name of the enumeration in XML schema references.
8c.	Each enumeration literal is put in the value XML attribute of an enumeration XML element.
8d.	The name of the enumeration literal.

8.2.2 Fixed Schema Declarations

There are some elements of the schema that are fixed, constituting a form of “boilerplate” necessary for every XMI 2 schema. These elements are described in this sub clause. These declarations are in the namespace “`http://www.omg.org/spec/XMI/20110701`”

Only the schema content of the fixed declarations is given here. For a complete description of the semantics of these declarations, see Clause 9.

The fixed declarations are contained in file XMI.xsd that may be imported into generated XML Schemas; or these declarations may be copied.

9 XML Document Production

9.1 Purpose

This clause specifies the XMI production of an XML document from a model based on the MOF 2 Core. The EMOF and CMOF packages of MOF 2 are shared by both UML 2 and MOF 2, so that XMI production rules support both. XMI describes an XML syntax that leverages the capability of XML schema. A set of objects are written to an XML document following the grammar defined here.

Key requirements for successful model interchange are:

- All significant aspects of the metadata are included in the XML document and can be recovered from it. No information is lost.
- The XML document is as compact as possible without loss of information.
- The XML document reflects the model being serialized in an intuitive way, in order to gain acceptance in the XML community at large.

The first requirement has been addressed by both XMI 1 and 2. The second and third requirements have been highlighted by organizations like eBXML and GIS, in which XMI 1 did not find acceptance. XMI 2 made great progress in reducing document size and improving readability. This specification maintains that progress, and streamlines the specification to make it easier to understand and implement.

9.2 Introduction

XMI's XML document production process is defined as a set of production rules. When these rules are applied to a model or model fragment, the result is an XML document. The inverse of these rules can be applied to an XML document to reconstruct the model or model fragment. In both cases, the rules are implicitly applied in the context of the specific metamodel for the metadata being interchanged.

The production rules are provided as a specification of the XML document production and consumption processes. They should not be viewed as prescribing any particular algorithm for XML producer or consumer implementations.

9.3 Serialization Model

The number of XML serialization patterns available for use by XMI are small. These serialization patterns are shown in Figure 9.1. By limiting the XMI EBNF rules to these patterns, then mapping each modeling element to a pattern, we can reduce the size of the EBNF in the XMI spec, thus simplifying it.

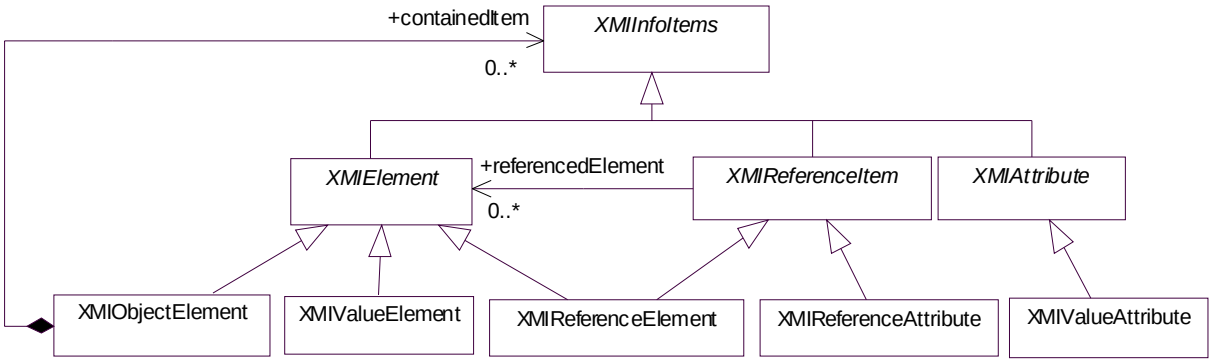


Figure 9.1 - Serialization Model

An **XMIObjectElement** is an XML element that can contain other information items (XML elements and attributes). An **XMIValueElement** is an XML element that can have a value, but cannot contain other XML elements or attributes. An **XMIReferenceElement** is an XML element with an idref or href attribute that references another XMIElement, by id, URI, or URI and XPointer. An **XMIReferenceAttribute** is an XML attribute that references an XMIElement by id. An **XMIValueAttribute** is simply an XML attribute with a value.

9.4 XMI Representation of the Core Packages

XMI production rules are defined for elements in UML as constrained by the EMOF and CMOF compliance levels in the MOF 2 Core. The rules for these packages are consistent and build upon each other: the rules for CMOF refine the EMOF rules. The rules are defined by mapping the model elements to the serialization model.

9.4.1 EMOF Package

The overall rules are shown in the table below.

Instance of Model Element	XMI Representation
A Class	XMIObjectElement
A Property, type is a PrimitiveType or Enumeration	Choice of: 1. XMIValueAttribute 2. Nested XMIValueElement The value of an Enumeration is its name. When the value of a Property is null, it is serialized as XMIValueElement with attribute nil=’true’. When the Property is multi-valued, it is serialized as multiple nested XMIValueElements.

Instance of Model Element	XMI Representation
A Property, type is not a PrimitiveType or Enumeration, isComposite = false	Choice of: 1. XMIReferenceAttribute 2. Nested XMIReferenceElement
A Property, type is not a PrimitiveType or Enumeration, isComposite = true	Normally, serialized properties with isComposite = true are serialized as nested XMIObjectElements. Exceptionally it may be the case that a containing object has more than one serialized class-typed property with isComposite = true that contain the same object or include it among their collection of objects. In such an exceptional case, because of MOF constraints, only one of those properties can have an opposite with a non-empty clot. Objects of the property with the non-empty opposite slot are serialized as nested XMIObjectElements, and the other references to the same object are serialized either as XMIReferenceAttributes or nested XMIReferenceElements.

The following additional rules are defined to suppress redundant information. They can be overriden using XMI tags.

- Derived information is not serialized.
- Properties whose values are the default values are not serialized.
- For Properties with isComposite=true, the opposite Property is not serialized.

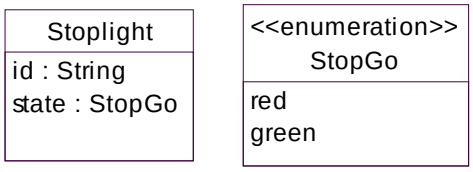
9.4.1.1 Examples

```
<complexco:Department xmi:id="Department_1"/>
```

Figure 9.1 - Instance of a class, the namespace name is its package name.

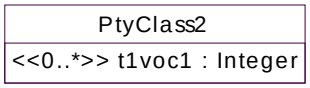
```
<Department number="13"/>
```

Figure 9.2 - Instance of a class with primitive typed property.




```
<Stoplight id="sl06" state="red"/>
```

Figure 9.3 - Instance of a class with an enumerated property.




```
<PtyClass2>
      <t1voc1>1001</t1voc1>
      <t1voc1>1001</t1voc1>
</PtyClass2>
```

Figure 9.4 - Multi-valued property, with each value serialized as an XML element.

Figure 9.5 - Composite property serialized as XML elements, the opposite property is not serialized.

```
<Department id="13">
    <member name="Glozic" xmi:type="Employee"/>
    <member name="Andrews" xmi:type="Employee"/>
</Department>
```

9.4.2 CMOF Package

The overall rules are the same as for the EMOF Package, with additions shown in the table below.

Instance of Model Element	XMI Representation
Properties of a DataType	Choice of: 1. XMIObjectElement 2. XMIValueAttribute 3. Nested XMIValueElement By default instances of structured datatypes are serialized as if they were classes, as described in Sub clause 7.8.7. This can be overridden by the tag org.omg.xmi.flattenStructuredDatatypes in which case the values of the Properties are serialized as a single string separated (by default) by commas. The default separator can be overridden by the XMI org.omg.xmi.valueSeparator tag.
An Association	XMIObjectElement

The following additional rules are defined to suppress redundant information. They can be overriden using XMI tags:

- Additional to the first bullet in rules for the EMOF package: for some metamodels, it may be desirable to serialize particular derived Properties instead of the information they are derived from because the derived form is more compact. In this case default behavior can be overridden by setting the org.omg.xmi.serialize tag to 'true' for the derived property. This means that either the base or derived form can be serialized, but for a particular metamodel construct only one may be chosen. To allow import, derived properties should only be made serializable if they are writeable (isReadOnly=false) and it is possible to reverse-derive the base information from the derived form.

- In the case where a Property redefines another Property, only the redefining Property is serialized. (Note that when serializing an instance of a concrete supertype whose Property has been redefined, the supertype is unaware of the redefinition, and the Property as defined on the supertype is serialized.)

No special serialization rules need to be defined for subsetted Properties. Following EMOF rule 1, when one of the subsetted or subsetting Properties is derived, it is not serialized by default. Properties that are not derived are serialized.

9.4.2.1 Examples

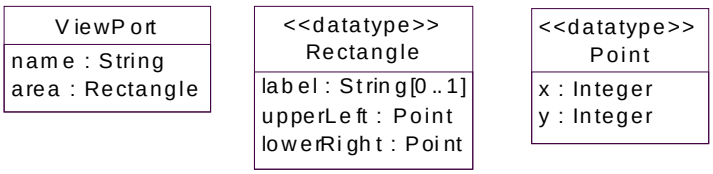


Figure 9.6 - Datatypes with properties

```
<display xmi:type=”g:Viewport” name=”normal”>
   <area xmi:type=”g:Rectangle”label=””>
      <upperLeft xmi:type=”g:Point” x="1" y="2"/>
      <lowerRight xmi:type=”g:Point” x="3" y="4"/>
   </area>
</display>
```

9.5 EBNF Rules Representation

The XML produced by XMI is represented here in Extended Backus Naur Form (EBNF). The XML specification does not require XML processors to preserve the order of XML attributes within an XML element. Therefore, although this grammar indicates that XML attributes should be serialized in a particular order for each XML element, the XML attributes may be serialized in any order. Also, XML attributes are normalized by XML processors, so whitespace may not be preserved. You may choose to serialize parts of objects as XML elements rather than XML attributes using the org.omg.xmi.element tag, as explained below.

The following sub clauses provide the production rules. The items in italics are terminal values.

9.5.1 Overall Document Structure

```
1:Document::= 1a:XMI | //Content Elements//

1a:XMI ::= "<xmi:XMI"
             1e:Namespaces ">"
           ( 2a:XMIObjectElement)*
           ( 3:Extension )*
           "</xmi:XMI>"
1b to 1d: // rules deleted //
```

```
1e:Namespaces          ::= 1f:XMINamespaceDecl ?
                           ( "xmlns:" 1h:NsPrefix "='" 1i:NsURI "'"
                            )*
1f:XMINamespaceDecl    ::= "xmlns:xmi='http://www.omg.org/spec/XMI/20100901'"
1g:Namespace           ::= ( 1h:NsPrefix> ":" )?
1h:NsPrefix              ::= Name of namespace prefix
1i:NsURI               ::= URI of namespace
```

1.	The content of an XMI document may be enclosed in an XMI XML element, but it does not need to be. The XML specification requires that there be one root element in an XML document for the document to be well formed. The XMI elements(identified via the XMI namespace) may appear anywhere in an arbitrary XML document, intermingled with non-XMI elements – though this can be somewhat restricted through the use of the org.omg.xmi.contentType tag
1a.	An XMI element has XML attributes that declare namespaces and specify the version of XMI, and the XMI element contains XML elements that make up the header, content, differences, and extensions for the XMI document.
1e.	The XMI namespace and the namespaces associated with a model must be declared or already be visible to the XMI element in the XML document. Since there is no requirement that the XMI XML element be the root element, these namespaces may be declared in XML elements that contain the XMI element. The namespace declarations must include the following if tag org.omg.xmi.includeNils is true for at least one Property in the metamodel, or org.omg.xmi.useSchemaExtensions is true: xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
1g.	The use of a namespace prefix, including a ":" separator. If the namespace prefix is blank, the result is the empty string.
1h.	A particular namespace prefix. Document producers can choose their own namespace prefixes, as long as doing so results in legal XML documents, or they may choose to use the value of the org.omg.xmi.nsPrefix tag.
1i.	The logical URI of the namespace. Note that namespaces are resolved to logical URIs, as opposed to physical ones, so that there is no expectation that this URI will be resolved and that there will be any information at that location. The URI is obtained from the org.omg.xmi.nsURI tag.

