XML Metadata Interchange (XMI)

Proposal to the OMG OA&DTF RFP 3:
Stream-based Model Interchange Format (SMIF)

Joint Submission

Cooperative Research Centre for Distributed Systems Technology (DSTC)

International Business Machines Corporation

Oracle Corporation

Platinum Technology, Inc.

Unisys Corporation

Supported by:

Cayenne Software
Genesis Development
Inline Software
Rational Software Corporation
Select Software
Sprint Communications Company
Sybase, Inc.

OMG Document ad/98-07-01
July 6, 1998

Table of Contents

Preface 1

1.1 Cosubmitting Companies and Supporters

The following companies are pleased to co-submit the XML Metadata Interchange specification (hereafter referred to as XMI) in reponse to the Object Analysis & Design Task Force RFP3 - Stream based Model Interchange Format (SMIF):

- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- International Business Machines Corporation
- Oracle Corporation
- Platinum Technologies, Inc.
- Unisys Corporation

The following companies are pleased to support the XMI specification:

- Cayenne Software
- Genesis Development
- Inline Software
- Rational Software Corporation
- Select Software Tools
- Sprint Communications Company
- Sybase, Inc.

1.2 Introduction

The main purpose of XMI is to enable easy interchange of metadata between modeling tools (based on the OMG UML) and between tools and metadata repositories (OMG MOF based) in distributed heteroogeneous environments. XMI integrates three key industry standards:

- XML - eXtensible Markup Language, a W3C standard
- UML - Unified Modeling Language, an OMG modeling standard
- MOF - Meta Object Facility and OMG modeling and metadata repository standard

The integration of these three standards into XMI marries the best of OMG and W3C metadata and modeling technologies allowing developers of distributed systems share object models and other meta data over the Internet.

SMIF (XMI) and OMG Repository Architecture

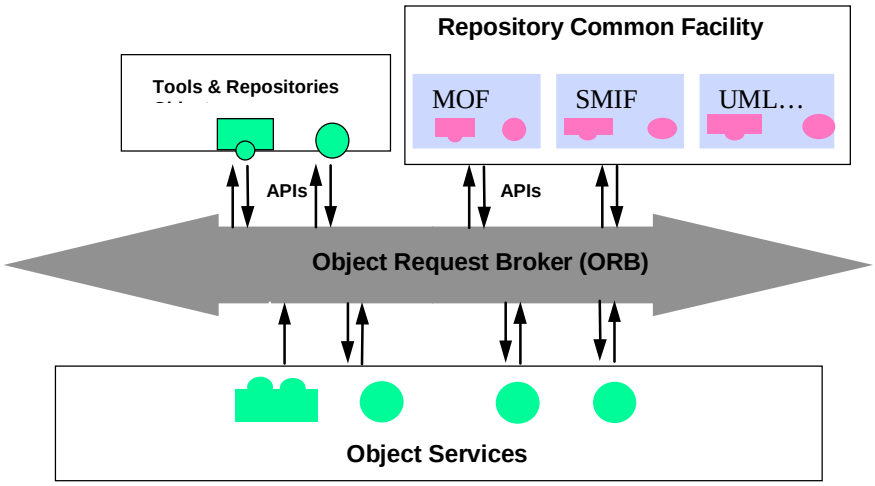

Figure 1-1 The OMG Repository Architecture and the SMIF

XMI, together with MOF and UML form the core of the OMG repository architecture that integrates object oriented modeling and design tools between each other and with a MOF based extensible repository framework as illustrated in Figure 1-1. This architecture allows tools to share metadata programmatically using CORBA interfaces specified in the MOF and UML standards or by using XML based stream (or file) containing MOF and UML compliant modeling specifications. This allows the widest degree of latitude for tool, repository and object framework developers and lowers the barrier to entry for implementing OMG metadata standards. The OMG OA&DTF members have already begun extending this architecture to managing data warehousing metadata in the Common Warehouse Metadata Interchange (CWMI) initiative.

This submission mainly consists of:

- A set of XML Document Type Definition (DTD) production rules for transforming MOF based metamodels to XML DTDs

- A set of XML Document production rules for encoding and transfering MOF based metadata
- Design principles for XMI based DTDs
- Concrete DTDs for UML and MOF

This submission defines these standards and provides proof of concept that covers key aspects of the XMI. The submission represents the integration of work currently underway by the co-submitters and supporters in the areas of object repositories, object modeling tools, web authoring technology and business object management in distributed object environments. The co-submitters intend to commercialize the XMI technology within the guidelines of the OMG.

Adoption of this submission would enhance meta data management and meta data interoperability in distributed object environments in general and in distributed development environments in particular. While the initial RFP (XMI) addresses stream based meta data interoperability in object analysis and design domain, the submitters anticipate the XMI (in part because it is MOF based) to be rich enough to support additional domains. Examples include metamodels that cover the application development life cycle as well as additional domains such as data warehouse management and business object management. OMG is expected to issue new RFPs to cover these additional domains. The submitters expect this version of the XMI to evolve in the future to address new requirements.

The adoption of the UML and MOF specifications in 1997 was a key step forward for the OMG and the industry in terms of achieving consensus on modeling technology and repositories after years of failed attempts to unify both areas. The adoption of XMI is expected to address the plethora of proprietary meta data interchange formats and minimally succesful attempts of the Meta Data Coaltion (Meta Data Interchange Specification) and Case Data Interchange Format (EIA CDIF) because of widespread adoption of W3C (XML) and OMG (UML, MOF) standards as well as industry pressures on integrated and interoperable development environments composed of tools from multiple vendors. XMI is also expected to ease the integration of CORBA, Java, and COM based development environments which are both evolving to similar extensible repository architectures based on standard information models, repository interfaces and interchange formats.

1.3 Submission contact points

Please send comments on this submission to xmi-feedback@omg.org.

All questions about this submission should be directed to:

Sridhar Iyengar
Unisys Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
Phone: +1 949 380 5692
Email: sridhar.iyengar2@unisys.com

Stephen A. Brodsky, Ph.D.
International Business Machines Corporation
555 Bailey Ave., L19/F320
San Jose, CA 95141
Phone: +1 408 463 5659
Email: SBrodsky@us.ibm.com

Contact information for the other co-submitting companies is:

Dr. Kerry Raymond
CRC for Distributed Systems Technology
University of Queensland 4072 Australia
Phone: +61 73365 4310
Email: kerry@dstc.edu.au

Dr. Stephen Crawley
CRC for Distributed Systems Technology
Email: crawley@dstc.edu.au

Simon McBride
CRC for Distributed Systems Technology
Email: sjm@piglet.dstc.edu.au

Tim Grose
International Business Machines Corporation
Email: TGrose@us.ibm.com

Peter Thomas
Oracle Corporation
Oracle Parkway
Thames Valley Park
Reading
Berkshire
RG6 1RA
Phone: +44 118 924 5132
Email: pthomas@uk.oracle.com

John Cramer
Platinum Technology, Inc.
8045 Leesburg Pike, Suite 300
Vienna, VA 22182
Phone: +1 703 848 3288
Email: cramer@platinum.com

Dr. Gene Mutschler
Unisys Corporation
Email: Gene.Mutschler@unisys.com

GK Khalsa
Unisys Corporation
khalsa@objectrad.com

Contact information for the supporting companies is:

Naresh Bhatia
Cayenne Software
Email: Bhatian@cayennesoft.com

David Frankel
Genesis Development
Email: DFrankel@gendev.com

Bill Dudney
Inline Software
Email: BDudney@inline-software.com

Jack Greenfield
Inline Software
Email: Jack@inline-software.com

Magnus Christerson
Rational Software Corporation
Email: Christerson@rational.com

Lydia Patterson
Select Software
Email: Lydiap@selectst.com

Abdul Akram
Sprint Communications Company
Email: Abdul.Akram@mail.sprint.com

Andrew Eisenberg
Sybase, Inc.
Email: Andrewe@sybase.com

The co-submitters and supporters of the XMI submission appreciate the contributions of the following individuals during the SMIF submission process:

Don Baisley, Robert Blum, Dan Chang, Keith Duddy, Johannes Ernst, Alexander Glebov, Craig Hayman, Kurt Kirkey, Woody Pidcock, Ashit Sawhney, and Dave Stringer.

1.4 Status of this Document

This document is an initial submission. A revised submission has been scheduled for October 20, 1998, as described in the RFP. Refer to the OMG web site, http://www.omg.org for the latest schedule.

1.5 Guide to the Submission

This proposal is presented in the following sections:

Section 1 Overview

Introduces the submission and provides the context for the XMI technology within the OMG architecture

Section 2 Proof of Concept

Describes proof of concept efforts and results, in demonstration of the proposal's technical viability.

Section 3 Response to RFP Requirements

Identifies the specific RFP requirements and this proposal's response to each requirement.

Section 4 Design Rationale

Describes the design goals and rationale of this proposal, giving an overview of the proposed solution and insight into the motivation and design forces.

Section 5 Usage Scenarios

Describes how the XMI is expected to be used by customers and tool vendors

Section 6 DTD Design Principles

Provides a discussion of Document Type Definition (DTD) usage, generation and standard parts.

Section 7 DTD Generation Rules

Specifies the production rules for DTDs, as part of the encoding of MOF based metamodels into the proposed format.

Section 8 XML Production Principles

Discusses the manner in which a model is represented as an XML document.

Section 9 XML Document Production

Specifies the production rules for encoding any model, with a MOF- defined meta-model, in the proposed format.

Section 10 Compatibility with other standards

This section discusses how the XMI specification is related to other industry standards

Section 11 Conformance Issues

This section discusses conformance - mandatory and optional; compliance points in the XMI specification.

References

Lists the references used in this specification

Glossary

This section describes a glossary of terms relevant to the MOF, UML and XMI specifications.

Appendix A

The UML 1.1 DTD

Appendix B

The MOF 1.1 DTD

Appendix C

Example encodings of models

1.6 Conventions

IDL appears using this font.

XML appears using this font.

```
Object Constraint Language (OCL) appears using this font.
```

Caution – Cautionary information appears with this prefix, framing, and in this font.

Note – Items of note appear with this prefix, framing, and in this font

Please note that any change bars have no semantic meaning. They show the places that errata were discovered since the last submission. They are present for the convenience of readers and submitters so that the final edits can be identified.

Proof of Concept 2

2.1 Copyright Waiver

In the event that this specification is adopted by OMG, the submitters grant to the OMG, a non-exclusive, royalty-free, paid-up, worldwide license to copy and distribute this specification document and to modify the document and distribute copies of the modified version. For more detailed information, see the disclaimer on the inside of the cover page of this submission.

2.2 Proof of Concept

XMI cosubmitters and supporters have extensive experience in the areas of meta data repositories, modeling tools, CORBA and the related problems of interchage of meta data across tools in distributed heterogeneous environments. Relevant portions of their experience is highlighted below :

- Unisys, IBM, Oracle and Platinum are experienced in the implementation of commercial meta data repositories that have enabled meta data interchange using APIs (proprietary, OMG MOF based, COM based etc.) and file based interchange formats (proprietary, CDIF, MDIS etc.). These meta data repository vendors have already begun prototyping the integration of XMI with their respective products. Most of the leading repository vendors have announced plans to support XMI.
- Platinum, Rational and Select are leading modeling tool vendors implementing UML and are committing to using XMI as the interchange format. IBM and Unisys have already prototyped round trip engineering of UML models using the XMI UML DTD for the Rational Rose and Select Enterprise products. This prototype includes the exporting a model from Select Enterprise and importing it into Rational Rose proving interoperability between tools produced by different vendors.
- Unisys has prototyped and is implementing IDL generation from a MOF and is extending this work to generate both XML DTDs and XML based streams from a MOF based repository server.

- IBM has prototyped and is implementing generating both XML DTDs and XML based streams from their repository server. IBM has also prototyped XMI stream differencing.
- DSTC has developed prototypes for a MOF repository, along with meta-model compilers, IDL generators and server generators. These are currently being used to prototype generators for XMI interchange software that can emit an XML steram for a model held in a MOF-based repository, and can populate a MOF-based repository from an XML stream. The interchange software is being trialed with a wide range of realistic meta-models and test cases.
- The XMI work is based on two key available meta data standards - OMG MOF and W3C XML - that are being implemented by several vendors. The first major use of XMI will be for the interchange of UML models based on the OMG standard UML metamodel
- IBM and Microsoft have implemented XML parsers which were used in our proof of concepts.

The submitters expect to demonstrate some of these proof of concepts in upcoming OMG meetings.