9.5.2 Object Structure

```
2:XMIElement           ::= 2a:XMIObjectElement
                         | 2b:XMIValueElement
                         | 2c:XMIReferenceElement
```

```
2a:XMIObjectElement   ::= ( "<" 2k:QName 2d:XMIAttributes "/>" )
                          | ( "<" 2k:QName 2d:XMIAttributes ">"
                             (2:XMIElement)*
                            "</" 2k:QName ">" )
2b:XMIValueElement    ::= ( "<" //xmiName// ">" //value//
                            "</" //xmiName// ">" )
                          | ( "<xsi:nil='true'/>" )
2c:XMIReferenceElement::= "<" //xmiName//
                           2l:LinkAttribs "/>"

2d:XMIAttributes      ::= (1e:Namespaces)?
                          (2e:IdentityAttribs)?
                          (2g:TypeAttrib)
                          (2h:FeatureAttrib)*
2e:IdentityAttribs    ::= ( 2f:IdAttribName "='" //id// "'")?
                         ( "xmi:label='" //label// "'" )?
                         ( "xmi:uuid='" //uuid// "'" )?
2f:IdAttribName       ::= "xmi:id"
2g:TypeAttrib         ::= "xmi:type='" 2k:QName "'"

2h:FeatureAttrib      ::= 2i:XMIValueAttribute
                          | 2j:XMIReferenceAttribute

2i:XMIValueAttribute  ::= //xmiName// "='" value "'"
2j:XMIReferenceAttribute ::= //xmiName// "='"
                                ( //refId// )+ "'"

2k:QName              ::= ( //prefix// ":" )? //xmiName//
2l:LinkAttribs         ::= "xmi:idref='" //refId// "'"
                           | 2m:Link

2m:Link                ::= "href='" 2n:URIref "'"

2n:URIref              ::= (2k:QName)? //URI reference//
```

2a.	An object has a starting element, contents, and a closing element. If the contents are empty, you may end the starting element with "/>". You use this production rule to serialize top-level objects and to serialize objects that are the values of Properties. If the object is a top-level object, the tag name is the namespace prefix followed by ":" and the XMI name for the object. The XMI name for the object is either the name of the object's class or the value of the org.omg.xmi.xmiName tag. If the object is the value of an attribute or reference, the XMI name is the name of the Property, or the value of the org.omg.xmi.xmiName tag. The namespace prefix is ignored for an object that is the value of a Property. The order of the elements for properties must follow any prescribed XML Schema ordering as defined in Clause 5 – even if no XML Schema has actually been generated. Furthermore if the ordered tag is true, the values of multi-valued properties must follow the order in the model (if isOrdered is true for the property) otherwise alphabetic order of the string rendition of that property value.
2b.	Each value of a property is represented by an XML element; for multi-valued properties, there is one XML element for each value. Null values may be serialized as well, unless the value of the org.omg.xmi.includeNils tag is "false," in which case you may not serialize null values.
2c.	Use this production rule to serialize a reference to an object using an XML element. If you use identity attributes, the values of the identity attributes must match the values of the identity attributes for the object that is referenced.
2d.	The XML attributes for an object are the optional start attributes, identity attributes, and attributes corresponding to an object's Properties. The start attributes must be written if the object is a top-level object and it.
2e.	The identity attributes consist of an optional id, label, and uuid. If the element has a MOF uuid, it may be used here.
2f.	The name of the identity attribute is "id" in the XMI namespace.
2g.	You must specify the class name using the "type" attribute. The value of this attribute is defined by the XML Schema Part 1: Structures specification to be a QName, consisting of a namespace prefix for the value's class (if there is one and it is not the default prefix for the document), a ":" and the name of the value's class. The QName can be either "xmi" (referring to the XMI namespace) or "xsi" (referring to the XML Schema Instance namespace). See the schema specification for more details. You may only use the XML schema instance type attribute if org.omg.xmi.useSchemaExtensions is true.
2h.	The XML attributes of the element correspond to Properties whose type is a data value or enumeration, or Class-typed properties whose values are objects in the document. A specific Property shall be serialized in an object either as an XML attribute or as XML elements but not both. You must not serialize a Property as an XML attribute if the value of the org.omg.xmi.element tag is "true." You must not serialize a Property at all if the value of the org.omg.xmi.serialize tag is "false;" or the value of that tag is "non-derived" and the Property has isDerived="true." You must not serialize a Class-typed Property at all if the org.omg.xmi.remoteOnly tag is true and the Property has a value that is an object in the same XML document. You may serialize classifier-level attributes with an object.

2i.	Use this production rule to serialize a Property whose type is not an object and whose value can be represented by a string. Multi-valued DataType-typed Properties cannot be serialized as XML attributes. If the Property's type is one of the types defined by the XML Schema Part 2: Datatypes specification, serialize the value as specified in that specification. Also use this production rule if the Property type is an enumeration and whose value is one of the legal enumeration literals. If the org.omg.xmi.xmiName is specified for the literal, the value of that tag should be used; otherwise, the name of the enumeration literal specified in the model is used.
2j.	Use this production rule to serialize Class-typed Properties whose values are objects that are serialized in the same document. The value of the XML attribute contains the XMI ID of each referenced object, separated by a space.
2k.	The name of an XMI element or attribute with an optional namespace prefix.
2l.	Use the idref attribute to specify the id of an XML element that is referenced in the document; use the href attribute to specify an XML element in another document. If the org.omg.xmi.href tag is "true," you must not use the idref attribute; use the href attribute for references within the document and across documents.
2m.	An XMI link. The value of the href attribute is a URI reference that refers to an XML element in another document or in the same document.
2n.	A URI Reference, optionally preceded by the type of the object being referenced. The URI reference refers to an XML element in another document or in the same document. For example, if the href is "someFile.xmi#someId," the href refers to an XML element in the "someFile.xmi" document whose XMI ID is "someId." If the URI reference is""#anotherId," it refers to an XML element whose XMI ID is "anotherId" in the same document. XLinks are also supported in XMI. See 7.10.2, 'Linking' for more information. See the W3C XLink and XPointer specification for production rules. The URI reference can be preceded by the type of the object being referenced. For example, a Property's type is a Classifier, which is abstract. When one of the concrete subclasses of Classifier is actually instantiated, it is not clear which one it is unless the URI is dereferenced. By serializing the QName emof:Class, you can tell it is a Class without needing to load and process the file at the URI.

9.5.3 Extension

```
3:Extension              ::= "<xmi:extension" "xmi:type='xmi:Extension'"
                             (" extender='" // extender // "'")?
                             (" extenderID='" // extenderID // "'")?
                             ">"
                             // Extension elements //
                             "</xmi:extension>"
```

3.	Extension elements may be provided to complement the serialized model with additional information, such as tool-specific diagram data, for example. Each extension element has an optional extender and extenderID attribute; its content can be anything (see for examples).

10 XML Schema Infoset Model

10.1 Introduction

This clause describes the MOF model for XML Schema using UML notation. The model is a straightforward mapping from the XML Schema specification: classes in the model have a direct correspondence to XML Schema components.

This model replaces the XSD model in the XMI 2.0 specification (http://www.omg.org/technology/documents/formal/xmi.htm), which was created prior to the introduction of the XML Infoset and the XML Schema abstract data model into the XML Schema specification. This model is called the XML Schema Infoset Model to distinguish it from the earlier version.

The specification of the XML Schema Infoset model assumes a strong working knowledge of XML Schema and refers throughout to the XML Schema specification for the detailed description of constructs that are defined by XML Schema.

The description of the model is divided into two sub clauses: the first describing elements of the model that primarily represent XML Structures, and the second describing elements that primarily represent XML Schema Datatypes.

The final sub clause shows an example of an XML Schema represented as an instance of the XSD Infoset model.

The model diagrams are color coded for easier reading:

- Yellow - concrete classes
- Turquoise - abstract classes
- Orange - enumerations
- Gray - datatypes

10.2 XML Schema Structures

This sub clause defines the model elements corresponding to XML Schema Subpart 1, Structures. There are eight diagrams in this sub clause. The first set of diagrams show aspects of the XSD Infoset model that represent the XML Schema abstract data model defined in the XML Schema specification:

- Figure 10.1
- Figure 10.2 on page 69
- Figure 10.3 on page 70
- Figure 10.4 on page 71

The next set of diagrams show aspects of the model that represent the XML Schema concrete syntax:

- Figure 10.5 on page 72
- Figure 10.6 on page 73
- Figure 10.7 on page 74

The final diagram (Figure 10.8 on page 75) shows how concrete components resolve into abstract components.

The sub clauses following these diagrams describe the model classes in detail. The sub clauses are alphabetically ordered by class name.

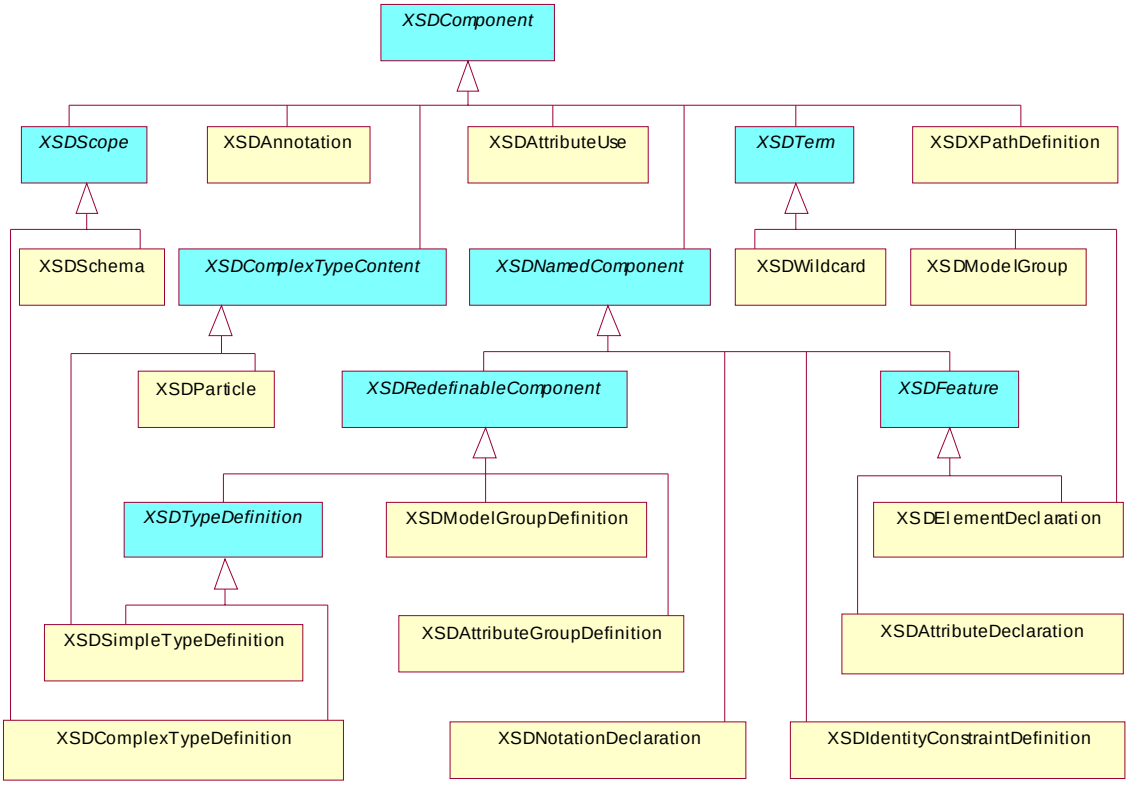


Figure 10.1 - Component Hierarchy

The Component Hierarchy diagram introduces classes representing the abstract XML Schema components. Schema components are the building blocks that comprise the abstract data model of the schema. An XML Schema is a set of schema components.

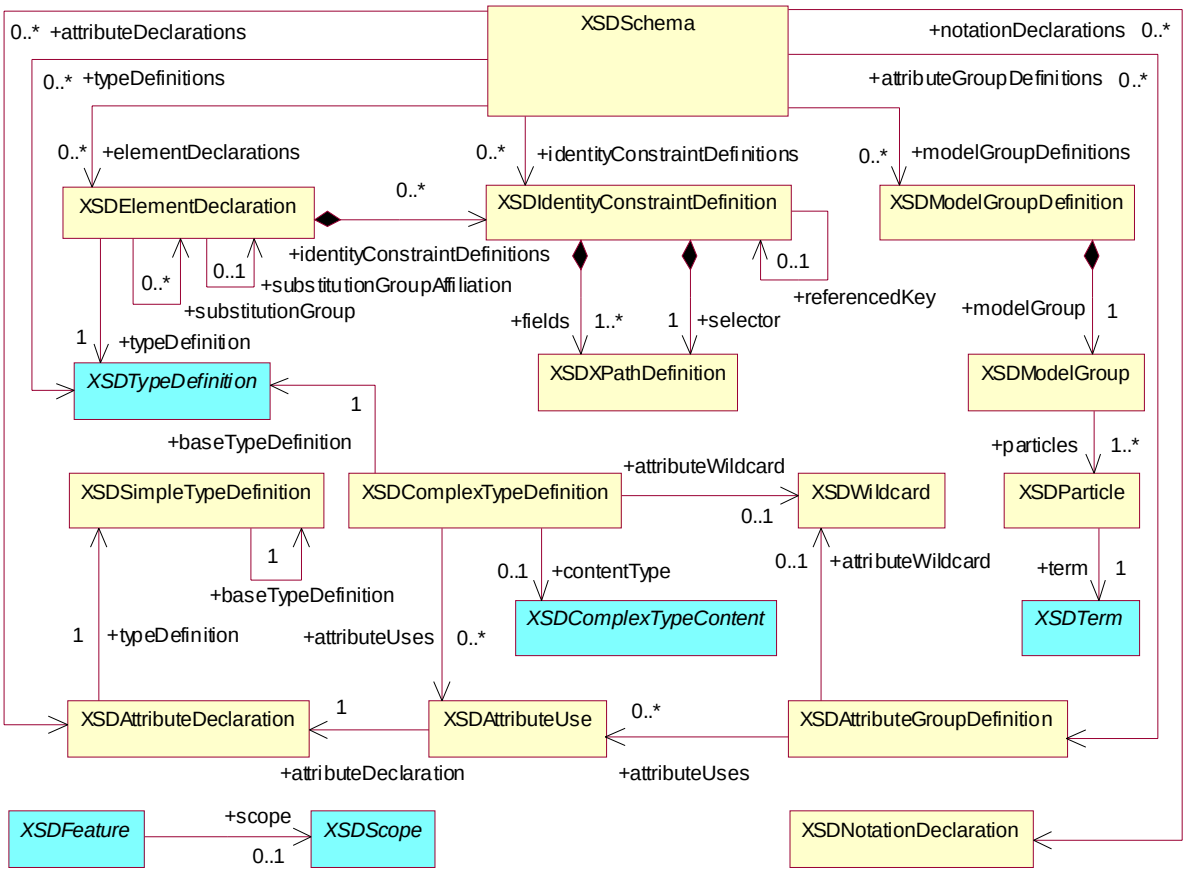


Figure 10.2 - Component Relations

The Component Relations diagram shows the interrelationships between XML Schema components: which components can contain or reference other components. The XSDAnnotation relations are shown separately in Figure 10.4 on page 71. This is closely aligned with the (non-normative) Schema Components Diagram in XML Schema Part 1: Structures.

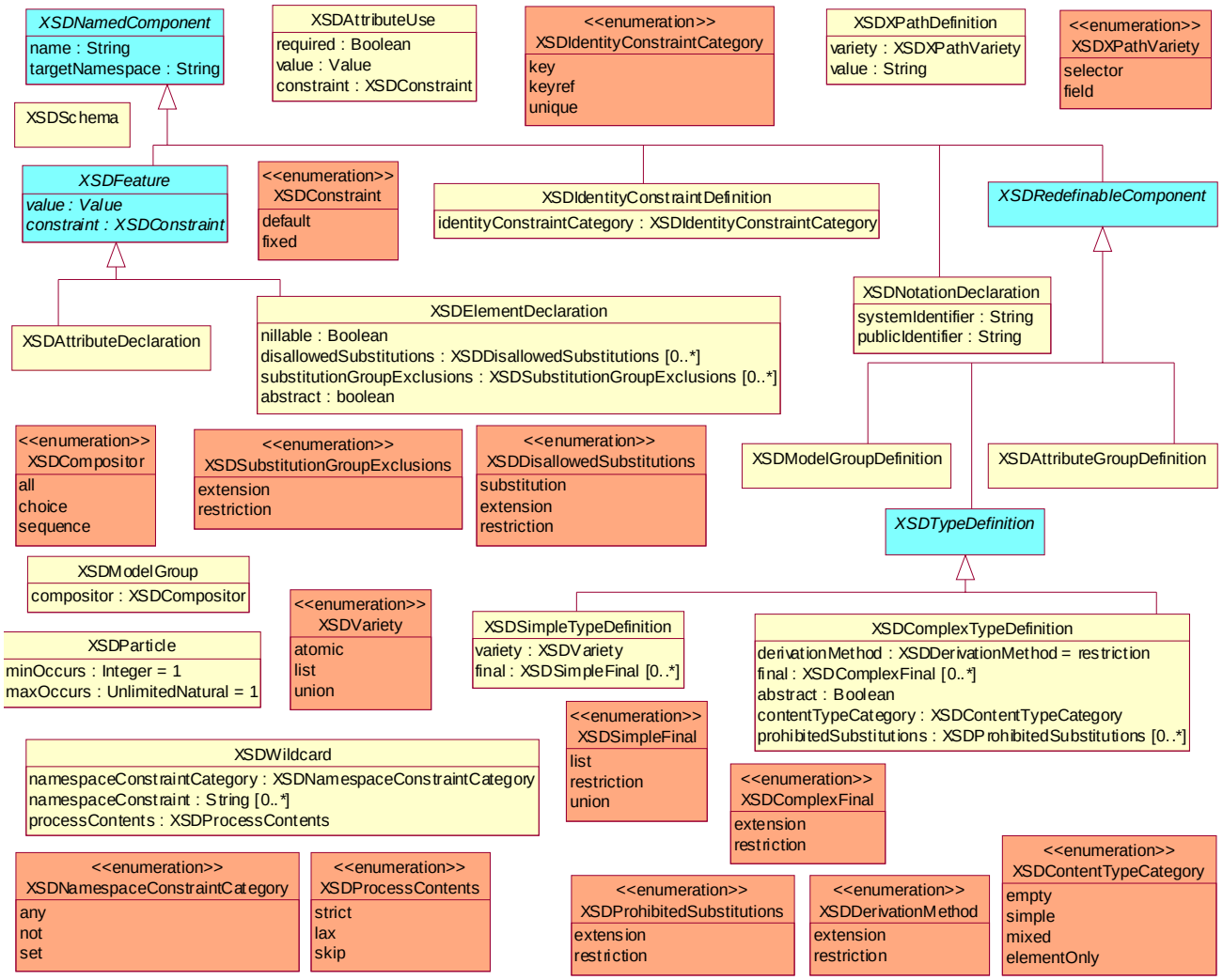


Figure 10.3 - Component Properties

The Component Properties diagram shows the properties of the XML Schema component classes that are associated with the abstract data model. The enumerations that are used as property types are also shown.

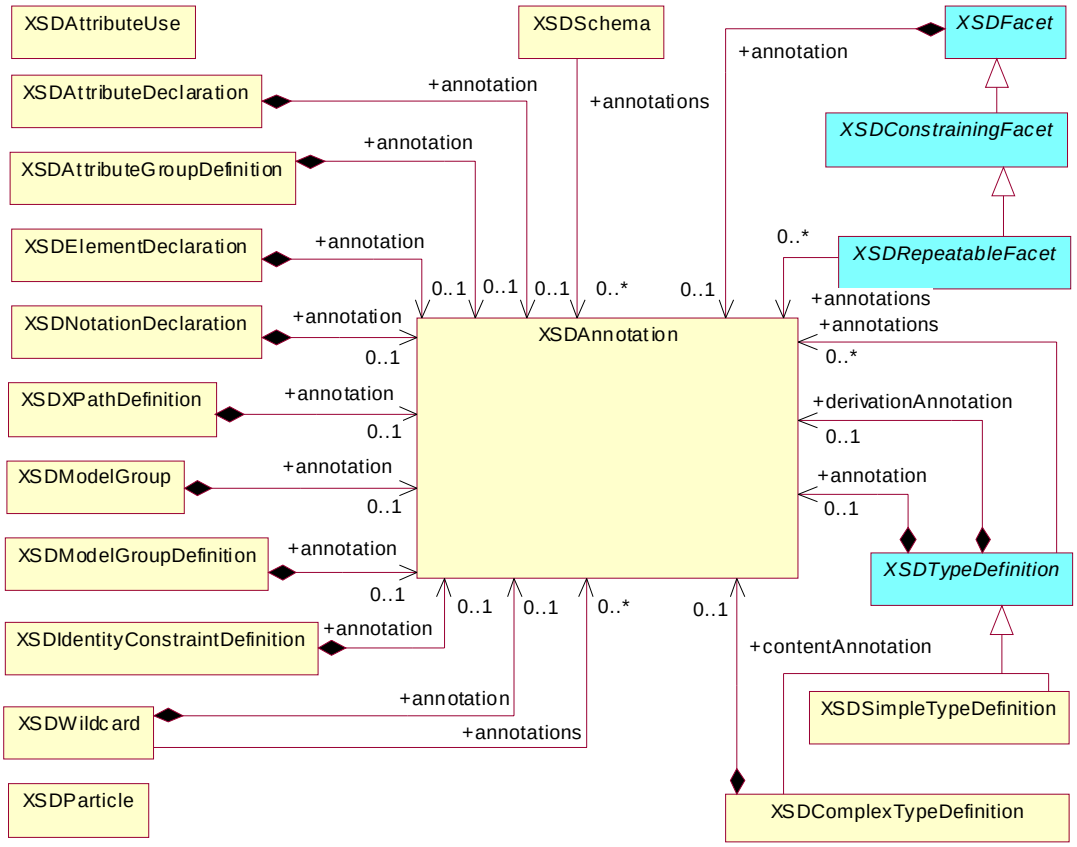


Figure 10.4 - Component Annotations

The annotation schema component provides for human- and machine-targeted annotations of other schema components. The Component Annotations diagram models the structure and usage of the annotation component by other abstract components.