Response to RFP Requirements 3

3.1 Mandatory Requirements

3.1.1 Required Meta-metamodel

Proposals shall use the MOF as its meta-metamodel.

The XMI proposal uses MOF as its meta-metamodel.

Any model or model fragment that has a MOF compliant metamodel can be exchanged using XMI, as can the metamodels themselves. The XMI proposal specifies how any MOF compliant meta-model maps to XML DTDs, and how a corresponding model or model fragment maps to XML.

3.1.2 Syntax and Encoding

Proposals shall provide a complete specification of the syntax and encoding needed to export/import models and meta-model extensions included in-line as part of the transfer stream. This syntax and encoding shall have an unambiguous identification to support evolution of this technology.

The XMI specification provides a complete specification for syntax and encoding needed to export and import meta-models and models including extensions. Evolution of the XMI technology is also specified. Please refer to Section 6, *XMI DTD Design Principles* on page 35 and Section 8, *XML Generation Principles* on page 93 for details on syntax and encoding. Example DTDs for XMI encoding of UML models and MOF metamodels are provided in the Appendices.

Evolution of technology is supported using the following specific mechanisms:

1. The XML header identifies the XML version - currently 1.0 as adopted by W3C.
2. The XMI header identifies the XMI specification version number - currently 1.0.

3. The XMI header identifies the MOF metamodel(s) for the model information encoded in an XMI transfer stream, giving metamodel names, versions and links to their definitions.

4. The XMI.extensions element allows XMI to handle extensions to a metamodel; for example to represent the layout of a model's diagram. Extension meta-data can be transmitted inline as part of the transfer stream.

3.1.3 Referenced Concepts

Proposals shall provide a means for unambiguous identification of any concept specified in a MOF-compliant metamodel that is referenced (but the specification is not included) in a transfer stream.

The XMI.references element is used to refer to concepts used but not included in an XMI specification. Refer to Section 9.4.3, *Metamodel Extension Mechanism* on page 104 for details.

3.1.4 UML Support

Proposals shall demonstrate support for import/export of UML models and the UML metamodel. This demonstration shall include demonstration of a round-trip model exchange without information loss. Submissions will be evaluated regarding the extent of the UML metamodel subset (including any MOF-compliant extensions) covered by the submitter's choice of examples.

XMI has been used extensively by the co-submitters as described in Section 2 Proof of Concept. This prototyping includes:

1. Round-trip transfer of UML models from a tool (e.g.: Rational Rose) to an XML file and back.

2. Transfer of UML models from between tools (e.g.: Select Enterprise to XML file to Rational Rose)

3. Transfer of UML models between a repository and tools (e.g.: Unisys UREP or IBM TeamConnection to XML file to Rational Rose)

4. Transfer of the complete UML metamodel between tools.

Refer to the appendix for details.

3.1.5 International Codesets

Proposals shall support use of international standard codesets.

The XMI uses the optional encoding declaration of XML to specify the character set. This follows the ISO-10646 (also called the Unicode) standard. XML also permits switching of encodings in a file.

3.2 Optional Requirements

3.2.1 Compact Data Representation

The interchange of metamodels may require a compact data representation in addition to the text-based representation as an alternative to the interface-based representation defined in the MOF.

Not addressed in this proposal.

3.2.2 Compatibility with other Metamodels and Interchange Formats

In order to preserve the investments of OMG members, proposals may be upward-compatible with the EIA/CDIF 1994 (CDIF94) Transfer Format standards. This does not imply downward-compatibility. The SMIF specification may contain constructs unsupported by CDIF94.

Not addressed in this proposal.

Proposals may contain an unambiguous, complete mapping of the concepts in the CDIF94 meta-meta-model to the concepts in the MOF.

Not addressed in this proposal.

Proposals may identify the impact of the proposed SMIF specification on transfer files produced using the CDIF94 Transfer Format standards. This includes identification of any changes to CDIF transfer files required to produce valid syntax and encoding per the proposed SMIF specification. This requirement may be met by providing a specification for a conversion utility for transfer files created using the CDIF94 Transfer Format standards to make them compliant with the proposed SMIF specification.

Not addressed in this proposal.

Proposals may provide transfer stream examples that use concepts from other industry standard metamodels.

Not addressed in this proposal.

Proposals may identify specific modeling language differences between EXPRESS and the MOF/UML and discuss ways to map between these languages. A direct mapping of all the concepts in either language to the other may not be possible.

Not addressed in this proposal.

Proposals may identify the impact of the proposed SMIF specification on existing schema definitions and transfer files produced using STEP EXPRESS. This may include identification of any changes to STEP EXPRESS files required to produce valid syntax and encoding per the proposed SMIF specification. Submissions may include a specification for converting STEP schemas and/or transfer files created using STEP EXPRESS standards to make them compliant with the proposed SMIF specification.

Not addressed in this proposal.

The submitters may consider addressing some of these optional requirements for the final submission.

3.3 Issues for discussion

Proposals in response to this RFP may discuss the usage and relevance of related technologies such as Meta-Object Definition Language (MODL), Object Constraint Language (OCL) and Universal Object Language (UOL) to the SMIF RFP. Note that these languages have been discussed in the OMG OAD Task Force recently.

MODL (non-normatively referenced in the OMG MOF standard) is a text-based language that is expressly designed for expressing MOF metamodels. Naturally, it has a direct correspondence with the MOF meta-metamodel. MODL was initially developed by the DSTC to support the MOF submission.

UOL is a text-based object modeling language for expressing UML and OML models. The alignment of the core concepts in the UML metamodel with those of the MOF meta-metamodel mean that UOL can also be used to express MOF metamodels. UOL is being developed jointly by Recerca Informàtica, Universitat Politècnica de Catalunya and Daimler-Benz Research and Technology in response to the SMIF RFP.

Since both MODL and UOL can both express MOF compliant meta-models, they can both be used as human-readable interchange formats for MOF meta-models. In the same way, UOL is a human-readable interchange format for UML models. However, neither of MODL or UOL is suitable as an interchange format for models in general.

OCL, as defined in the UML standard, is a language for expressing constraints over a collection of objects. OCL has been used to define semantic aspects of the MOF and UML standards, and is used in this proposal to define the XMI stream production rules. OCL can also be used to define semantic constraints in MOF metamodels and UML models. However, since OCL has no capability of modeling data structures, it is not directly applicable to model or metamodel interchange.

Note: the separation of information from presentation issues is a key feature of both XML and XMI. While this proposal does not address this issue, it will be feasible to use W3Cís eXtensible Style Language (XSL) to define “style sheets” for XMI. For example, XSL style sheets can be defined to map XMI encodings of MOF compliant metamodels onto either MODL or UOL. Similarly, we can map XMI encodings of UML compliant models onto UOL or the UML graphical notation.

Proposals in response to this RFP should discuss how to support semantic interoperability between tools that share and manipulate STEP schemas and STEP schema instances in addition to tools that support sharing and manipulation of OAD models. The proposal may provide or reference different specifications for transferring schemas and transferring schema instances as long as there is a way to reference the schemas when transferring schema instances.

This proposal does not address STEP schema interoperability. However, the MOF and its precursors have been used in a number of domains which entail model and schema transformations. Assuming that MOF metamodels for STEP schemas are defined, XMI could therefore be used to interchange STEP schemas and instances.

Proposals should include information on how to perform conformance tests (for checking syntax and transfer stream specific validation rules for schemas and schema instances) on transfer streams prior to import into other applications.

The XML Recommendation provides explicit rules for XML document validation, based on both the syntax of XML and the specific DTD of the document. This validation can be performed by any validating XML parser. An XML consumer can choose to validate the entire document before beginning the decoding process.

In XMI, the specific DTD for a document is produced from the model's MOF metamodel according to mapping rules in this specification. The DTD expresses the structural aspects of the meta-model. This means that any validating XML parser can check that an XMI document containing a model is structurally conformant to the model's meta-model.

The XML DTD language is not rich enough to represent all aspects of a MOF meta-model. In particular, it cannot express multiplicity constraints (i.e. cardinality and uniqueness) or arbitrary semantic constraints. Hence validation of an XMI stream by a standard XML parser does not guarantee full conformance.

Sharing of metamodels is the anticipated basis for full validation. An XMI stream header includes an unambiguous reference to the model's metamodel. Thus, an XMI enhanced XML parser can ensure total model conformance by validating an XMI stream against a local copy of its metamodel. Similarly, a MOF compliant model repository for a given metamodel can validate any model that is loaded into it. Note however, that exchange of incomplete models is also supported.

This may include recommendations for adding additional functionality to the MOF to satisfy transfer file conformance test requirements identified by the STEP community.

Proposals should discuss an approach to address this difference in problem scope. For example, proposals may describe how to use the MOF to describe STEP schemas at the same level as the UML meta-model.

The submitters believe that MOF is rich enough to be used to define STEP schemas at the same level as the UML metamodel. A possible approach is to define a mapping between the STEP meta-metamodel and the MOF meta-metamodel so that STEP schemas can be treated as MOF metamodels. Alternately, a MOF metamodel for STEP that allows STEP schemas to be expressed as MOF based models.

The MOF does not need extensions to handle conformance rules. The MOF already provides meta-metamodel elements (e.g. Model::Constraint) for attaching well-formedness rules (e.g. expressed in OCL or any other language) to a MOF metamodel. The MOF standard also addresses conformance and well-formedness of models. If we assume that STEP is incorporated into the MOF metadata framework using the second

alternative above, STEP conformance requirements can be handled as part of the MOF metamodel for STEP.

The focus of the XMI proposal on current and emerging OMG metadata standards. The submitters believe that integration of XMI and STEP EXPRESS to address EDI and related requirements is an important next step.

Proposals should discuss the connection, if any, between the proposed transfer format syntax and encoding and the Objects-by-Value syntax and encoding.

There is no direct connection between the XMI proposal and the new OMG Object-by-Value specification.

The MOF supports the use of the complete range of CORBA data types in metamodels using CORBA TypeCodes. This allows the MOF to evolve with extensions to the CORBA data types. As new CORBA data types are defined, XMI will be extended to support their transmission in models. The new Object-by-Value “value” types are no exception.

Object-by-Value encoding could be used for transmission of models in compact binary form. However, the submitters have not chosen to address this optional requirement.

Design Rationale 4

4.1 Design Overview

This submission proposes the Extensible Markup Language (XML), as defined by the W3C Recommendation 1.0, as the Stream-based Model Interchange Format. That recommendation includes specification of XML in Extended Backus-Naur Form (EBNF) notation, which is LL(1) parsable.

The encoding of metadata is specified by the ***XML Document Production Rules***. When these rules are applied to a model or model fragment, the result is an XML document. The rules ensure that encoding a given model or model fragment will always result in equivalent XML documents. Since these documents contain all of the information in the original model or model fragment, a stream consumer can apply the XML Document Production Rules in reverse to produce metadata that is an identical copy the original.

The XML Document Production Rules are expressed in detail using a combination of grammar fragments and OCL expressions.

4.2 XMI and the MOF

XMI is an interchange format for models and meta-models that are defined in conformance with the Meta Object Facility (MOF) standard. This section provides an overview of the MOF and gives a rationale for basing XMI on the MOF rather than some other modelling technology.

4.2.1 An Overview of the MOF

The MOF is the OMG's adopted technology for modelling metadata and representing it as CORBA objects. The MOF can support any kind of metadata that is describable using object modelling. The designers envisaged that the MOF would be used for a wide range of metadata; for example

- metadata repositories to support the software analysis, design and development processes,
- type repositories for types used by infrastructure services such as COS Trading, COS Events and ultimately the CORBA Interface Repository itself.
- metadata repositories for data warehousing and mining and database interoperability,
- metadata indices for free-text data sources such as online document collections and the world-wide web.

The MOF specification has three main parts; i.e. the MOF Model, the MOF IDL Mapping and the MOF's interfaces. The purpose of these components and the relationship between them will become clear as they are described.

The MOF Model

The "MOF Model" is the MOF's builtin meta-metamodel. The best way to understand the MOF Model is to think of it as an "abstract language" for defining MOF metamodels. This is analogous to the way that the UML metamodel is an abstract language for defining UML models. The MOF uses the UML notation rather than specify its own graphical notation.

There is an even closer parallel between MOF and UML in this area. While the two facilities have been designed for two different kinds of modelling (i.e. metadata versus object modelling), the MOF Model and the core of the UML metamodel are closely aligned in their modelling concepts. Indeed, the alignment is so close that UML's object modelling notation can easily be used to express MOF metamodels.

The three main metadata modelling concepts supported by the MOF are Classes, Associations and Packages.

- Classes can have Attributes and Operations at both "object" and "class" level. Attributes have the obvious usage; i.e. representation of metadata. Operations are provided to support metamodel specific functions on the metadata. Both Attributes and Operation Parameters may be defined as "ordered", or as having structural constraints on their cardinality and uniqueness. Classes may multiply inherit from other Classes.
- Associations support binary links between Class "instances". Each Association has two AssociationEnds that may specify "ordering" or "aggregation" semantics, and structural constraints on cardinality or uniqueness. When an Class is the type of an AssociationEnd, the Class may contain a Reference that allows navigability of the Association's links from a Class "instance".
- Packages are collections of related Classes and Associations. Packages can be composed by importing other Packages or by inheriting from them. Packages can also be nested, though this provides a form of information hiding rather than reuse.

The only other significant MOF Model concepts are DataTypes and Constraints. DataTypes allow the use non-object types for Parameters or Attributes. In the OMG

MOF specification, these must be data types or interface types expressible in CORBA IDL.

Constraints are used to associate semantic restrictions with other MOF model elements. This defines the well-formedness rules for the metadata described by a metamodel. Any language may be used to express Constraints, though there are obvious advantages in using a formal language like OCL.

The MOF IDL Mapping

The MOF "IDL Mapping" is a standard set of templates that map a MOF metamodel onto a corresponding set of CORBA IDL interfaces. If the input to the mapping is the metamodel for a given kind of metadata, then the resulting IDL interfaces are for CORBA objects that can represent that metadata. The mapped IDL are typically used in a repository for storing the metadata.

The IDL mapping is too large to describe here in detail. Instead, we will simply the main correspondences between elements in a MOF metamodel (M2-level entities) and the CORBA objects that represent metadata (M1-level entities).

- A Class in the metamodel maps onto an IDL interface for metadata objects and a metadata class proxy. These interfaces support the Operations, Attributes and References defined in the metamodel, and in the case of class proxy, provide a factory operation for metadata objects.
- An Association maps onto an interface for a metadata association proxy that supports association queries and updates.
- A Package maps onto an interface for a metadata package proxy. A package proxy acts as a holder for the proxies for the Classes and Associations contained by the Package, and therefore serves to define a logical extent for metadata associations, classifier level attributes and the like.

The IDL that is produced by the mapping is defined in great detail so that different vendor implementations of the MOF can generate compatible repository interfaces from a given metamodel. Similarly, the semantics of the mapped interfaces are defined by the MOF specification so that the metadata repositories can be interoperable.

In addition to the metamodel specific interfaces for the metadata (defined by the IDL mapping), MOF metadata objects also inherit from a group of Reflective base interfaces. These interfaces allow a 'generic' client program to access and update metadata without either being compiled against the metamodel's generated IDL or having to use the DII.

The MOF Interfaces

The final component of the MOF specification is the set of IDL interfaces for the CORBA objects that represent a MOF metamodel. These are typically not of interest to the meta-modeller who would use vendor supplied graphical editors, compilers and generator tools to access a MOF Model repository. However, they are of interest to

MOF-based tool vendors, and to programmers who need to access metadata using the Reflective interfaces.

In fact, there is not a lot to say about these interface, except to explain how they were derived. Conceptually, the MOF Model can be viewed as meta-metadata defined by a higher meta-level model. In the MOF specification, the MOF Model is defined using the MOF Model as its own modelling language. The IDL mapping is then applied to this metamodel (or strictly speaking meta-metamodel) to produce the MOF Model's IDL interfaces. Likewise, the MOF Model IDL's operational semantics are largely defined by the mapping and the OCL constraints in the MOF Model specification.

4.2.2 The relationship between XMI and MOF

The purpose of XMI is to allow the interchange of models in a serialised form. It is fairly obvious that in the overall context of the OMG there are many different kinds of model. Indeed, any "complete" set of metadata is arguably a model of something. The MOF is the OMG's adopted technology for describing metadata and defining metadata repositories.

From the point of view of a user of MOF-based metadata repositories, XMI represents an alternate way of transferring metadata from one repository to another. Since XMI is a transfer format rather than a CORBA interface, there is no need for ORB to ORB connectivity to effect the transfer: indeed any mechanism capable of transferring ASCII text will do. Thus XMI enables a mode of metadata transfer that significantly enhances the usefulness of the MOF.

From a wider point of view, XMI can be viewed as a common interchange format that can be used between any kind of metadata repository or between arbitrary XMI applications such as modeling tools, repositories, web authoring tools, etc. Any repository that can encode and decode XMI streams can use this capability to exchange metamodels with other repositories with the same capability. There is no need for such a repository to implement MOF defined CORBA interfaces, or even to "speak" CORBA at all.

Since XMI is text based and self descriptive, it also provides a route for interchange of meta-data with repositories that use other transfer syntaxes. This may be a possible solution for interoperability with CDIF-based repositories for example.

XMI is based on XML which does not have the same expressiveness as the MOF Model. Thus it is not possible to express Attribute cardinality and uniqueness constraints, or arbitrary metamodel Constraints in an XML DTD. In theory, this means that someone or something could produce an XMI document that, while conforming to the metamodel's XMI DTD, does not represent a well-formed model. However, this should not be a problem in practice. Firstly, a compliant MOF repository can detecting that a metamodel that is inserted into it is malformed. Secondly, a compliant MOF repository can store a partial or malformed model anyway.

Since the MOF Model is defined in terms of ifself, there is no reason why MOF servers cannot also exchange MOF meta-models using XMI. Indeed, a MOF repository sends the XMI files for both a model and its MOF meta-model, a receiving MOF

repository has in theory got enough information to fully reproduce the meta-model, even if it had no prior knowledge of the meta-model.

4.2.3 The relationship between XMI, MOF and UML

There are two points to make under this heading. First, as mentioned above, there is a close relationship (alignment) between the (meta-)modelling concepts of MOF and UML. Thus the increasing popularity of and knowledge of UML modelling concepts should make an XMI based on the MOF more accessible than an XMI based on other meta-modelling concepts (for example CDIF).

The second point is that the adopted OMG UML specification defines the UML meta-model as a MOF meta-model. For XMI, UML and MOF are the first of two OMG modeling standards that will be supported.

4.2.4 Why use the MOF as the basis for XMI?

There two ways of answering this question. One is to look of the advantages of the MOF, and the other is to look at the disadvantages of the alternatives.

The advantages of using MOF meta-modelling concepts in XMI are self-evident. The MOF is the adopted OMG technology for metadata and meta-modelling. This allows any OMG metadata (including UML models) to be encoded. In addition, the MOF's alignment with UML core means that a UML literate user should have less problems understanding XMI than would be the case with some alternatives.

At this stage there appear to be two alternative approaches proposed for SMIF. One is to use CDIF as the model interchange format, and the other is to define a model interchange format for UML.

A CDIF-based proposal would have the problem that the MOF Model and CDIF meta-metamodel are not fully aligned. This may present technical problems when trying to exchange metadata described by a MOF metamodel; e.g. UML models. [If you try to translate between CDIF and MOF at the m2-level you lose information. On the other hand, if you try to make CDIF the "top of the meta-stack" (i.e. by modelling the MOF Model as a CDIF metamodel) then the SMIF to model mapping must be defined an extra meta-level removed. Finally, if you respecify all MOF metamodels as CDIF metamodels, you have effectively taken MOF out of the meta-data picture!]

An approach which defines a model interchange format for UML alone is flawed in two respects. First, there are many kinds of model for which there is a fundamental mismatch in modelling paradigms with UML; e.g. relational schemas. A model interchange format that supports only UML and its derivitives is not going to support such models. Second, if you try to use UML as a meta-modelling language, you run into the problem that, unlike the MOF, UML has no standardised mapping to CORBA IDL.

4.3 XMI and XML

4.3.1 The roots of XML

The Web is the visual interface to the Internet's vast collection of resources. HTML (HyperText Markup Language) is the predominant form for expressing the Internet's web pages. HTML consists of a set of display tags which specify the visual layout of the page contents for web browsers. Between the tags is the content, the information designed to be displayed on the page. The content (data) and the meaning of the content (metadata) are mixed with the layout information to provide visually interesting results for a human viewer when displayed in a web browser. For automated access to web sites, however, the extraction of information is quite difficult since visual interpretation is often required. HTML, while flexible enough to provide visual web pages, lacks to the capability to deliver general electronic interchange to the Internet.

HTML is a subset of the more powerful SGML (Standard Generalized Markup Language), a sophisticated tag language which separates view from content and data from metadata. Due to the complexity of SGML's rich feature set, widespread use is not practical for many applications.

XML, the Extensible Markup Language, is a new data format for electronic interchange designed to bring structured information to the web. XML is an open technology standard of the World Wide Web Consortium (W3C), the standards group responsible for maintaining and advancing HTML.

XML is a subset of SGML which maintains the important architectural aspects of contextual separation while removing nonessential features. XML focuses on the ability to express rules for the structure of data (grammar) and a document format for clearly expressing the data within its contextual metadata. Document contents can be more easily interchanged on the Internet since automated systems can to separate the data and metadata and validate the document with its grammar. The XML document may be expressed visually for human users by applying layout style information with technologies such as XSL (Extensible Style Language). Web sites and browsers are rapidly adding XML and XSL to their functionality.

Another important feature of XML is its inherent simplicity. Like HTML, there is very little required to get started. XML documents can be created by hand with any text editor. XML documents are similar in ease of use and human readability to HTML, and, due to its more structured nature, is in some cases simpler.

4.3.2 Benefits of XML

There are several benefits of basing metamodel interchange on XML. XML is an open standard, platform and vendor independent. XML supports the international character set standards of extended ISO Unicode. XML is metamodel-neutral and can represent metamodels compliant with OMG's meta-metamodel, the MOF. XML is programming language-neutral and API-neutral. XML APIs are provided in additional standards,

giving the user an open choice of several access methods to create, view, and integrate XML information. Leading XML APIs include DOM, SAX, and WEB-DAV.

XML is validated through the wide experience and proven capabilities of the members of the XML family: SGML, used in high-end document processing, and HTML, the predominant language of the web. XML is the next step in the evolution of the web, as demonstrated by its incorporation into the latest upcoming versions of the leading web browsers by Netscape and Microsoft. This enhances the ability of XMI documents based on XML to be smoothly integrated into the information web of the Internet.

There is a growing set of tools available for XML development, including a complete, free, commercially unrestricted XML parser written in Java available from one of the submitting companies (IBM). A variety of other support tools are available on the Internet. The simplicity of XML and widespread tool support provide a very low cost of entry.

4.3.3 XML and the industry

Applications using are described in many locations on the web. Included are web commerce, publishing, repositories, modeling, databases and data warehouses, services, financial, health care, semiconductors, inventory access, and more. Companies involved in standardizing XML include: Adobe, ArborText, DSTC, HP, IBM, Microsoft, Netscape, Oracle, Platinum, Select, Sun, and Xerox**.**

XML has spawned a large number of books in response to the widespread interest it has received. Amazon.com lists 28 books published in the last year on XML, including two books in the “XML for Dummies” series. The cover article of Byte Magazine’s March 1998 issue was on XML, with a multi-page article by Bill Gates.

4.3.4 How XML works

This section provides a simple overview of XML technology. Additional features are described in sections of the submission which use particular aspects XML extensively.

Structure elements

XML documents are tree-based structures of tags containing nested tags and data. In combination with its advanced linking capabilities, XML can encode a wide variety of information structures. The rules which specify how the tags are structured are called a DTD, or a Document Type Declaration.

XML tags can be very simple. A tag consists minimally of a tag name enclosed by less- and greater-than signs. For example, <car> is an XML tag. Tags in XML are always nested as open-close pairs, similar to the concept in most programming languages of Begin and End. To close a tag, precede the tag name with a slash symbol. For example, </car> closes the tag above. Tags may contain other tags which may contain other tags in turn. The innermost tag must be closed before its containing tag may be closed. The requirement to match the beginning and ending tags is what

provides XML with the tree data structure and an architectural foundation missing from HTML.

Enclosed tags and text are together called the “content” of the enclosing tag. The formal name for an opening and closing tag pair is an “element.”

Example

This is an example document describing a car.

```
<Car>
<Make> Ford </Make>
<Model> Mustang </Model>
<Year> 1998 </Year>
<Color> red </Color>
<Price> 25000 </Price>
</Car>
```

The car contains five elements which describe it more detail: Make, Model, Year, Color, and Price. Each of those elements contain text with a value and a closing tag.