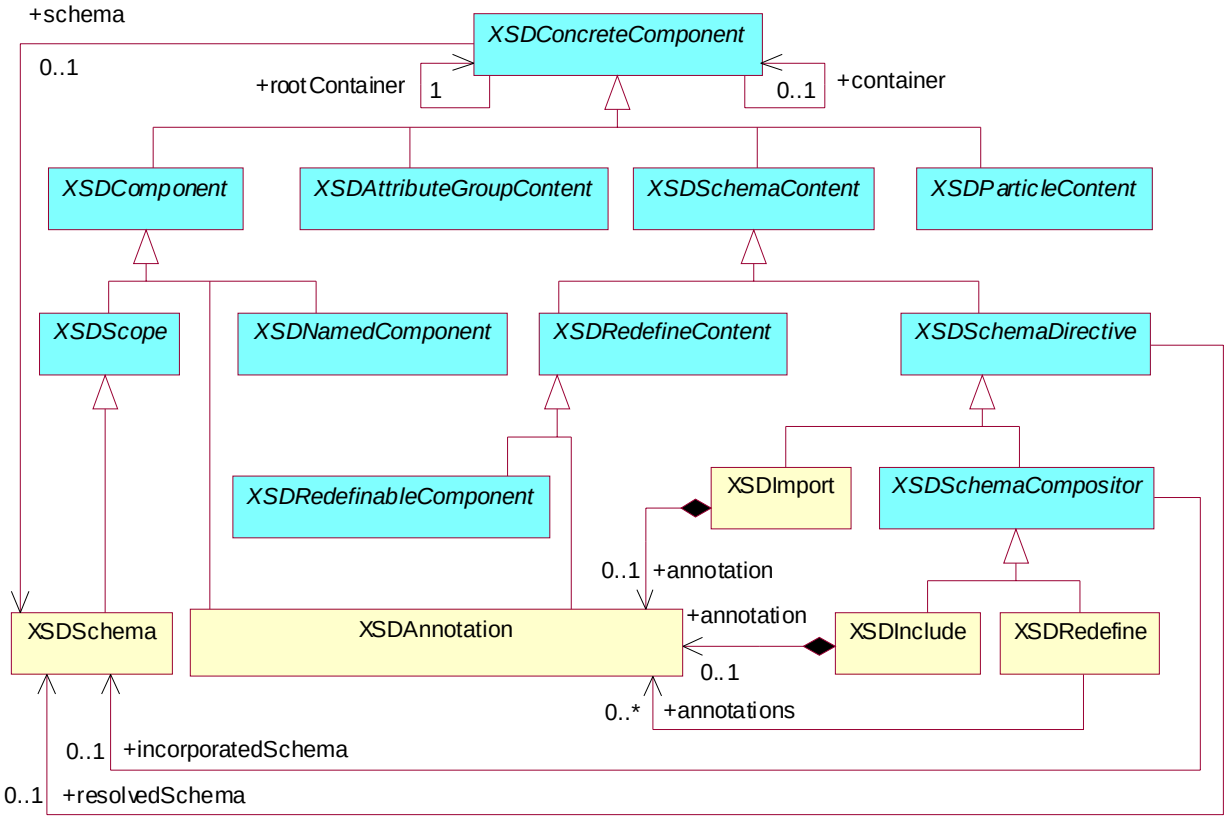


Figure 10.5 - Concrete Components

The Concrete Components diagram shows the additions and extensions to the abstract XML Schema components for representing the concrete syntax. For example, it introduces classes XSDImport (for the import element) and XSDInclude (for the include element).

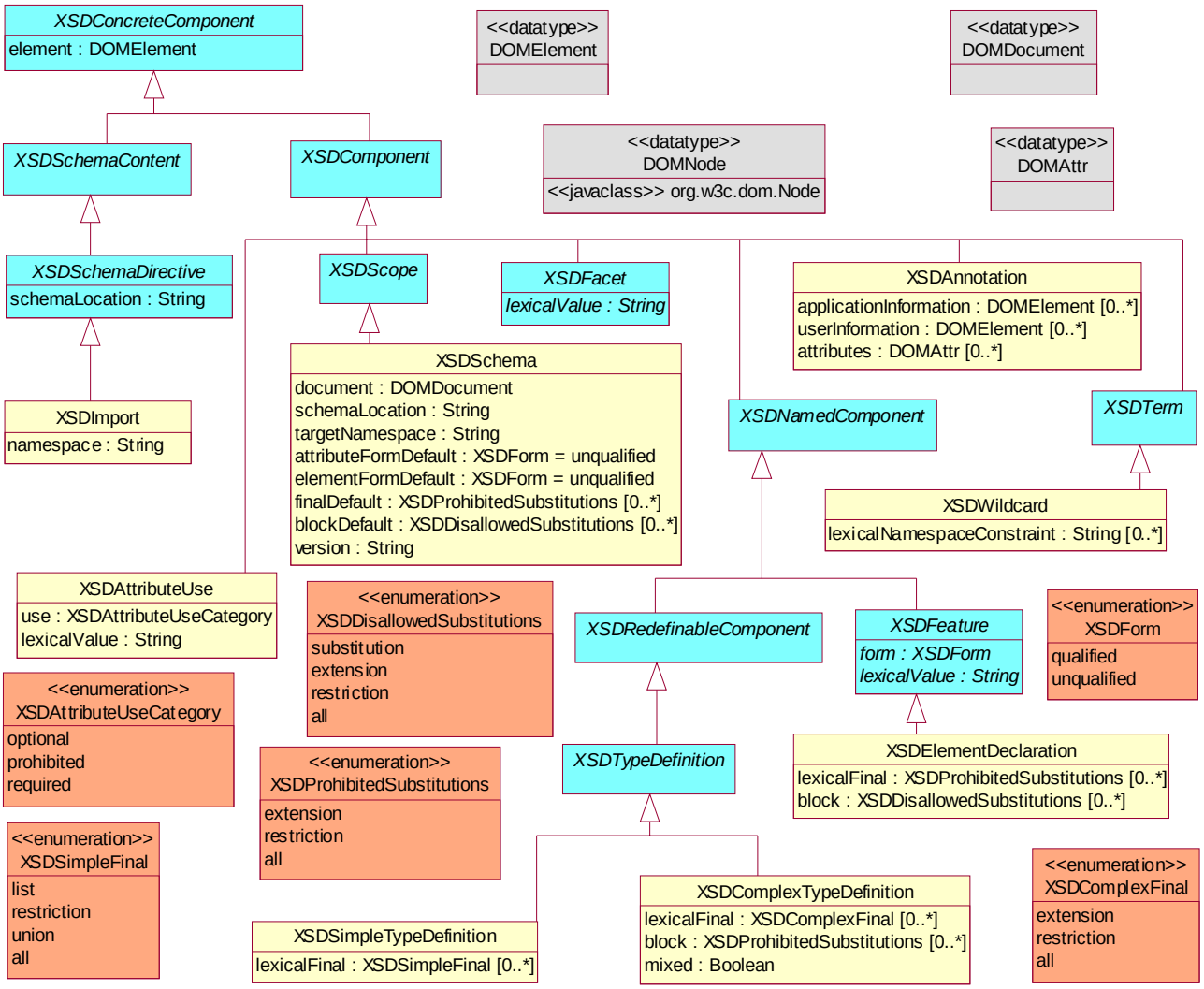


Figure 10.6 - Concrete Properties

The Concrete Properties diagram shows the additional properties required to represent the concrete syntax.

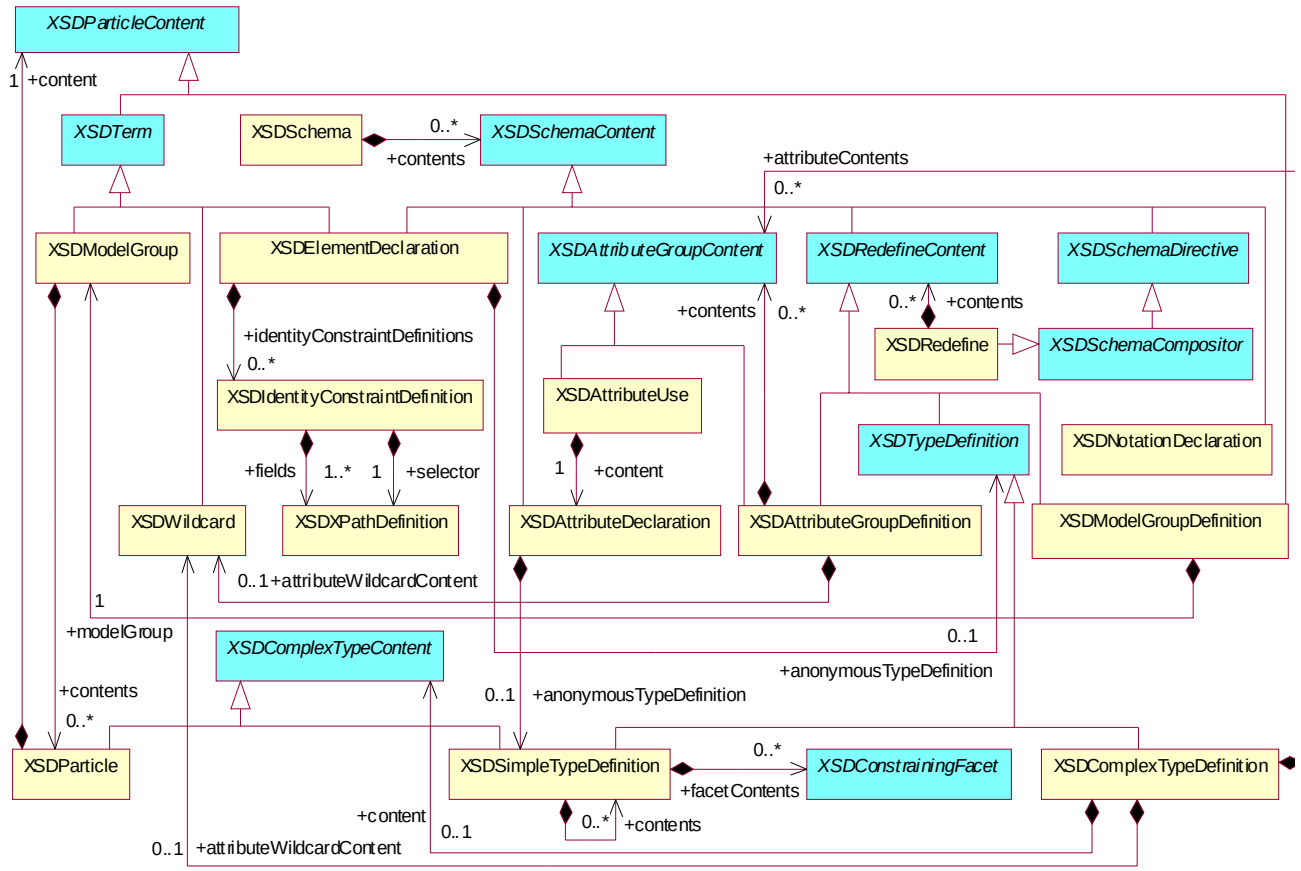


Figure 10.7 - Concrete Containment

The Concrete Containment diagram models the contents of concrete components.