DTD

A DTD for the car would contain the following declaration: <!Element Car (Make, Model, Year, Color, Price)> This indicates that for a Car to be valid, it must contain each of the Make, Model, Year, Color, and Price elements. The declaration for an element can have a more complex grammar, including multiplicities (zero to one ‘?’, one ‘ ‘, zero or more ‘*’, and one or more ‘+’) and logical-or ‘|’.

DTDs are typically external files referenced using a URI. For example, “http://www.xmi.org/car.dtd”, or “file:car.dtd”.

The DTD specifies the metamodel by declaring the rules the model elements must follow. The document is the model since it carries the model elements following the DTD metamodel.

Attributes

In addition to contents, the element declaration may contain the declaration of element attributes. The attributes are specified as part of the opening tag. For example: <Class name=”c1”> </Class>. The declaration of the attributes in the DTD using an ATTLIST. For example, <!ATTLIST car name CDATA #REQUIRED >. This indicates that specifying the name of the Class is required in every Class tag, and that the name consists of a character data string.

XML has a special attribute, the ID, which provides a uniqueness identifier to an element within a document. The ID is discussed in detail in the section on XMI IDs.

Correctness

The example document above is called “well-formed” because the elements are properly structured as a tree with the opening and closing tags correctly nested. Well-formed documents are essential for information exchange.

The next level of semantic reliability is “validation.” The element structure may have grammatical rules regarding the placement of elements specified in the DTD. Although a DTD is not required to be specified in a document, and it is an option of the receiver as an optimization technique not to use the DTD, without the DTD the highest level of correctness XML can assert is “well-formed.”

The highest level of reliability is semantic correctness, a level beyond the capabilities of XML, but not the document creator and readers. This level requires domain knowledge that is not expressed the document, such as “is that color manufactured for that combination of make, model, and year.”

Architecture

XML as used in XMI is fully compatible with the four layers of the OMG meta-modelling architecture, illustrated in Table 1 below. To transfer an (M1 level) model, an (M2 level) XML DTD that corresponds to an (M2 level) MOF metamodel describes the encoding of an (M1 level) XML document that contains the model. For example, a UML model is encoded in conformance to a UML DTD which corresponds to the UML metamodel.

MOF compliant metamodels can be interchanged at the next meta-level in the metadata architecture. Thus, an (M2 level) metamodel such as the UML metamodel is encoded in conformance with an (M3 level) XML DTD for the (M3 level) MOF meta-metamodel.

The XMI proposal includes concrete DTD’s for UML and MOF, as well as DTD generation rules for additional future MOF compliant metamodels, or future versions of existing metamodels.

Table 1: OMG MetaModelling Architecture

M3	MOF MetaMetaModel	MOF DTD	
M2	UML MetaModel (and others)	UML DTD (and others)	MetaModels as XML Documents
M1	UML Models (and others)		UML Models (and others) as XML Documents
M0	Instances		

4.3.5 XML and the OMG

There is strong synergy between the OMG technologies and XML. OMG defines CORBA as the medium for interchange of data between objects. XML is an ideal interchange medium for OMG metadata.

OMG can use the MOF and XMI to leverage XML by taking the following steps. The OMG can initiate processes to standardize MOF-based metamodels for metadata of significance to industry. XMI can then be used to generate standard XML DTDs for these metamodels. The DTDs would allow the interchange of metadata, both between and beyond CORBA-based systems.

The XMI submitters believe that this approach would enhance the OMG's position as providing leadership in the data and metadata interchange standards of the future.

4.3.6 XML technologies

The following are capsule summaries of additional XML technologies which are in the process of being standardized by the W3C and other organizations and will further enhance the capabilities of XML. The XMI submission is designed to be upwards compatible with these technologies. However, since none are in their final form as adopted recommendations, XMI does not place any dependencies nor directly make use of any nonstandard technology. In addition, some of these technologies may be adopted at a later time by their respective organizations, and it is possible although not anticipated that XMI may be revised at a later time to enable their more efficient use.

Namespaces - The namespace draft by the W3C is work in progress with the goal of providing support for multiple DTDs in the same document. Each DTD is given a local namespace within a document (no global registration necessary) which prevents any conflicts by differing definitions of similarly named constructs.

Links - There are two linking technology drafts in progress at the W3C which provide advanced linking facilities which are integrated with web technology. XLink is for cross document links and XPointer is for links within a document. They are used together and are discussed in more detail in the discussion of the XMI Reference Element section.

There are three proposals for enhancing the base capabilities of XML at the W3C. RDF (Resource Description Framework) is a working draft specification for infrastructure to support web metamodels. RDF-Schema is a working draft to provide types for XML. XML-Data is a note to the W3C for public comment on providing schemas and types for XML.

XSL - Extensible Style Language is a working draft of the W3C which specifies user-definable declarative transforms of XML documents with the goal of providing formatting style information. XSL is used in conjunction with XML to create the visual layout of the underlying XML data and metadata.

There are three major APIs to XML. DOM, the Document Object Model, is a language-neutral interface to XML documents for creation and reading data and metadata information. DOM also works with style processing and scripts. SAX is an

event-driven API for XML parsing. Web-DAV is an API for Web based Distributed Authoring and Versioning and is currently a working draft of the IETF (Internet Engineering Task Force) standards body. It uses the HTTP protocol to provide online, distributed XML access and modification.

4.4 Specific Design Goals and Rationale

4.4.1 Universally Applicable Solution

The SMIF proposal shall provide the means to define an interchange format for the data of any metamodel which is an instance of the OMG Meta Object Framework (MOF), without requiring specific knowledge of the metamodel.

The XMI proposal defines DTD generation and stream production rules that can be used to transfer models described by any MOF-based metamodel.

Since XMI allows interchange of MOF metamodels, it is feasible to implement tools that can consume and produce fully valid XMI model documents with no prior knowledge of the metamodel. (This assumes that all of the Constraints in the metamodel are expressed in a constraint language that the tools can interpret.)

4.4.2 Model Fragments

The SMIF proposal shall allow model fragments to be produced and consumed.

Obtaining closure over an entire model could encompass a great many more model elements than are required by a stream consumer. The consumer might already have many of those elements, such as built-in types. The flexible generation of DTDs, and the use of XML linking – via the XML Linking Language (XLink) – makes it possible to exchange arbitrary model fragments.

4.4.3 Ill-Formed Models

The SMIF proposal shall not require a model to be well-formed.

Requiring a modeler to bring a model into compliance with all well-formededness rules before sharing is too restrictive. Ideas need to be shared before all the details are filled in. For a given MOF-defined metamodel, the candidate model only must meet:

- the XML validation rules (including those specified by the DTD corresponding to the metamodel);
- the set of constraints defined in the metamodel with the evaluationPolicy attribute having a value of immediate; and
- the intrinsic constraints on the metamodel which are immediately enforced (maximum multiplicity constraints, type constraints, etc.).

4.4.4 Standardised Transfer Syntax

The SMIF proposal shall define the generation of a standard transfer syntax for a model, based solely on the model's metamodel.

The typical means of specifying the syntax for a data interchange format is in the form of a reference document which lists the contents of files and fields, etc. While useful to the human coder who must implement the import and export programs, such a document can be ambiguous or incomplete, since it is prepared by a human author. Errors and omissions by the syntax author mean that the import/export coder must make arbitrary decisions, resulting in cases where data cannot be exchanged.

The rules provided in this specification allow for the automated generation of XML DTDs based on the original MOF specification of a metamodel. Such DTDs do not have the problem of ambiguities and other shortcomings introduced by human authors. They are also machine-readable, which has the potential for the developmen of automated tools to help in the development of import/output programs.

4.4.5 Model Versions

The SMIF proposal shall support versions of models.

The XMI proposal allows model and metamodel version information to be included in the XMI header. It is up to the producers and consumers of XMI streams to manage the allocation of version numbers.

4.4.6 Model Extensibility

The SMIF proposal shall allow metadata conforming to a standard metamodel and one or more non-standard extensions to be transmitted simultaneously

The XMI proposal takes advantage of a key attribute of XML; i.e. an XML document is self describing. XMI documents are divided into two parts. The first part contains metadata that conforms to the MOF metamodel. The second part contains additional metadata that is not described by the base metamodel. This part may have multiple sections, each corresponding to the model extensions made by a particular tool.

For example, many UML tool vendors add extra attributes to various UML classes to support "value added" features of their tools. While UML provides Tagged Values and Stereotypes to support these extensions, this approach is clumsy and can result in name conflicts when metadata is exchanged between different vendors' tools. Using XMI, tool vendors can define new classes to extend the standard UML classes. The resulting metadata is encoded a separate, self-contained section of the XMI document, simplifying its management.

4.4.7 MOF as an Information Model

The SMIF proposal shall be capable of being used to transmit operational data as well as metadata.

The distinction between the MOF Model as a meta-metamodel and a metamodel is only in the use of the models it defines. When an instance of the MOF Model is used to define the UML meta-model, the MOF Model is a meta-metamodel. When a MOF Model defines a model and instances of that model are not intended as models, then the MOF Model is a meta-model.

Usage Scenarios 5

5.1 Purpose

This section describes some of the problems that IT users and vendors face today and illustrates how XMI helps to address these problems.

5.2 Combining tools in a heterogeneous environment

Implementing an effective and efficient IT solution for an enterprise requires a detailed understanding of processes, rules and data used by the business and how each map to supporting applications. Without this information, it is difficult to assess the effectiveness of the application components in use, to identify opportunities for improvement and to evaluate candidate solutions. A further complication is that the applications in use will probably originate from a variety of sources and consequently be a mix of custom solutions and packaged applications implemented in a variety of technologies.

The reality is that no single tool exists for both modelling the enterprise and documenting the applications that implement the business solution. A combination of tools from different vendors is necessary but difficult to achieve because the tools often cannot easily interchange the information they use with each other. This leads to translation or manual re-entry of information, both of which are sources of loss and error.

XMI eases the problem of tool interoperability by providing a flexible and easily parsed information interchange format. In principle, a tool needs only to be able save and load the data it uses in XMI format in order to inter-operate with other XMI capable tools. There is no need to implement a separate export and import utility for every combination of tools that exchange data.

The makeup of an XMI stream is important too. It contains both the definitions of the information being transferred as well as the information itself. Including the semantics of the information in the stream enables a tool reading the stream to better interpret the

information content. A second advantage of including the definitions in the stream is that the scope of information that can be transferred is not fixed; it can be extended with new definitions as more tools are integrated to exchange information.

5.3 Co-operating with common metamodel definitions

The extent of the information that can be exchanged between two tools is limited by how much of the information can be understood by both tools. If they both share the same metamodel (the definition of the structure and meaning of the information being used), all of the information transferred can be understood and used. However, gaining consensus on a totally shared meta model is a difficult task even within a single company. It is more likely that a subset of the meta model can be shared with each tool adding its own extensions. The need to agree the structure and syntax for encoding as a stream adds further complexity.

XMI builds on the OMG Meta Object Facility that already provides a standard way to define metamodels within the OMG. UML is one example of a metamodel that can be defined in the MOF and which has already adopted as a standard by the OMG. The model definitions required for the transfer of UML models using XMI are included with this submission as a set of concrete XML DTD's. Any tool vendor can use these definitions to save and load UML models in XMI format without the need for an implementation of the MOF. This is a practical step to encourage as many tool vendors as possible to adopt the standard by keeping their initial investment low.

However, manually writing the XML DTD's for a metamodel is tedious, error prone and subject to variations in how model concepts are implemented in XML. Using XMI, the XML DTD's for a metamodel are obtained by defining the metamodel in MOF and then applying the XMI generation rules. The generation approach ensures that a given metamodel will always map to the same set of XML DTD's regardless of which vendor implemented the MOF and the XMI stream protocol.

The fact that the MOF meta-metamodel, (the description of the MOF itself), can be defined in the MOF itself means that XMI can also be used to transfer metamodel definitions from one MOF to another. Being able to share metamodel definitions is an important step to promoting the use of common metamodels by different tool vendors. The combination of the MOF and XMI provides an effective way for vendors to co-operate on the definition and use of common models.

As mentioned earlier, having a shared model is not enough on its own. Each vendor must be able to extend the information content of the model to include items of information that have not been included in the shared model. XMI allows a vendor to attach additional information to shared definitions in a way that allows the information to be preserved and passed though a tool that does not understand the information. Loss-less transfer of information through tools is necessary to prevent errors that may be introduced by the filtering effect of a tool passing on only that information it can understand itself. Using this extension mechanism, XMI stream can be passed from tool to tool without suffering information loss.

5.4 Working in a distributed and intermittently connected environment

Another aspect of sharing metadata is encountered when trying to provide effective consultancy services. This requires the ability to exploit and share best practices between the consultants of the group. However, consultants on site typically have restricted connectivity to the network and limited bandwidth for exchanging models and design information with their colleagues.

The use of XMI for a metadata interchange facilitates the exchange of model and design data over the Internet and by phone. Appearing as set of hyper-linked Internet documents, the data to be transferred can be transported easily through firewalls and downloaded using a modem. The documents in a related set are accessed on-demand and cached locally to eliminate the retransmission of frequently used sub-documents.

The remote consultant would be equipped with a notebook installed with a set of tools that can import and export metadata in XMI format. Connecting to the home site via the Internet or dialup networking, the consultant can download metadata resources published as links from pages on a standard WEB server. The same mechanism can be used to upload modification that the consultant wants to publish for his colleagues.

Typically, the type definitions that defines the semantics of a transfer do not change frequently and can be stored in a separate document from the actual data to be transferred. The type definitions are versioned to allow consistency checking. On the first use of the type definitions, the document containing the type definitions would be downloaded and cached on the consultant's machine. Subsequent transfers are be faster because only the metadata content is transferred while the cached type definitions are reused.

5.5 Promoting design patterns and reuse

Consultants will often need to integrate their work with the development tools being used at customer site. This often results in the consultants actually using the same tool set as the customer. Of course, the tools used will differ from customer to customer.

The problem in this scenario is that it is difficult to develop and exploit best practices across the consulting group without being able to exchange model and design data between different tool sets.

XMI addresses this problem by defining a standard format for interchange of model and design data between different tool sets. It does not require the tool vendors to invest in the same technology stack. It only requires them to agree on the Meta models for the data to be shared, plus a standard mechanism for extending that Meta model with their own types of metadata.

The XMI format allows Meta models to be standardised and revised over time, the set of Meta models being extensible. For example, this initial submission covers just the UML Meta model but other Meta models can be agreed and added without affecting the current set of Meta models.

Vendor extensions to a standard meta model are designed to enable other vendors tools to process and use the standardised information while being able easily retain and pass through vendor specific extensions.

XMI DTD Design Principles 6

6.1 Purpose

6.2 Overview

This chapter contains a description of the XML Document Type Definitions (DTDs) that may be used with the XMI specification to allow some metamodel information to be verified through XML validation. The use of DTDs in XMI is described first, followed by the requirements that each DTD used by XMI must satisfy. Then a description of the XML elements defined by this specification is presented. That description is followed by an explanation of a DTD representing the UML metamodel.

It is possible to define how to automatically generate a DTD from the MOF metamodel to represent any MOF-compliant metamodel. That definition is presented in chapter 7.

6.3 Use of XML DTDs

An XML DTD 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 DTD can be generated for any MOF-based metamodel. However, the use of DTDs is optional; an XML document need not reference a DTD, 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.

It can be advantageous to perform XML validation on the XML document containing MOF metamodel 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,

however, since XML validation does not perform all of the verification that could be done.

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

It is possible to use an internal DTD to provide all of the declarations of XML elements described in this chapter. However, it is advantageous to use an external DTD, because the DTD need not be transmitted along with each XML document that contains the metadata. An internal DTD may be used in addition to an external DTD, for example to specify extensions to the metamodel.

When the XML Namespace specification is adopted by the W3C and the XML specification is extended to support multiple DTD validation, multiple DTDs can be specified. This will allow a DTD for XMI, a DTD for the metamodel, and DTDs for extensions all to be used at once. With Namespaces, the document including DTDs specifies the local name of each DTD. The local name acts as a prefix to all the elements declared in a DTD and avoids any name collisions so that it will not be necessary to prefix the XMI elements.

6.3.1 XML Validation of XMI documents

XML validation can determine whether the XML elements required by this specification are present in the XML document containing metamodel 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 metamodel data conforms to a metamodel. 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 metamodel'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 metamodel in an XML DTD, and the rules for automatic generation of a DTD preclude the use of semantic constraints that could be encoded in a DTD manually, but cannot be automatically encoded.

Finally, XML validation can be used to validate extensions to the metamodel, because extensions must be represented as elements declared in either the external DTD or the internal DTD.

6.3.2 Requirements for XMI DTDs

Each DTD used by XMI must satisfy the following requirements:

- All XML elements defined by the XMI specification must be declared in the DTD.
- Each metamodel construct (class, attribute, and association) must have a corresponding element declaration, as described below. The element declaration may be defined in terms of entity declarations, also, as described below.
- Any XML elements that represent extensions to the metamodel must be declared in the external DTD or internal DTD.

6.4 Basic Principles

This section discusses the basic organization of an XML DTD for XMI. Detailed information about each of these topics is included later in Section 6 of this chapter.

6.4.1 Required XML Declarations

This specification requires that a number of XML element declarations be included in DTDs that enable XML validation of metadata that conforms to this specification. These declarations must be included in the DTD because there is no mechanism currently available in XML to validate a document against more than one external DTD. Some of these XML elements contain metadata about the metadata to be transferred, for example, the identity of the metamodel associated with the metadata, the time the metadata was generated, the tool that generated the metadata, whether the metadata has been verified, etc. Other XML elements enable associations to be made between XML elements within a single XML document or between XML elements in different XML documents.

All XML elements defined by this specification have the prefix “XMI.”. They have this prefix to avoid name conflicts with XML elements that would be a part of a metamodel. After XML namespaces become a W3C recommendation rather than a working draft, it may be possible to place all of the required XML elements in a single namespace and use the XML namespace mechanism to avoid name conflicts.

In addition to required XML element declarations, there are two attributes that must be defined according to this specification. Every XML element that corresponds to a metamodel class must have a required attribute of XML type ID. This attribute is used to associate an XML element with another XML element. The other attribute determines whether the XML element is defined locally or whether it is a proxy for an XML element in another document.

6.4.2 Metamodel Class Representation

Every concrete metamodel class is represented in the DTD by an XML element whose name is the class name. The element definition lists the attributes of the class; references to association roles of the class; and the classes that this class contains, either explicitly or through composition associations.

Every attribute of a metamodel class is represented in the DTD by an XML element whose name is the attribute name. The attributes are listed in the content model of the XML element corresponding to the metamodel class in the order they are declared in the metamodel.

Each association between metamodel classes is represented by two XML elements that represent the roles of the association ends. The multiplicities of the association ends are translated to the XML multiplicities that are valid for specifying the content models of XML elements. If the association does not represent containment, the content model of the XML element representing the role of the association end contains an XML element that allows XML elements to reference other XML elements.

If the association represents containment, the content model of the XML element that represents the container class has an XML element with the name of the role at the contained association end, with the multiplicity defined for its association end. The XML element representing the contained role has a content model that allows XML elements representing the contained class and any of its subclasses to be included.

6.4.3 Metamodel Extension Mechanism

Every XMI DTD contains a mechanism for extending a metamodel class. Any number of “XMI.extension” elements can be included in the content model of any class. These extension elements have a content model of ANY, allowing considerable freedom in the nature of the extensions. In addition, the top level XMI element may contain zero or more “XMI.extensions” elements, which provides for the inclusion of any new information. One use of the extension mechanism might be to associate display information for a particular tool with the metamodel class represented by the XML element.

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. Also, any XML elements that are put in either the “extension” or “extensions” XML elements must be declared.

6.5 XMI DTD and Document Structure

Every XMI DTD consists of the following declarations:

- An XML version processing instruction. Example: <? XML version=”1.0” ?>
- An optional encoding declaration which specifies the character set, which follows the ISO-10646 (also called Unicode) standard. Example: <? XML version="1.0" ENCODING=”UCS-2” ?>.

- Any other valid XML processing instructions.
- The required XMI declarations specified in Section 5.
- Declarations for a specific metamodel.
- Declarations for extensions.

Every XMI document consists of the following declarations:

- An XML version processing instruction.
- An optional encoding declaration that specifies the character set.
- Any other valid XML processing instructions.
- An optional external DTD declaration with an optional internal DTD declaration. Example: <! DOCTYPE XMI SYSTEM “http://www.xmi.org/xmi.dtd“ >

XMI imposes no ordering requirements beyond those defined by XML. After the XML Namespace specification is adopted, external DTDs are expected to be referenced in a manner similar to: <? xml:namespace ns='http://www.xmi.org/xmi.dtd' prefix='xmi' ?>. This example should allow all elements declared in the namespace to be unambiguously prefixed with xmi.

The top element of the XMI information structure is the XMI element. 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 which extend XMI and embed XMI elements within their XML elements.

6.6 Necessary XMI DTD Declarations

This section declares the elements and element attributes whose definitions must appear in any valid XMI DTDs.

6.6.1 Necessary XMI Attributes

XMI.id

The element representing a metamodel class in an XMI DTD must include the XMI.id attribute in its Attlist. An example of the use of this attribute is

```
<!ELEMENT x …>
<!ATTLIST x XMI.id ID #REQUIRED…>
```

The XMI.id attribute is used as the target of a reference by the XMI.reference element (defined in the XMI Elements section). The policies for handling IDs are discussed in the section “XMI IDs.”

6.6.2 XMI.remote

The element representing a metamodel class in an XMI DTD must contain the XMI.remote attribute in its attlist. An example of the use of this attribute is

```
<!ELEMENT x …>
<!ATTLIST x XMI.id ID #REQUIRED    XMI.remote (true|false) “false”…>
```

The XMI.remote attribute is used to specify when the XMI.remoteContent element (defined in the XMI Elements section) is used. This specification requires that XMI.remote be set to "true" when XMI.remoteContent is used; if XMI.remoteContent is not used, it may be left unspecified or explicitly set to "false".

These entities may be grouped into an entity for convenience. An example, from the example UML DTD is:

```
<!ENTITY % XMI.ElementAttributes ‘XMI.id ID #REQUIRED
                                 XMI.remote (true | false) “false” ‘>
```

6.6.3 Common XMI Elements

Every XMI-compliant DTD must include the declarations of the following XML elements:

- XMI
- XMI.header
- XMI.content
- XMI.extensions
- XMI.documentation
- XMI.metamodel
- XMI.owner
- XMI.contact
- XMI.longDescription
- XMI.shortDescription
- XMI.exporter
- XMI.exporterVersion
- XMI.exporterID
- XMI.notice
- XMI.reference
- XMI.remoteContent
- XMI.field

- XMI.struct
- XMI.seqItem
- XMI.sequence
- XMI.arrayLen
- XMI.array
- XMI.enum
- XMI.discrim
- XMI.union
- XMI.any

6.6.4 XMI

The top level XML element for each XMI document is the XMI element. Its declaration is:

```
<!ELEMENT XMI (XMI.header, XMI.content, XMI.extensions*) >
<!ATTLIST XMI
        xmi-version CDATA #FIXED "1.0"
        timestamp   CDATA #IMPLIED
        verified (true | false) #IMPLIED
>
```

The “xmi-version” attribute is required to be set to “1.0”. This indicates that the metadata conforms to this version of the XMI specification. Revised versions of this standard will have another number associated with them, but there is no guarantee that any particular numbering scheme will be used. The “timestamp” indicates the date and time that the metadata was written. The “verified” attribute indicates whether the metadata has been verified. If it is set to “true”, verification of the model was performed by the document creator at the full semantic level of the metamodel. In that case, XML validation should find errors only in encoding or transmission.

The format for timestamps is not defined in this initial submission.

6.6.5 XMI.header

The “XMI.header” XML element contains XML elements which identify the metamodel, as well as an optional XML element which contains various information about the metadata being transferred. Note that at least one metamodel XML element must be present. The “XMI.header” declaration is:

```
<!ELEMENT XMI.header (XMI.documentation?, XMI.metamodel+) >
```

6.6.6 XMI.content

The “XMI.content” XML element contains the actual metadata being transferred. It may represent model information or metamodel information. Its declaration is:

```
<!ELEMENT XMI.content ANY >
```

6.6.7 XMI.extensions

The “XMI.extensions” XML element contains XML elements which contain metadata that is an extension of the metamodel. This information might include presentation information associated with the metadata, for example. Its declaration is:

```
<!ELEMENT XMI.extensions ANY >
```

6.6.8 XMI.documentation

This XML element contains information about the metadata being transmitted, for instance the owner of the metadata, a contact person for the metadata, long and short descriptions of the metadata, the exporter tool which created the metadata, the version of the tool, and copyright or other legal notices regarding the metadata. In addition, other information can be included as text within this element, since its content model is mixed. The declaration is:

```
<!ELEMENT XMI.documentation (#PCDATA |
                XMI.owner | XMI.contact |
                XMI.longDescription |
                XMI.shortDescription | XMI.exporter |
                XMI.exporterVersion | XMI.notice)* >

<!ELEMENT XMI.owner ANY >
<!ELEMENT XMI.contact ANY >
<!ELEMENT XMI.longDescription ANY >
<!ELEMENT XMI.shortDescription ANY >
<!ELEMENT XMI.exporter ANY >
<!ELEMENT XMI.exporterVersion ANY >
<!ELEMENT XMI.exporterID ANY >
<!ELEMENT XMI.notice ANY >
```

6.6.9 XMI.metamodel

This XML element identifies the metamodel to which the metadata that is transferred conforms. There may be multiple metamodels, if the metadata conforms to more than one metamodel. There may also be more than one metamodel if it is desired to identify which version of the MOF metamodel the metamodel or metamodels are compliant with. Including this element enables tools to perform more verification of

the metadata to the metamodel than is possible to perform by XML validation. This element is expected to become a simple XLink when it becomes a recommendation of the W3C.