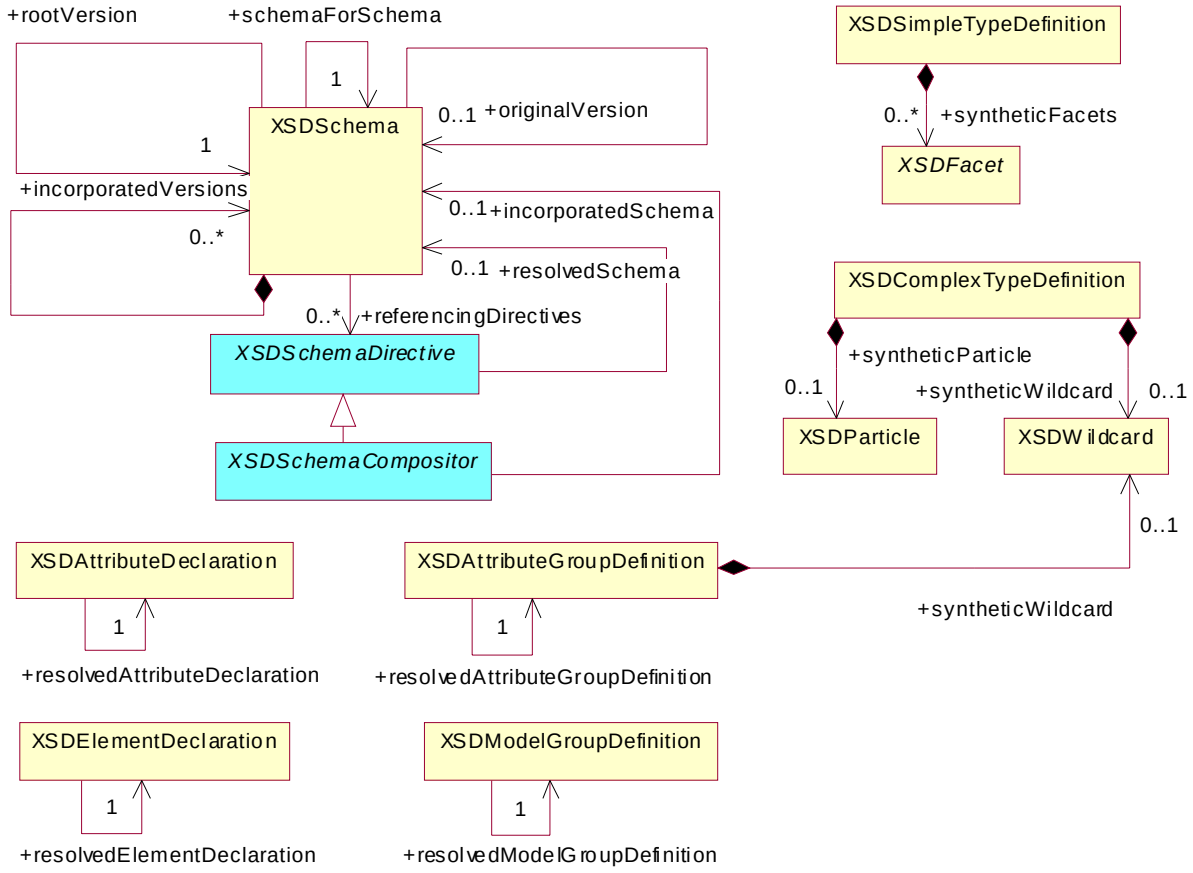


Figure 10.8 - Concrete Schema Composition

The Concrete Schema Composition diagram shows how concrete components resolve to abstract components.

10.2.1 XSDAnnotation

A representation of the model object “Annotation.” Access to the contents of an annotation is provided via their DOM representation.

applicationInformation

This represents the application information infoset property (i.e., a list of appinfo elements).

userInformation

This represents the user information infoset property (i.e., a list of documentation elements).

attributes

This represents the attributes infoset property.

10.2.2 XSDAttributeDeclaration

A representation of the model object ‘Attribute Declaration.’

attributeDeclarationReference

This concrete property is false when the XSDAttributeDeclaration refers to itself as its resolvedAttributeDeclaration.

An infoset feature will never return an instance for which this is true since this is a concrete attribute that is used to represent an attribute declaration with a ref attribute.

annotation

References the XSDAnnotation for this declaration.

anonymousTypeDefinition

This concrete reference represents a simple type definition defined within the body of an attribute element.

typeDefinition

This represents the type definition infoset property.

resolvedAttributeDeclaration

This concrete reference represents the attribute declaration resolved by the ref attribute.

10.2.3 XSDAttributeGroupDefinition

A representation of the model object ‘Attribute Group Definition.’

attributeGroupDefinitionReference

This concrete property is false when the XSDAttributeGroupDefinition refers to itself as its resolvedAttributeGroupDefinition.

annotation

This represents the annotation infoset property.

contents

This concrete reference represents the contents defined within the body of an attributeGroup element.

attributeUses

This represents the attribute uses infoset property. It is computed from the contents.

attributeWildcardContent

This concrete reference represents the attribute wildcard defined within the body of an attributeGroup element.

attributeWildcard

This represents the attribute wildcard infoset property. It is computed from the attribute wildcard content.

resolvedAttributeGroupDefinition

This concrete reference represents the attribute group definition resolved by the ref attribute.

syntheticWildcard

This contains the attribute wildcard infoset property, if the rules require a synthesized component.

10.2.4 XSDAttributeUse

A representation of the model object ‘Attribute Use.’

required

This represents the required infoset property.

value

This represents the value of the value constraint infoset property. It is computed from the lexical value.

constraint

This represents the constraint of the value constraint infoset property.

use

This concrete attribute represents the value of the use attribute.

lexicalValue

This concrete attribute represents the value of the default or fixed attribute.

attributeDeclaration

This represents the attribute infoset property. It is computed from the content.

content

This concrete reference represents the underlying concrete attribute element.

10.2.5 XSDComplexTypeContent

A representation of the model object ‘Complex Type Content.’ It is the contentType of XSDComplexTypeDefinitions.

10.2.6 XSDComplexTypeDefinition

A representation of the model object ‘Complex Type Definition.’

derivationMethod

This represents the derivation method infoset property.

final

This represents the final infoset property. It is computed from the lexical final.

abstract

This represents the abstract infoset property.

contentTypeCategory

This represents the category of the content type infoset property. It is computed from the type of the content and from the setting of mixed.

prohibitedSubstituations

This represents the prohibited substitutions infoset property. It is computed from the block.

lexicalFinal

This concrete attribute represents the value of the final attribute.

block

This concrete attribute represents the value of the block attribute.

mixed

This concrete attribute represents the value of the mixed attribute.

contentAnnotation

This concrete reference represents the annotation content of a complexContent element or a simpleContent element.

baseTypeDefinition

This represents the base type definition infoset property.

content

This concrete reference represents the simple type content or particle content of a complexType element. It will be null, an XSDSimpleTypeDefinition, or an XSDParticle.

contentType

This represents the value of the content type infoset property. It is computed from the content. It will be null, an XSDSimpleTypeDefinition, or an XSDParticle.

attributeUses

This represents the attribute uses infoset property. It is computed from the attribute contents.

attributeContents

This concrete reference represents the attribute contents defined within the body of a complexType element.

attributeWildcard

This represents the attribute wildcard infoset property. It is computed from the attribute wildcard content.

attributeWildcardContent

This concrete reference represents the attribute wildcard defined within the body of an complexType element.

rootTypeDefinition

This walks the base type definitions until it hits the one that has the ur-type definition as its base type definition.

syntheticParticle

This represents the value of the content type infoset property, if the rules require a synthesized particle.

syntheticWildcard

This represents the attribute wildcard infoset property, if the rules require a synthesized wildcard.

10.2.7 XSDComponent

A representation of the model object ‘Component.’ It is the root of the infoset hierarchy.

10.2.8 XSDFeature

A representation of the model object ‘Feature.’ It is used to represent aspects common to ‘Element Declarations’ and ‘Attribute Declaration.’

value

This represents the value of the attribute value constraint or element value constraint infoset property. It is computed from the lexical value.

constraint

This represents the constraint of the attribute value constraint or element value constraint infoset property.

form

This concrete attribute represents the value of the attribute form attribute or the element form attribute. It, along with the attribute form default and element form default of the schema, affects the target namespace of locally scoped features.

lexicalValue

This concrete attribute represents the value of the attribute fixed or default attribute or the element fixed or default attribute.

global

This indicates whether the feature is globally scoped. Its value is false if the feature is declared within a complex type definition, an attribute group definition, or a model group definition.

featureReference

This is the same result as either the ‘Element Reference’ attribute or the ‘Attribute Reference’ attribute.

scope

This represents the attribute scope or element scope infoset property.

resolvedFeature

This is the same result as either the ‘Resolved Element Declaration’ reference or the ‘Resolved Attribute Declaration’ reference.

type

This is the same result as either the element ‘Type Definition’ reference or the attribute ‘Type Definition’ reference.

10.2.9 XSDIdentityConstraintDefinition

A representation of the model object ‘Identity Constraint Definition.’

identityConstraintCategory

This represents the identity constraint category infoset property.

annotation

This represents the annotation infoset property.

referencedKey

This represents the referenced key infoset property.

selector

This represents the selector infoset property.

fields

This represents the fields infoset property. The fields are of type XSDXPathDefinition.

10.2.10 XSDModelGroup

A representation of the model object ‘Model Group.’

compositor

This represents the compositor infoset property.

annotation

This represents the annotation infoset property.

contents

This concrete reference represents the particle contents defined within the body of a sequence, choice, or all element.

particles

This represents the particles infoset property.

10.2.11 XSDNamedComponent

A representation of the model object ‘Named Component.’ It is used to represent aspects common to attribute declarations, attribute group definitions, complex type definitions, element declarations, identity constraint definitions, model groups definitions, notation declarations, and simple type definitions.

name

This represents the value of the attribute declaration name, attribute group definition name, complex type definition name, element declaration name, identity constraint definition name, model group definition name, notation declaration name, or simple type definition name (*) infoset property.

targetNamespace

This represents the value of the attribute declaration target namespace, attribute group definition target namespace, complex type definition target namespace, element declaration target namespace, identity constraint definition target namespace, model group definition target namespace, notation declaration target namespace, or simple type definition target namespace (*) infoset property. It is computed from the target namespace of the schema and should typically not be set directly; in the case of locally scoped features, the value is also affected by the form.

aliasName

This is a constructed name for an anonymous component. In order to make it relatively meaningful, it can be constructed by using the name of the containing component and an indication of the relation to that component, For example, “E_._type” would be the alias name of the anonymous type definition of the element “E” and “LT_._item” would be the alias name of the anonymous item type definition of the list type defintion “LT.”

uRI

This is equivalent to the string

<target namespace>#<name>

where a null target namespace is taken to mean an empty string.

aliasURI

This is equivalent to the string

<target namespace>#<alias name>

where a null target namespace is taken to mean an empty string.

qName

This concrete attribute is this named component's ‘QName.’