The “XMI.metamodel” declaration is:

```
<!ELEMENT XMI.metamodel ANY>
<!ATTLIST XMI.metamodel
        name    CDATA #REQUIRED
        version CDATA #REQUIRED
        href    CDATA #IMPLIED
>
```

The “name” and “version” attributes are the name and version of the metamodel, respectively. The “href” attribute may contain a URI that contains metamodel data. Since the content is ANY, additional documentation is possible.

6.6.10 XMI.reference

This XML element is the mechanism used by XMI to associate XML elements with other XML elements, either within one XML document or between XML documents. It will rely on the W3C XLink and XPointer recommendations when they become available. XLink specifies references between documents and XPointer is used to navigate within documents. Under those specifications, this element is expected to become a simple inline XLink. Once the working draft is a recommendation, all of the attributes available will be included.

The declaration of “XMI.reference” is:

```
<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
        target IDREF #IMPLIED
        href   CDATA #IMPLIED
        expectedType CDATA #IMPLIED
        content-title CDATA #IMPLIED
>
```

The “target” attribute may be used to specify the XML ID of an XML document within the current XML document. Every construct that can be referred to has a local XML ID, a string that is locally unique within a single XML file. The XPointer part of a Reference uses the ID to find the construct . The XPointer specification also has relative addressing capabilities within a document that may be used. The choice of absolute ID-based addressing or relative addressing is made by the document creator on a per-reference basis.

The “href” attribute may be used to specify an optional URI and XPointer that identify an XML element in another XML document. The "href" attribute must contain a locator for the model construct referred to. This model construct should be of the form URI "|" NAME, where URI locates the file that contains the model construct, and NAME is the value of the ID attribute of the referenced model construct. If the URI is

not given, then NAME must be the value of an ID attribute in the current file. NAME is a shorthand for XPointer id(NAME).

In elementary use, href could refer to another element id in the same XML file using href="|id".

The “expectedType” attribute may be used to indicate what type of XML element should be referred to. Specifying a value for this attribute will enable tools to perform validation that the other XML element is of the expected type, as well as allow optimizations regarding following the link.

In XML there is currently no mechanism to enforce that the actual type of the XML element referred to is the desired one. Some tools might issue a warning if the type does not match the type of model construct actually referred to. This caching of expected information could be extended with other expected information attributes.

Any type of content can be allowed for the Reference XML element. This allows the receiver of the XML document to add additional processing to the content. For example, the content could be empty, contain an SQL query into a repository, a phone number, or a human readable version of the target's name (useful in web browsers or any other convention desired.

The “XMI.reference” element is used in the content models of XML elements representing the roles of association ends, as well as with the XMI.remoteContent element to locate the XML element for which it is acting as a proxy.

The following is an example DTD fragment used in the examples below:

```
<!ELEMENT From ANY>
<!ATTLIST From XMI.id ID #REQUIRED>

<!ELEMENT To EMPTY>
<!ATTLIST To XMI.id ID #REQUIRED>
```

This first example is one way to refer to a "To" element from a "From" element when both elements are in the same file:

```
<From id="from1">
    <XMI.reference href="|to1" />
</From>
<To id="to1" />
```

The second example is a document fragment where the "From" element refers to a "To" in another file:

```
<From id="from2">
    <XMI.reference href="http://www.distant.com/xml/target.xml|to2"
        content-title="This is a reference"
            expectedType="To">
    </XMI.reference>
</From>
```

The document “target.xml,” contains the following "To" element:

```
<To id="to2" />
```

The third example is a more complex case where the URI links to a database using the contents for an open-ended addressing expression.

```
<From id="from3">
    <XMI.reference   href="http://www.distant.com/db?output=xml"
                content-title="This is a reference"
                expectedType="To">
                Select * from ToTable where primaryKey = 3
    </XMI.reference>
</From>
```

The resulting XML could have been generated on demand or the query executed at a later time. Note that the output of the query could be in a form other than XML, stored in a file, etc.

XMI.remoteContent

This element is used in XML elements that are proxies for XML elements in other documents. If an XML element corresponding to a metamodel class has its “XMI.remote” attribute set to “true”, then its content will be a “XMI.remoteContent” element, which in turn will contain an “XMI.reference” element.

The “XMI.remoteContent” declaration is:

<!ELEMENT XMI.remoteContent (XMI.reference) >

6.6.11 XMI Datatype Elements

It is necessary to include some MOF datatypes to represent data values so they can be accurately represented and reified. The MOF datatypes that must be included in each DTD allow users to represent structures, sequences, arrays, enumerations, unions, and the CORBA datatype ANY.

The declarations of these elements are as follows:

```
<!ELEMENT XMI.field ANY >
<!ELEMENT XMI.struct (field)+ >

<!ELEMENT XMI.seqItem ANY >
<!ELEMENT XMI.sequence (seqItem)* >

<!ELEMENT XMI.arrayLen ANY >
```

```
<!ELEMENT XMI.arrayItem ANY >
<!ELEMENT XMI.array (XMI.arrayLen, XMI.arrayItem*) >

<!ELEMENT XMI.enum (#PCDATA) >

<!ELEMENT XMI.discrim ANY >
<!ELEMENT XMI.union (XMI.discrim, XMI.field*) >

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
        type CDATA #IMPLIED
>
```

For more information about these datatypes, refer to the MOF specification.

6.7 Metamodel Class Specification

This section describes in detail how to represent information about metamodel classes in a XMI compliant DTD. It uses the rules for generating a Hierarchical Entity DTD (Rule Set 3) as described in the "XML DTD Production" chapter to describe the manner in which attributes, associations, and containment relationships are represented in an XML DTD, and how inheritance between metamodel classes is handled. It uses a short example to explain the encoding.

The Hierarchical Entity DTD generation rules use the XML entity substitution technique extensively. The declaration of entities for commonly used information reduces the repetition of declarations used in multiple areas. They provide a single declaration point of frequently used information and allow regular formats for expressing copy-down inheritance in element declarations. Note that entities have no effect on the final form of the generated XML since they are always completely expanded out of the element definitions.

6.7.1 Class specification

Every metamodel class is decomposed into three parts: properties, associations, and compositions. Three entities are declared for every metamodel class, whose prefix is the name of the class and whose suffix is "Properties", "Associations", and "Compositions". The properties entity contains a list of the XML elements which correspond to metamodel attributes. The associations entity contains the XML elements representing roles of association ends. The compositions entity contains the XML elements which represent the role of associations that are aggregations.

The representation of a metamodel class named "c" is shown below for the simplest case where "c" does not have any attributes, associations, or containment relationships:

```
<!ENTITY % cProperties ''>

<!ENTITY % cAssociations ''>
```

```
<!ENTITY % cCompositions ‘’>

<!ELEMENT c (XMI.remoteContent?) >
<!ATTLIST c
    %XMI.ElementAttributes;
>
```

In the case where “c” has attributes, associations, and containment relationships for a metamodel class, the declaration is as follows:

```
<!ENTITY % cProperties ‘propertiesForC’ >

<!ENTITY % cAssociations ‘associationsForC’>

<!ENTITY % cCompositions ‘compositionsForC’>

<!ELEMENT c (XMI.remoteContent |
    (%cProperties,
    %cAssociations,
    %cCompositions)) >
```

If one of the three associated entities (cProperties, cAssociations, cCompositions) has no content, it is not included in the declaration of element “c” after XMI.remoteContent to maintain valid XML syntax. If none of the three entities have content, the or-bar and all the entities are suppressed, and the multiplicity of XMI.remoteContent is set to 0..1 with the ‘?’ symbol.

If the class is purely abstract, the declaration of the entire element may be suppressed because there are no instances of abstract classes to exchange. The three entities for the class must still be declared, however, because the abstract class may still have subclasses that will need the entity declarations of their superclass.

6.7.2 Inheritance Specification

XML does not currently have a built-in mechanism to represent inheritance. In its place, XMI specifies that inheritance will be copy-down inheritance. Inheritance is represented by using the required properties, associations, and compositions entities for each class.

For example, if a class “c1” has a direct superclass “c0” in the metamodel, then the declaration of the required entities for class “c1” is as follows:

```
<!ENTITY % c1Properties ‘%c0Properties; properties for c1, if any...’>

<!ENTITY % c1Associations ‘%c0Associations; associations for c1, if any...’ >

<!ENTITY % c1Compositions ‘%c0Compositions; compositions for c1, if any...’ >
```

Should there be a class, c2, derived from c1, then the entity declarations for c2 would be:

<!ENTITY % c2Properties '%c1Properties; *properties for c2, if any...*'>

<!ENTITY % c2Associations '%c1Associations; *associations for c2, if any...*' >

<!ENTITY % c2Compositions '%c1Compositions; *compositions for c2, if any...*' >

And so on down an inheritance hierarchy.

In this manner, the properties, associations, and compositions are copied directly from each superclass via the substitution capability of entities. Since XML requires entities to be declared in a DTD before being used, this method of representing inheritance requires that the entities of superclasses in a metamodel precede the declarations of entities and elements of their subclasses.

6.7.3 Attribute Specification

The representation of each attribute of metamodel class "c" uses XML elements instead of XML attributes. The reasons for this encoding choice are several, including: Elements may have more complex encodings than those allowed in XML attributes, 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 which apply only to element contents.

The declaration of each attribute named "a" with a non-enumerated type is as follows:

<!ELEMENT a (*type specification* | XMI.reference) >

The type specification for an element is usually one of the XMI-supplied common types. If the data is a string type, then its type is mixed, and the specification must take the form:

<!ELEMENT a (#PCDATA| XMI.reference)* >

When "a" is an attribute with enumerated values, a modified declaration is used to allow an XML processor to validate that the value of the attribute is one of the legal values of the enumeration.

<!ELEMENT a EMPTY >
<!ATTLIST a XMI.value (enum1 | enum2 | ...) #REQUIRED >

where enum1, enum2, … are are replaced with an entry for each member of the enumeration set.

For example, if a class is named "c" with attributes "a1" and "a2", where "a2" is Boolean, the attributes are represented as follows:

<!ELEMENT a1 (#PCDATA | XMI.reference) *>

<!ELEMENT a2 EMPTY >
<!ATTLIST a2 XMI.value (true | false) #REQUIRED >

<!ENTITY % cProperties 'a1, a2' >

Default values for attributes should not be specified in DTDs because XML allows the processor reading the document the option of not processing a DTD as an optional optimization. When tools skip processing the DTD, they do not obtain the default value of XML attributes.

6.7.4 Association Specification

Each association role is represented by an XML entity and an XML element. The multiplicity of the role must be translated into the XML multiplicities that are allowed. The representation of an association role named "r" with multiplicity "m" for a metamodel class "c" is:

<!ENTITY % cAssociations 'r*m*' >

<!ELEMENT r (XMI.reference) >

Each XML element representing the role of an association contains an "XMI.reference" element that should refer to an instance "c" or one of its subclasses at the end of the association link.

The valid multiplicities for "*m*" in XML are '?' if the multiplicity is zero or one, ' ' if the multiplicity is exactly one, '+' if the multiplicity is one or more, or '*' if the multiplicity is any other value. This set is the same as the those defined by the MOF, except that the '*' value is used to represent those values which cannot be represented in XML as well as the normal zero or more. For such multiplicities, the semantics of XML validation are not sufficient to enforce them.

6.7.5 Containment Specification

Each association end that represents containment is also represented by an XML entity and an XML element. The content model of the XML element representing the association end is the XML element corresponding to the class, and the XML elements corresponding to each of the subclasses of the class. If a class "c" is at the container end of an association link representing composition, and the other association end has role "r" with multiplicity "m" for a class "c1" with concrete subclass "c2", the representation in an XML DTD is as follows:

<!ELEMENT r (c1 | c2) >

<!ENTITY % cCompositions 'r*m*' >

Note that "*m*" is defined as in the previous subsection.