10.2.12 XSDSchema

A representation of the model object ‘Schema.’

document

This is the optional DOM document of this schema (i.e., the owner of the element).

schemaLocation

This concrete attribute represents the URI of the resource that contains this schema. It is used to complete any relative schemaLocation URI in an import, include, or redefine.

targetNamespace

This concrete attribute represents the value of the targetNamespace attribute.

attributeFormDefault

This concrete attribute represents the value of the attributeFormDefault attribute.

elementFormDefault

This concrete attribute represents the value of the elementFormDefault attribute.

finalDefault

This concrete attribute represents the value of the finalDefault attribute.

blockDefault

This concrete attribute represents the value of the blockDefault attribute.

version

This concrete attribute represents the value of the version attribute.

contents

This concrete reference represents the contents defined within the body of a schema element.

elementDeclarations

This represents the element declarations infoset property. It is computed from the contents.

attributeDeclarations

This represents the attribute declarations infoset property. It is computed from the contents.

attributeGroupDefinitions

This represents the attribute group definitions infoset property. It is computed from the contents.

typeDefinitions

This represents the type definitions infoset property. It is computed from the contents.

modelGroupDefinitions

This represents the model group definitions infoset property. It is computed from the contents.

identityConstraintDefinitions

This represents the model group definitions infoset property. It is computed from the contents.

notationDeclarations

This represents the notation declarations infoset property. It is computed from the contents.

annotations

This represents the annotations infoset property. It is computed from the contents.

referencingDirectives

This represents the directives that have this schema as their ‘Resolved Schema’ reference or ‘Incorporated Schema’ reference.

rootVersion

This walks the original versions until it hits one that has no original version.

originalVersion

This represents the schema from which an incorporated version originates. The root version has itself as its original version.

incorporatedVersions

This represents those versions of this schema that have been included into a schema with a different namespace or have been otherwise redefined.

schemaForSchema

This represents the ‘schema for schemas.’ It is computed from the schema for schema namespace.

10.2.13 XSDScope

A representation of the model object ‘Scope.’ This is used to represent the types the scope property of XSDFeature (i.e., ‘Schema’ and ‘Complex Type Definition.’

10.2.14 XSDSimpleTypeDefinition

A representation of the model object ‘Simple Type Definition.’ For the properties with names of the form effectiveXxxFacet, effective means that the value of the property is computed based on the direct facets of this type, or, if the facet is not present, is computed recursively from the base type.

variety

This represents the variety infoset property. It is computed based on the presence or absence of an item type or of member types.

final

This represents the final infoset property. It is computed from the lexical final.

lexicalFinal

This concrete attribute list represents the value of the final attribute.

validFacets

This computed attribute list represents the facet name of each type of facet that is valid for this simple type definition.

contents

This concrete reference list represents the anonymous simple type definition content of a restriction, list, or union element.

facetContents

This concrete reference list represents the facet contents of a restriction. There are properties with names of the form XxxFacet that provide direct access to the individual facets.

facets

This represents the facets infoset property. It is computed from the facet contents.

memberTypeDefinitions

This represents the member type definitions infoset property. When constructing a union type, each anonymous member type should be added to both this list and to the contents list. The variety is determined automatically by the presence of member type definitions.

fundamentalFacets

This represents the fundamental facets infoset property. It is a computed property.

baseTypeDefinition

This represents the base type definition infoset property.

primitiveTypeDefinition

This represents the primitive type definition infoset property.

itemTypeDefinition

This represents the item type definition infoset property. When constructing a list type, an anonymous item type should be both set using this method and added to the contents list. The variety is determined automatically by the presence of an item type definition.

rootTypeDefinition

This walks the base type definitions until it hits that one that has the ur-type definition as its base type definition.

minFacet

This represents the XSDMinFacet of the facet contents.

maxFacet

This represents the XSDMaxFacet of the facet contents.

maxInclusiveFacet

This represents the XSDMaxInclusiveFacet of the facet contents.

minInclusiveFacet

This represents the XSDMinInclusiveFacet of the facet contents.

minExclusiveFacet

This represents the XSDMinExclusiveFacet of the facet contents.

maxExclusiveFacet

This represents the XSDMaxExclusiveFacet of the facet contents.

lengthFacet

This represents the XSDLengthFacet of the facet contents.

whiteSpaceFacet

This represents the XSDWhiteSpaceFacet of the facet contents.

enumerationFacets

This represents the XSDEnumerationFacet of the facet contents.

patternFacets

This represents the XSDPatternFacet of the facet contents.

cardinalityFacet

This represents the XSDCardinalityFacet of the fundamental facets.

numericFacet

This represents the XSDNumericFacet of the fundamental facets.

maxLengthFacet

This represents the XSDMaxLengthFacet of the facet contents.

minLengthFacet

This represents the XSDMinLengthFacet of the facet contents.

totalDigitsFacet

This represents the XSDTotalDigitsFacet of the facet contents.

orderedFacet

This represents the XSDOrderedFacet of the fundamental facets.

boundedFacet

This represents the XSDBoundedFacet of the fundamental facets.

effectiveMaxFacet

This represents the XSDMaxFacet of the facets.

effectiveWhiteSpaceFacet

This represents the XSDWhiteSpaceFacet of the facets.

effectiveMaxLengthFacet

This represents the XSDMaxLengthFacet of the facets.

effectiveFractionDigitFacet

This represents the XSDFractionDigitsFacet of the facets.

effectivePatternFacet

This represents the XSDPatternFacet of the facets.

effectiveEnumerationFacet

This represents the XSDEnumerationFacet of the facets.

effectiveTotalDigitsFacet

This represents the XSDTotalDigitsFacet of the facets.

effectiveMinLengthFacet

This represents the XSDMinLengthFacet of the facets.

effectiveLengthFacet

This represents the XSDLengthFacet of the facets.

effectiveMinFacet

This represents the XSDMinLengthFacet of the facets.

syntheticFacets

This represents the facets infoset property, if the rules require a synthesized facet.

10.2.15 XSDTerm

A representation of the model object ‘Term.’ It is used as the type for the XSDParticle term property.

10.2.16 XSDTypeDefinition

A representation of the model object ‘Type Definition.’ It is used to represent aspects common to ‘Simple Type Definitions’ and ‘Complex Type Definitions.’

annotation

This concrete reference represents the direct annotation content of a complexType element or a simpleType element.

derivationAnnotation

This concrete reference represents the direct annotation content of a complex content extension, complex content restriction, simple content extension, simple content restriction, simple type restriction, simple type list, or simple type union element.

annotations

This represents the complex type definition annotation or simple type definition annotation infoset property. It is computed from the annotation, content annotation, derivationAnnotation.

rootType

This walks the base types until it hits that one that has the ur-type definition as its base type.

baseType

This represents the same result as either the simple ‘Base Type Definition’ reference or the complex ‘Base Type Definition’ reference.

simpleType

This represents either the ‘Simple Type Definition’ itself or the complex ‘Content Type’ reference, if it is simple.

complexType

This represents the complex ‘Content Type’ reference, if it is complex (i.e., if it is a ‘Particle’).

10.2.17 XSDWildcard

A representation of the model object ‘Wildcards.’

namespaceConstraintCategory

This represents the category of the namespace constraint infoset property.

namespaceConstraint

This represents the value of the namespace constraint infoset property. It is computed from the lexical namespace constraint and should typically not be modified directly.

processContents

This represents the process contents infoset property.

lexicalNamespaceConstraint

This concrete attribute represents the value of the any namespace or anyAttribute namespace attribute.

annotation

This concrete reference represents the annotation content of an any or anyAttribute element.

annotations

This represents the annotation infoset property. It is computed from the annotation.

10.2.18 XSDXPathDefinition

A representation of the model object ‘XPath Definition.’ It represents a field or selector of an Identity-constraint Definition. It defines a restricted XPath. It is used to represent the types of object returned by the 'Fields' reference list and the ‘Selector’ reference.

variety

This attribute represents whether this is a field or a selector.

value

This concrete attribute represents the value of the selector xpath or field xpath attribute.

annotation

This concrete reference represents the annotation contents defined within the body of a field or selector element.

10.3 XML Schema Datatypes

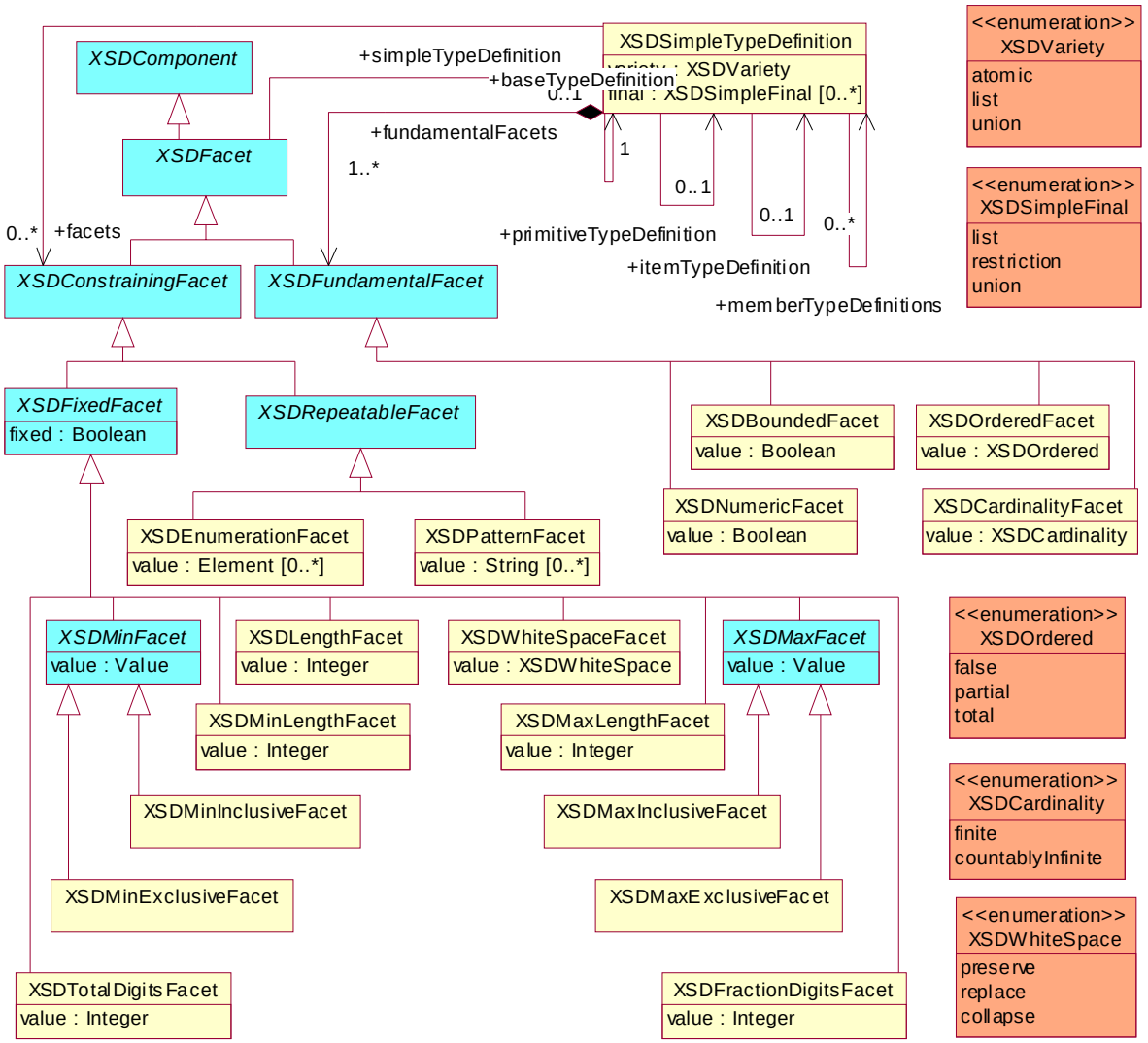


Figure 10.9 - Component Hierarchy, Relations, and Properties (Part 2: Datatypes)

The Component Hierarchy, Relations, and Properties (Part 2: Datatypes) diagram represents the abstract XML Schema components as defined in XML Schema Part 2: Datatypes. Each datatype has a value space, which is the set of values for that datatype. A facet is a single defining aspect of a ·value space·. Generally speaking, each facet characterizes a ·value space· along independent axes or dimensions. The facets of a datatype serve to distinguish those aspects of one datatype that differ from other datatypes.

Facets are of two types: fundamental facets that define the datatype, and constraining facets constrain the permitted values of a datatype. For example, the XML Schema string datatype has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

In contrast, the boolean dataype has these constraining facets:

- pattern
- whiteSpace

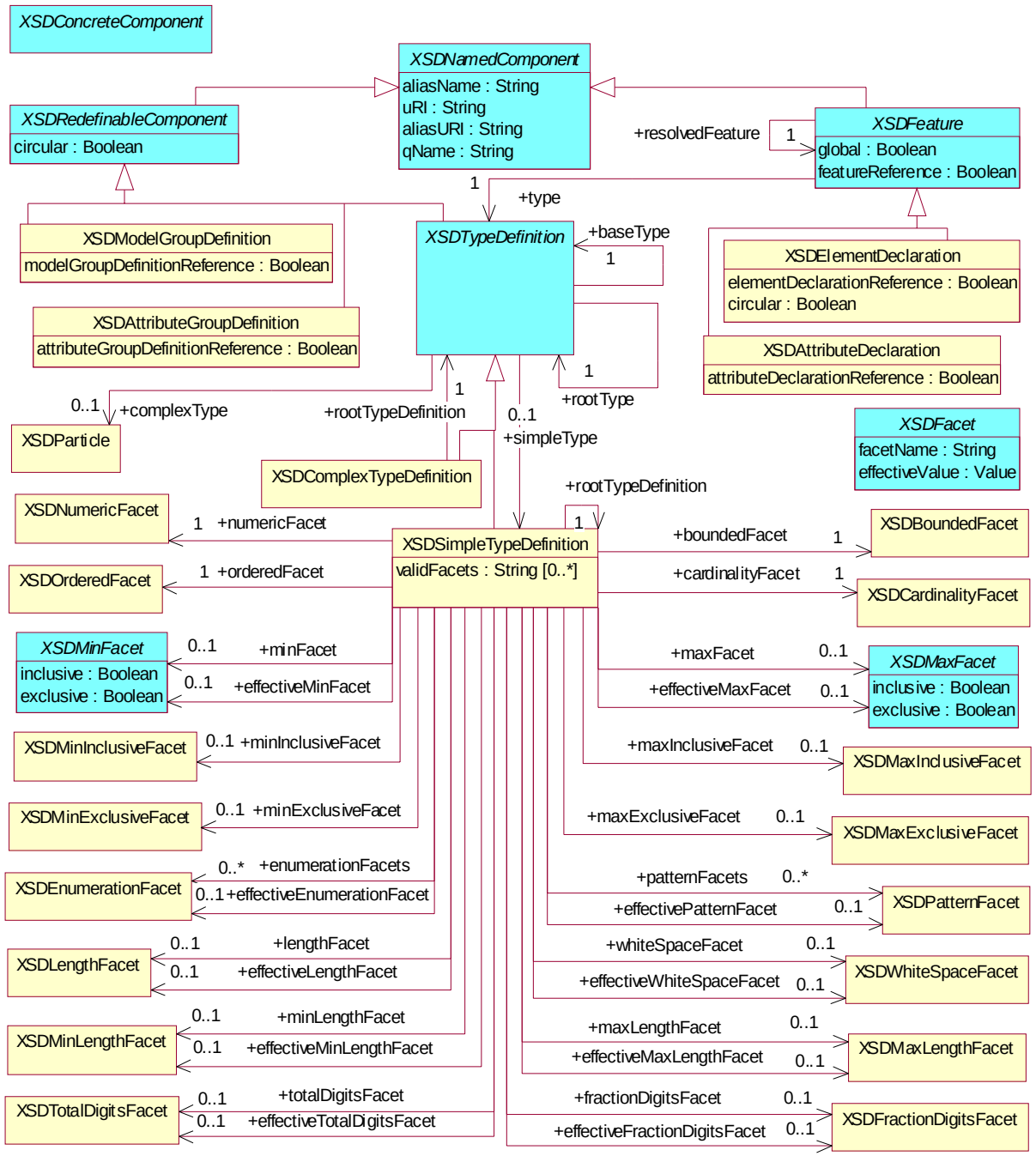


Figure 10.10 - Supplemental

The Supplemental diagram primarily models the relationships between type definitions and facets.

10.3.1 XSDBoundedFacet

A representation of the model object ‘Bounded Facet.’

value

This represents the value infoset property. It is a computed property.

10.3.2 XSDCardinalityFacet

A representation of the model object ‘Cardinality Facet.’

value

This represents the value infoset property. It is a computed property.

10.3.3 XSDConstrainingFacet

A representation of the model object ‘Constraining Facet.’

10.3.4 XSDEnumerationFacet

A representation of the model object ‘Enumeration Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.5 XSDFixedFacet

A representation of the model object ‘Fixed Facet.’

fixed

This represents the fractionDigitsFacet fixed, lengthFacet fixed, maxExclusiveFacet fixed, maxInclusiveFacet fixed, maxLengthFacet fixed, minExclusiveFacet fixed, minInclusiveFacet fixed, minLengthFacet fixed, totalDigitsFacet fixed, whiteSpaceFacet fixed infoset property.

10.3.6 XSDFundamentalFacet

A representation of the model object ‘Fundamental Facet.’

10.3.7 XSDFacet

A representation of the model object ‘Facet.’

lexicalValue

This concrete attribute represents the value of the value attribute of the facet element.

facetName

This concrete attribute represents the name of this type of facet.

effectiveValue

This represents a generic version of the value infoset property of this facet.

annotation

This represents the annotation infoset property; each type of facet has an annotation.

simpleTypeDefinition

This represents the containing simple type definition of the facet.

10.3.8 XSDFractionDigitsFacet

A representation of the model object ‘Fraction Digits Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.9 XSDLengthFacet

A representation of the model object ‘Length Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.10 XSDMaxExclusiveFacet

A representation of the model object ‘Max Exclusive Facet.’

10.3.11 XSDMaxFacet

A representation of the model object ‘Max Facet.’ It represents aspects common to ‘Max Exclusive Facet’ and ‘Max Inclusive Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

inclusive

The value is true if this is an XSDMaxInclusiveFacet.

exclusive

The value is true if this is an XSDMaxExclusiveFacet.

10.3.12 XSDMaxInclusiveFacet

A representation of the model object ‘Max Inclusive Facet.’

10.3.13 XSDMaxLengthFacet

A representation of the model object ‘Max Length Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.14 XSDMinFacet

A representation of the model object ‘Min Facet.’ It represents aspects common to ‘Min Exclusive Facet’ and ‘Min Inclusive Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

inclusive

The value is true if this is an XSDMinInclusiveFacet.

exclusive

The value is true if this is an XSDMinExclusiveFacet.

10.3.15 XSDMinExclusiveFacet

A representation of the model object ‘Min Exclusive Facet.’

10.3.16 XSDMinInclusiveFacet

A representation of the model object 'Min Inclusive Facet'.

10.3.17 XSDMinLengthFacet

A representation of the model object ‘Min Length Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.18 XSDNumericFacet

A representation of the model object ‘Numeric Facet.’

value

This represents the value infoset property. It is a computed property.

10.3.19 XSDOrderedFacet

A representation of the model object ‘Ordered Facet.’

value

This represents the value infoset property. It is a computed property.

10.3.20 XSDPatternFacet

A representation of the model object ‘Pattern Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute. *value* is a multi-valued property, in which each value is a String representing a pattern. The overall effect of the patterns is the logical intersection.

10.3.21 XSDRepeatableFacet

A representation of the model object ‘Repeatable Facet.’

Both pattern and enumeration facets may be repeated in the concrete syntax and yet they are merged into a single component in the infoset model. As a result, instances of these two facets are synthesized by the effectivePatternFacet and effectiveEnumerationFacet properties of XSDSimpleTypeDefinition.

annotations

This represents the enumeration annotation, or pattern annotation infoset property. It is computed from the concrete annotation content.

10.3.22 XSDTotalDigitsFacet

A representation of the model object ‘Total Digits Facet.’

value

This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.3.23 XSDWhiteSpaceFacet

A representation of the model object ‘White Space Facet.’

value

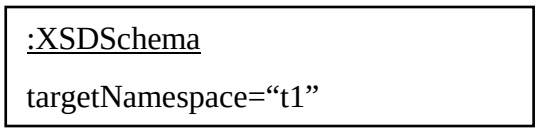
This represents the value infoset property. It is computed from the ‘Lexical Value’ attribute.