6.8 Document exchange with multiple tools

This section contains a recommendation for an optional methodology which can be used when multiple tools interchange documents. In this methodology, the ID 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 requirements by XML on IDs are that they are strings that must be unique within the document in which they are found. There may be a practical advantage to specifying a higher level of uniqueness in XMI.

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

6.8.1 Definitions:

General:

- MC - Model construct. A single unit of information to exchange using a single XML element tag. MCs may be nested, linked, etc.
- Extension - Extensions use the XMI.extension element. Extensions to MCs may be nested in MCs, linked to the Extension section(s) of the document, or linked outside the document. Each Extension should contain a tool-specific identifier. Extensions are considered private to a particular tool. An MC may have zero or more Extensions. Extensions may be nested.

IDs:

- XML ID - The public XML ID of an MC, expressed as the XML.ID attribute of an XML element tag. Example: <Class XML.id="ABCDEFGH">
- ToolID - The tool-specific ID of an MC. The ToolID is stored in an Extension of the MC when it differs from the XML ID.

Tool ID policies:

Every tool is either Open or Closed.

- Open Tool - A tool that will accept any XML ID as it's own.
- Closed Tool - A tool that will not accept an XML ID created by another tool. There must be an efficient algorithm for determining if a given XML ID is acceptable to the Closed Tool.

Tool Activities:

- Creating Tool - The tool that initially created the XMI document. There is one Creating Tool for the document.
- Importing Tool - The tool that is importing an XMI document.
- Modifying Tool - A tool writing an XMI document based on Importing a previous version of the document. The Modifying Tool for an MC may be indicated in an Extension. There may have been multiple Modifying Tools as several tools exchange information.

6.8.2 7.2 Procedures:

Document Creation:

- The Creating Tool writes a new XMI document. Each MC is assigned an XML ID that is identical to its ToolID.

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 XML IDs are accepted as the ToolID and no id conversion is needed.
2. If the Importing Tool is a Closed Tool, there are two options. If the MC contains an Extension with a ToolID from the Importing Tool, the Importing Tool uses that ToolID for the MC. Otherwise, the Importing Tool checks if the XML ID is acceptable and creates its own ToolID for the MC if the XML ID is unacceptable. The XML ID is stored internally for future merges if it differs from the ToolID.

1. Document Modification:

- The Modifying tool writes the MCs and any Extensions preserved from Import.
- For new MCs, the MC is assigned an XML ID that is identical to its ToolID.
- One of the following cases occurs for previously existing MCs:

1. If the XML ID is the same as the ToolID, the original XML ID is used. Extensions may be added for other tool-specific extensions.
2. If the XML ID differs the ToolID, the original XML ID is also used. An Extension is added to the MC containing the ToolID, the identity of the Modifying Tool, and other tool-specific extensions.

6.8.3 Example

This section describes a scenario in which Tool1 creates an XMI document which 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.

```
<Class name="c1" XMI.id="abcdefgh">
</Class>
```

2. The class is imported into Tool2. Tool2 assigns ToolID "JKLMNOPQRST". A second class is added with name "c2" and ToolID "X012345678"
3. The model is merged back to XMI:

```
<Class name="c1" XMI.id="abcdefgh">
```

```
    <XMI.extension>
        <XMI.exporter>Tool2</XMI.exporter>
        <XMI.exporterID>JKLMNOPQRST</XMI.exporterID>
    </XMI.extension>
</Class>
<Class name="c2" XMI.id="X012345678">
</Class>
```

4. The model is imported into Tool1. Tool1 assigns ToolID "ijklmnop" to "c2" and a new class "c3" is created with ToolID "qrstuvwxyz".

5. The model is merged back to XMI:

```
<Class name="c1" XMI.id="abcdefgh">
    <XMI.extension>
        <XMI.exporter>Tool2</XMI.exporter>
        <XMI.exporterID>JKLMNOPQRST</XMI.exporterID>
    </XMI.extension>
</Class>
<Class name="c2" XMI.id="X012345678">
    <XMI.extension>
        <XMI.exporter>Tool1</XMI.exporter>
        <XMI.exporterID>ijklmnop</XMI.exporterID>
    </XMI.extension>
</Class>
<Class name="c3" XMI.id="qrstuvwxyz">
</Class>
```

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

```
<Class name="c1" XMI.id="abcdefgh">
    <XMI.extension>
        <XMI.exporter>Tool2</XMI.exporter>
        <XMI.exporterID>JKLMNOPQRST</XMI.exporterID>
    </XMI.extension>
    <XMI.extension>
        <XMI.exporter>Tool3</XMI.exporter>
        <XMI.exporterID>s1234</XMI.exporterID>
    </XMI.extension>
</Class>
<Class name="c2" id="X012345678">
    <XMI.extension>
        <XMI.exporter>Tool1</XMI.exporter>
        <XMI.exporterID>ijklmnop</XMI.exporterID>
    </XMI.extension>
    <XMI.extension>
        <XMI.exporter>Tool3</XMI.exporter>
        <XMI.exporterID>s5678</XMI.exporterID>
    </XMI.extension>
```

```
</Class>
<Class name="c3" id="qrstuvwxyz">
    <XMI.extension>
        <XMI.exporter>Tool3</XMI.exporter>
        <XMI.exporterID>s90ab</XMI.exporterID>
    </XMI.extension>
</Class>
```

7. An open tool imports and modifies the file. There are no changes because the XML IDs are the same as its ToolIDs.

6.8.4 Alternatives

There are several advantages and disadvantages to this method of interchange.

Advantages:

- All Open Tools will be able to use XMI from multiple sources without Extensions for IDs.
- Extensions for IDs are not needed until two or more Closed Tools modify the document.
- The "namespace" or creator for each XML ID does not need to be specified.

Disadvantages:

- Closed tools that merge with existing documents will increase the overhead by adding their ToolID to every MC and all other tools will need to preserve this information.
- There is an assumption that IDs from multiple tools using different generation methods will still be unique.

Proposal variations:

- Establish XMI ID requirements, such as 128 bit globally unique ids. The advantage is that the ID is stronger and some tools already using such a format will become open. The disadvantage is that other tools, even those with more sophisticated IDs (256 bit, etc.), will become closed.
- Specify the creating tool for each MC so that the “namespace” of each XML ID can be determined.

6.9 8. UML DTD

Appendix A contains a DTD generated by hand that represents the UML metamodel. This DTD generally follows the specification of the above section on representing metamodel information. By examining this DTD, you can gain a better understanding of the types of metamodel information that can be represented in an XML DTD, and the information that cannot be specified.

The structure of the DTD closely corresponds to the document “UML Semantics version 1.1, 1 September 1997”. Each XML element corresponding to a class has a comment indicating which pages of that document describe the class. You can verify the accuracy of the DTD against the document by reading the pages of the document in the comments and verifying that the encoding for them is correct.

The DTD is organized according to the packages in the UML metamodel. For example, the Core package is presented first.

A DTD automatically generated from the MOF for UML using the Hierarchical Entity DTD generation rules (Rule Set 3) should closely resemble the example DTD, except that the example DTD uses an additional level of entity definition for elementary items such as attributes.

Considering the issues that arose from representing UML in an XML DTD, aided the development of this specification.

The UML DTD sample can also be used by tools which exchange UML information as a standard for importing and exporting UML metamodels. It can be used for that purpose even if the tools do not directly deal with the MOF.

Note that the UML DTD covers the UML semantics but not the UML notation. Additional work may address the issue of the UML diagrammatic information as an optional level of interchange.

XML DTD Production 7

7.1 Purpose

This section describes the rules for creating a DTD from a MOF-based metamodel. The DTD defined by the rules in this section describe the XML created by following the rules of Chapter 7, *XML Document Production*. While it uses the production rules in that section as a basis, it does not repeat them, since the rules for generation of a DTD are quite different than those for the generation of the corresponding XML.

Conformance to the XMI specification is not based on any DTD format. A conforming implementation of the rules in this section may implement any or all of these rule sets or may use its own when generating a DTD for a metamodel.

The rules specifications below do not cover fixed content which must be part of every DTD. This content is listed separately at the end of this chapter.

The rules are specified by a combination of EBNF, which serves as a syntactic framework, and rules written in pseudocode which embody the rules for producing the metasyntactic elements in the EBNF specification. The EBNF is extended slightly to account for the fact that XML DTD constructs are being generated. Since what is being defined is textual content, spaces are sometimes important. The “S” metasyntactic element should be understood to mean “at least one space”. This is at variance with standard EBNF, where spaces are usually ignored. In addition, the “Q” metasyntactic element is intended to indicate either a single quote or a double quote, either of which is valid in the XML DTD constructs generated using these rules. XML requires that the quotes used in this way must match, and if they enclose quoted strings, they must differ from the quotes used in the string.

A note on notation: Non-terminal symbols, (except for FixedContent) on the right hand side (RHS) of the productions below are prefixed by a number followed by a colon (“:”). These numbers are the production in which the non-terminal is defined. If there is no prefix on a RHS symbol, then the symbol is a variable whose value is defined in the rules following the EBNF production.

7.2 Rule Set 1: Simple DTD

The DTD for a MOF-based metamodel consists of a set of DTD definitions for each of the outermost packages in the metamodel.

7.2.1 Rules

1. DTD

A complete XMI DTD consists of fixed DTD content which is required for any XMI DTD, followed by at least one set of package DTD elements. The “XMI” element, defined in this fixed content, is the XML document root type for a valid XMI document. The elements defined in the package DTD elements can be placed in the content model of this root element.

Note: In the productions and pseudocode below, the use of ‘DTD’ as a suffix means a fragment of a DTD, not a complete DTD.

1. DTD ::= FixedContent PackageDTD+

To generate a DTD:

Generate initial fixed XMI definitions common to all MOF metamodel DTDs
Generate the DTD elements for each Metamodel Package for containment in the XMI.content element of XMI.

2. PackageDTD

A PackageDTD is a sequence of DTD elements of various types, reflecting the contents of the package.

2. PackageDTD ::= (PackageDTD | ClassDTD | AttributeElementDef |
RoleElementDef | CompositionDTD |
AssociationDTD)*
PackageElementDef

To Generate a PackageDTD:

For Each Classifier-scope non-derived Attribute of the Classes of the Package Do
Generate an AttributeElementDef for the Attribute
End
For Each non-derived Association of the Package which is a composition Do
Generate the CompositionDTD for the Association
End
For Each non-derived Association of the Package which is not a composition and neither of whose AssociationEnds is the target of a Reference Do
Generate the AssociationElementDef for the Association

End
If there are unreferenced, non-derived, non-composition Associations, Then
  Generate RoleElementDefs
End
For Each non-abstract Class of the Package Do
  Generate the ClassDTD for the Class
End
For Each Package of the Package Do
  Generate the PackageDTD for the sub Package
End
Generate the PackageElementDef for the Package

3. ClassDTD

A ClassDTD is a sequence of DTD fragments for the non-derived instance-scope attributes of a non-abstract class, followed by an element definition for the class itself..

3. ClassDTD::= (AttributeElementDef | ReferenceElementDef)* ClassElementDef

To Generate a ClassDTD:

For Each non-derived instance-scope Attribute of the Class Do
  Generate the AttributeElementDef for the Attribute
End
For Each non-derived Reference of the Class Do
  Generate the ReferenceElementDef for the Reference
End
Generate the ClassElementDef for the Class

4. AttributeElementDef

An AttributeElementDef is the XML element definition for an attribute. It gives the name and type (which may be a reference to a Class) for the attribute. If the Attribute is an enumerated type, the XML element has an attribute list (attlist).

4. AttributeElementDef ::= '<!ELEMENT' S AttribName S AttribContents '>'
('<!ATTLIST' S AttribName S Q
AttribAttListItems Q'>')?

To Generate an AttributeElementDef:

Set AttribName := the qualified name of the Attribute.
If the Attribute has a "type" which is a DataType Then
  If the DataType is Boolean or enum Then
    Set AttribContents := 'EMPTY'
    Set AttribAttListItems := 'value (' + the enumerated values, separated by "|" chars
      + ') #REQUIRED'
  Else If the Data Type is a string or encodable as a string Then
    Set AttribContents := '(#PCDATA | XMI.reference)*'

```
    Else If the Data Type is a struct Then
      Set AttribContents := '(XMI.struct | XMI.reference)'
    Else If the Data Type is a sequence Then
      Set AttribContents := '(XMI.sequence | XMI.reference)'
    Else If the Data Type is an array Then
      Set AttribContents := '(XMI.array | XMI.reference)'
    Else
      Set AttribContents := '(XMI.any)'
    End
  Else If the Attribute has Class type Then
    Set AttribContents := 'XMI.reference'
  End
  Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents
    and AttribAttlistItems.
```

5. ReferenceElementDef

The ReferenceElementDef for a Reference in a Class is the XML element definition for the Reference. It gives the name of the Reference.referencedEnd and indicates that it is a reference.