10.4 Example

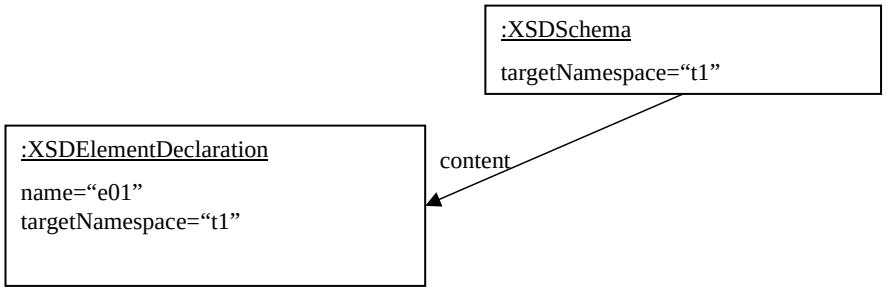
This section shows how a simple XML Schema is represented as an instance of the XML Infoset model. The schema in this example is:

```
<xs:schema targetNamespace=t1 xmlns:xs=http://www.w3.org/2001/XMLSchema>
  <xs:element name="e01" type="ct01"/>
  <xs:complexType name="ct01">
    <xs:choice>
      <xs:element name="inline" type="xs:string" minOccurs="2" maxOccurs="3"/>
      <xs:any namespace="##other"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

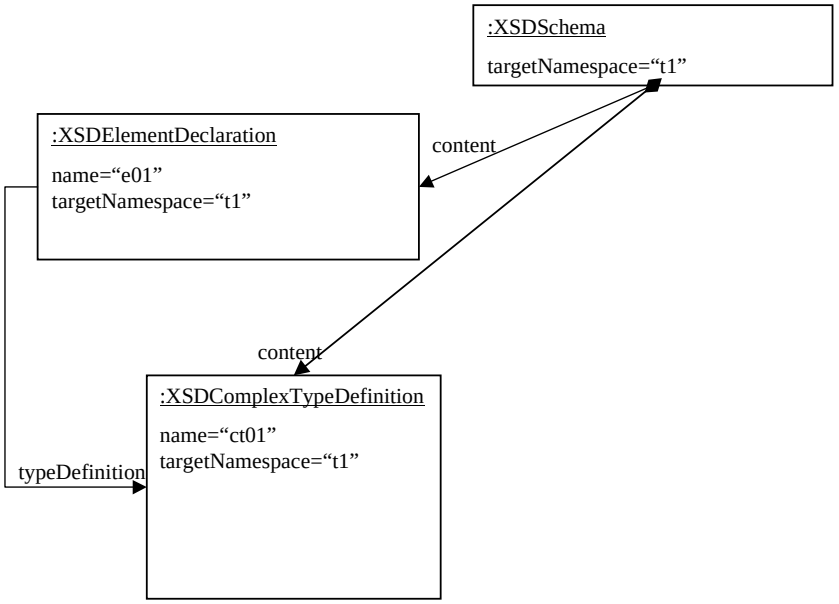
The information in the **xs:schema** tag is represented by an :XSDSchema.



The information in the xs:element tag adds in an XDSElementDeclaration (excluding for the moment the reference to its type).

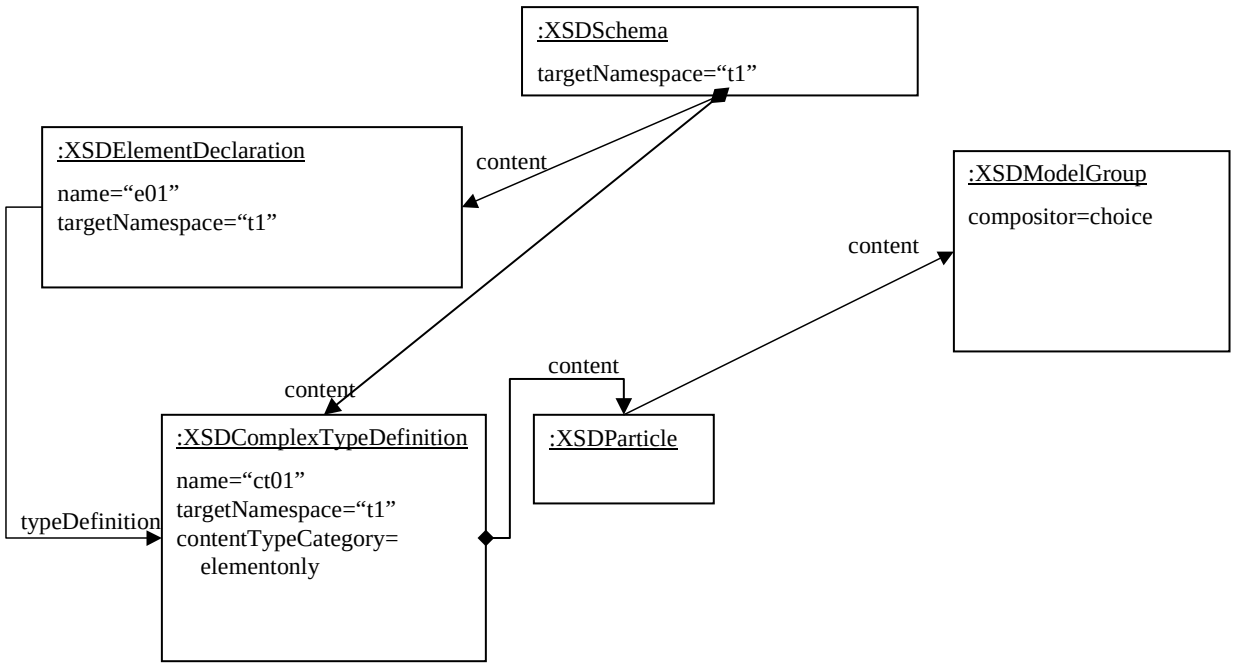


The information in the **xs:complexType** tag adds an :XSDComplexTypeDefinition. Since this is the type for element e01, the :XSDElementDeclaration references the :XSDComplexTypeDefinition through the typeDefinition property.

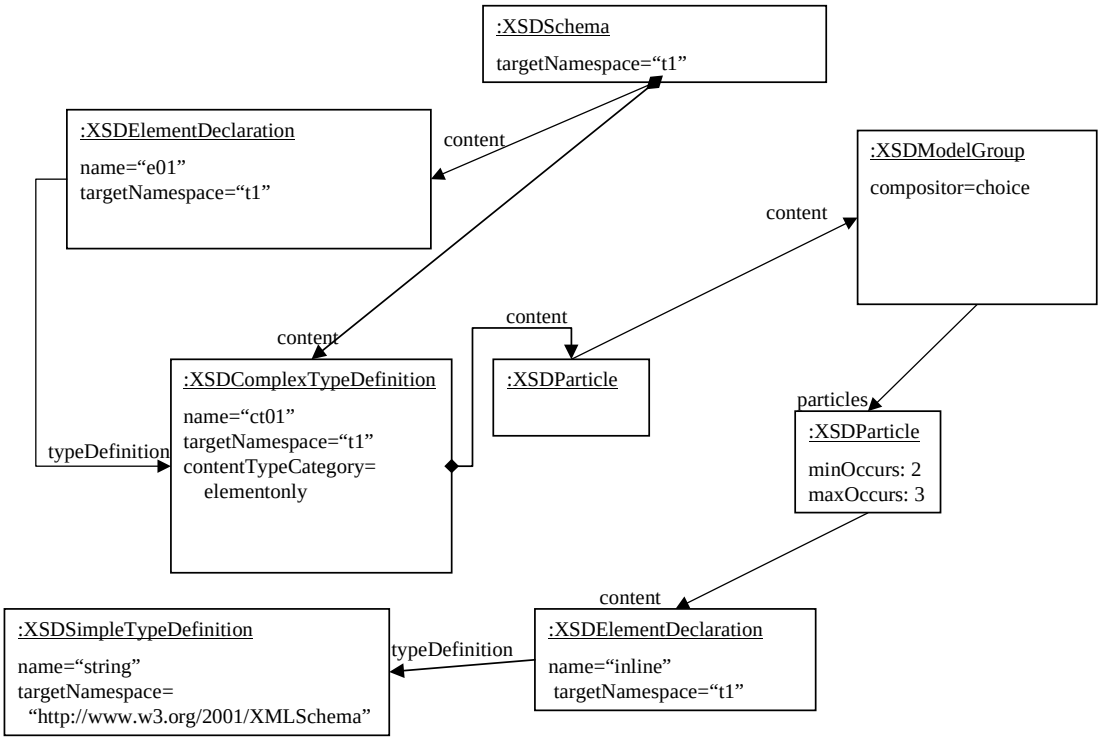


Complex type ct01contains an **xs:choice** tag. This means that the content type of the complex type definition is a pair consisting of

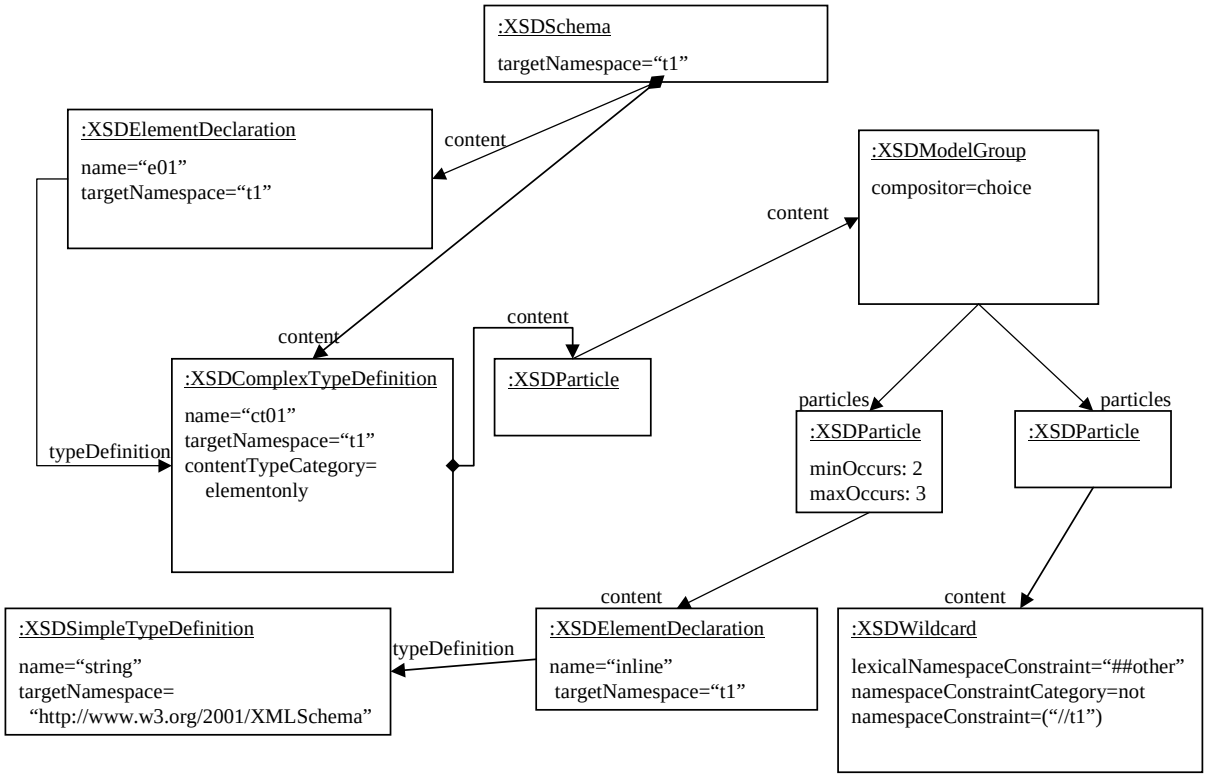
- *elementonly.* This is represented by the contentTypeCategory property in the XSDComplexTypeDefinition.
- The particle corresponding to the <choice>. Particles corresponding to a <choice> have terms that are model groups. This particle is represented in the instance diagram by an XSDParticle that references an XSDModelGroup whose compositor has the value "choice."



The first element within the choice is an **xs:element** tag named "inline." This means that the model group has a particle whose term is an element declaration. The **minOccurs** and **maxOccurs** attributes in the **xs:element** tag are represented by the XSDParticle's minOccurs and maxOccurs properties. The **type** attribute in the tag is represented by an XSDSimpleTypeDefinition for the XML Schema string type. This is shown in the instance diagram below.



The second element within the choice is an **xs:any** tag. This means that the model group has another particle, whose term is a wildcard. This is show in the instance diagram by an XSDInstance and an XSDWildcard, completing the representation of the schema:


:XSDSchema
targetNamespace="t1"
:XSDElementDeclaration
name="e01"
targetNamespace="t1"
content
:XSDModelGroup
compositor=choice
content
content
content
:XSDComplexTypeDefinition
name="ct01"
targetNamespace="t1"
contentTypeCategory=
elementonly
:XSDParticle
particles
particles
typeDefinition
:XSDParticle
minOccurs: 2
maxOccurs: 3
:XSDParticle
content
content
:XSDSimpleTypeDefinition
name="string"
targetNamespace=
"http://www.w3.org/2001/XMLSchema"
typeDefinition
:XSDElementDeclaration
name="inline"
targetNamespace="t1"
:XSDWildcard
lexicalNamespaceConstraint="##other"
namespaceConstraintCategory=not
namespaceConstraint=("//t1")

Annex A - References

[XML]	XML, a technical recommendation standard of the W3C. http://www.w3.org/TR/REC-xm
[XMLSchema]	XML Schemas, a proposed recommendation of the W3C. Primer: http://www.w3.org/TR/xmlschema-0/, Structured types: http://www.w3.org/TR/xmlschema-1/ Data types: http://www.w3.org/TR/xmlschema-2/
[NAMESP]	Namespaces, a technical recommendation of the W3C. http://www.w3.org/TR/REC-xml-names
[XLINK]	XLinks, a working draft of the W3C. http://www.w3.org/TR/WD-xlink and http://www.w3.org/TR/NOTE-xlink-principles
[XPath]	XPointer, technical recommendation of the W3C. http://www.w3.org/TR/xpath
[UML]	UML 2.0, an in progress standard of the OMG. More specifically, this refers to the UML 2.0 Infrastructure submission. See http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Infrastructure_RFP.html.
[MOF]	MOF 2.0, an in progress standard of the OMG. More specifically, this refers to the MOF 2.0 Core submission. See http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Core_RFP.html.
[XMI]	XMI 2.0, an adopted standard of the OMG. http://www.omg.org

The following is the Open Group DCE standard on UUIDs.

[UUID]	CAE Specification DCE 1.1: Remote Procedure Call Document Number: C706 http://www.opengroup.org/onlinepubs/9629399/toc.htm http://www.opengroup.org/onlinepubs/9629399/apdxa.htm (Definition/creation of UUIDs)