5. ReferenceElementDef ::= '<!ELEMENT' S RefName S RefContents '>'

To generate a ReferenceElementDef:

```
Set RefName := The qualified name of the AssociationEnd (role) referenced via the
                    "referencedEnd" reference of the Reference
Set RefContents := '(' + 'XMI.Reference' + ')'
Generate the !ELEMENT definition using RefName and RefContents
```

6. ClassElementDef

The ClassElementDef for a Class is the XML element definition for the Class. It gives the name of the Class and indicates the attributes, contained classes and references of the Class. Here, "contained classes" means, in addition to the classes actually in the Namespace of the class, those classes which are the types of the contained AssociationEnds (roles) of composition Associations which have this class as the containing class.

6. ClassElementDef ::= '<!ELEMENT' S ClassName S ClassContents '>'
'<!ATTLIST' S ClassName S Q ClassAttListItems Q'>'

To Generate a ClassElementDef:

```
Set ClassName := the qualified name of the Class
Set atts := GetInstanceLevelAttributes(this class, '')
Set refs := GetReferences(this Class, '')
If Length(refs) > 0 Then
```

```
      Set refs := refs + ‘,’
   End
   Set refs := refs + ‘XMI.extension*’
   Set comps1 := GetComposedRoles(this Class, ‘’)
   Set comps2 := GetContainedClasses(this Class, ‘’)
   Set ClassContents to match the pattern:
 ( atts ) , ( refs ) , ( ( comps1 ) | ( comps2 ) )*
Remove empty parentheses and resulting dangling commas from ClassContents
If Length(ClassContents) > 0 Then
        Set ClassContents := ‘ | (‘ + ClassContents + ‘)’
End
   Set ClassContents :=  ‘(‘ + ‘XMI.remoteContent’ + ClassContents + ‘)’
   Set ClassAttlistItems := ‘XMI.id ID #REQUIRED XMI.remote (true | false) "false"’
   Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents
      and ClassAttlistItems.
```

7. AssociationDTD

AssociationDTDs are defined only for unreferenced Associations. These are associations for which neither AssociationEnd is the Reference.referencedEnd of any Reference. The AssociationDTD contains only the AssociationElementDef and refers to the ‘role1’ and ‘role2’ AssociationEnd place holders..

7. AssociationDTD ::= AssociationElementDef

To generate an AssociationDTD:

```
Generate the AssociationElementDef
```

8. RoleElementDef

The RoleElementDef for a role in an unreferenced Association provides a means of holding the reference to the AssociationEnds of the Association. There are two of these in the DTD, one for ‘XMI.role1’ and one for ‘XMI.role2’.

8. RoleElementDef ::= ‘<!ELEMENT’ S RoleName S ‘(‘ ‘XMI.reference’ ‘)’ ‘>’

To Generate a RoleElementDef:

```
Set RoleName := ‘XMI.role1’
Generate the !ELEMENT definition using RoleName
Set RoleName := ‘XMI.role2’
Generate the !ELEMENT definition using RoleName
```

9. AssociationElementDef

The AssociationElementDef for an Association is the XML element definition for the Association. It gives the name of the association and the roles used in it..

9. AssociationElementDef ::= '<!ELEMENT' S AssnName S AssnContents '>' ('<!ATTLIST' S AssnName S Q AssnAttListItems Q'>')?

To Generate an AssociationElementDef:

Set AssnName := the qualified name of the Association
Set AssnContents := '(' + 'XMI.role1' + ',' + 'XMI.role2' + ')'
Set AssnAttListItems := 'XMI.id ID #REQUIRED XMI.remote (true | false) "false"'
Generate the !ELEMENT and !ATTLIST definitions using AssnName, AssnContents and AssnAttlistItems

10. CompositionDTD

The CompositionDTD is a set of DTD fragments for an Association which is a composition. It defines the class which is composed by the association directly plus any classes derived from it..

10. CompositionDTD ::= CompositionElementDef

11. CompositionElementDef

The CompositionElementDef for a composition is the XML element definition for the composition. It gives the name of the contained role and the classes which may take that role. It has no Attlist..

11. CompositionElementDef ::= '<!ELEMENT' S RoleName S CompContents '>'

To Generate a CompositionElementDef:

Set RoleName := the name of the role in the Association which is the contained role
Set CompContents := GetClasses(the class in the contained role)
Generate the !ELEMENT definition using RoleName and CompContents

12. PackageElementDef

The PackageElementDef gives the name of a package and indicates the contents of the package.

12. PackageElementDef ::= ‘<!ELEMENT’ S PkgName S PkgContents ‘>’
‘<!ATTLIST’ S PkgName S Q PkgAttListItems Q‘>’

To Generate a PackageElementDef

```
Set PkgName := the fully qualified name of the Package
Set atts := GetClassLevelAttributes(this Package)
Set atts2 := GetNestedClassLevelAttributes(this Package)
Set assns := GetAssociations(this Package)
Set classes := GetPackageClasses(this Package)
Set pkgs := GetContainedPackages(this Package)
Set PkgContents to match the pattern:
   ( atts ) , ( atts2) , ( assns | classes | pkgs ) *
Remove empty parentheses and any dangling commas from PkgContents
If Length(PkgContents) > 0 Then
   Set PkgContents := ‘(‘ + PkgContents + ‘)’
Else
   Set PkgContents := ‘EMPTY’
End
Set PkgAttlistItems := ‘XMI.id ID #REQUIRED XMI.remote (true | false) "false"’
Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents
   and PkgAttlistItems
```

7.2.2 Auxiliary functions

All of the auxiliary functions defined in this section are used in the Simple DTD rule set. Some are used in other rule sets.

GetInstanceLevelAttributes

The GetInstanceLevelAttributes function produces a string containing the names of all of the non-derived instance-scope Attributes of the class. This includes the Attributes defined in the class itself as well as those in the class(es) from which it is derived.

The “previousCls” parameter is used to avoid duplications of attributes due to multiple inheritance. It allows the attributes for a class to be entered into the result list only once.

```
Function GetInstanceLevelAttributes(in cls : Class, inout previousCls : String) Returns
String
   If cls appears in previousCls, return ‘’
   If cls has a parent Class Then
      Set parentAtts := GetInstanceLevelAttributes(parent Class, previous)
End
Set atts := ‘’
```

```
For Each Non-derived Instance-scope Attribute contained in cls Do
      Set name := Qualified name of the Attribute
      If the multiplicity of the Attribute is “1..*” Then
        Set m := ‘+’
      Else If the multiplicity of the Attribute is “0..1” Then
        Set m:= ‘?’
      Else If the multiplicity of the Attribute is not “1..1” Then
        Set m:= ‘*’
      Else
        Set m:= ‘’
      End
      If Length(atts) > 0 Then
        Set atts := atts + ‘,’
      End
      Set atts := atts + name + m
End
   If Length(parentAtts) > 0 and Length(atts) > 0) Then
      Set parentAtts := parentAtts + ‘,’
   End
   Add cls to previousCls
   Return parentAtts + atts
End
```

GetReferences

The GetReferences function gets all of the non-derived class References for the given class and the classes from which it is derived. An ordering is imposed: the references in the parent class must occur before the references of the current class. The references within a class may occur in any order.

The “previousCls” parameter is used to avoid duplications of references due to multiple inheritance. It allows the references for a class to be entered into the result list only once.

```
Function GetReferences(in cls : Class, inout previousCls: String) Returns String
   If cls appears in previousCls, return ‘’
   If cls has a parent Class Then
      Set parentRefs := GetReferences(parent Class)
End
Set refs := ‘’
For Each Reference contained in cls Do
      Set name := Qualified name of theReference
      If the multiplicity of the Reference is “1..*” Then
        Set m := ‘+’
      Else If the multiplicity of the Reference is “0..1” Then
        Set m := ‘?’
      Else If the multiplicity of the Reference is not “1..1” Then
        Set m := ‘*’
      Else
        Set m := ‘’
      End
      If Length(refs) > 0 Then
        Set refs := refs + ‘|’
      End
```

```
        Set refs := refs + Temp
    End
    If Length(refs) > 0 Then
        If Length(parentRefs) > 0 Then
          Set parentRefs := parentRefs + ‘,’
        End
        Set refs := ‘(‘ + refs + ‘)*’
    End
      Add cls to previousCls
      Return parentRefs + refs
    End
```

GetContainedClasses

The GetContainedClasses function returns a string describing the classes contained in a MOF class by means of the “Namespace-Contains-ModelElement” link only. It does not include the list of classes contained by composition.

The “previousCls” parameter is used to avoid duplications of contained classes due to multiple inheritance. It allows the contained classes to be entered into the result list only once.

```
Function GetContainedClasses(in cls : Class, inout previousCls : String) Returns String
  If cls appears in previousCls, return ‘’
  If cls has a parent Class Then
      Set parentClasses := GetContainedClasses(parent Class)
End
Set classes := ‘’
For Each  non-abstract Class contained in cls Do
      Set Temp := Qualified name of the Class
      If Length(classes) > 0 Then
        Set classes := classes + ‘|’
      End
      Set classes := classes + Temp
End
  If Length(parentClasses) > 0 and Length(classes) > 0) Then
      Set parentClasses := parentClasses + ‘,’
  End
  Add cls to previousCls
  Return parentClasses + classes
End
```

GetComposedRoles

The GetComposedRoles function returns a string describing the role names of classes “contained” by a MOF Class instance via composition associations. In this form of containment, a Class A is said to contain Class B by composition if there is an Association C for which one role (i.e. AssociationEnd instance), Rx, is an aggregation (composition) of the other role, Ry, and A inherits from the type of Rx and B inherits from Ry.

The “previousCls” parameter is used to avoid duplications of composed roles due to multiple inheritance. It allows the composed roles for a class to be entered into the result list only once.

```
Function GetComposedRoles(in cls : Class, inout previousCls : String) Returns String
  If cls appears in previousCls, return ‘’
  If cls has a parent Class Then
    Set parentRoles := GetComposedRoles(parent Class)
End
Set roles := ‘’
For Each non-derived Composition association for which this cls is the container Do
    Set name := Qualified name of the contained role
    If the multiplicity of the role is “1..*” Then
      Set m := ‘+’
    Else If the multiplicity of the role is “0..1” Then
      Set m := ‘?’
    Else If the multiplicity of the role is not “1..1” Then
      Set m := ‘*’
    Else
      Set m:= ‘’
    End
    If Length(roles) > 0 Then
      Set roles := roles + ‘|’
    End
    Set roles := roles + name + m
End
  If Length(parentRoles) > 0 and Length(roles) > 0) Then
    Set parentRoles := parentRoles + ‘|’
  End
  Add cls to previousCls
  Return parentRoles + roles
End
```

GetClasses

The GetClasses function returns a string containing the name of a class and all of the classes which are derived from it. This function is used in the creation CompositionElementDef element.

```
Function GetClasses(in cls : Class) Returns String
Set rslt :=  the name of this class
For Each subclass of cls Do
    Set Temp := GetClasses(the subclass)
    If (Length(Temp) > 0) Then
      Set Temp := Temp + ‘|’
    End
    Set rslt := rslt + Temp
End
Return rslt
End
```

GetClassLevelAttributes

The GetInstanceLevelAttributes function produces a string containing the names of all of the non-derived classifier-scope Attributes of the class. This includes the Attributes defined in the class itself as well as those in the class(es) from which it is derived.

```
Function GetClassLevelAttributes(in pkg : Package)  Returns String
  If pkg has a parent Package Then
    Set parentAtts := GetClassLevelAttributes(parent Package)
End
Set atts := ‘’
For Each classifier-scope Attribute contained in classes of pkg Do
    Set name := Qualified name of the Attribute
    If the multiplicity of the Attribute is “1..*” Then
      Set m := ‘+’
    Else If the multiplicity of the Attribute is “0..1” Then
      Set m := ‘?’
    Else If the multiplicity of the Attribute is not “1..1” Then
      Set m := ‘*’
    Else
      Set m := ‘’
    End
    If Length(atts) > 0 Then
      Set atts := atts + ‘,’
    End
    Set atts := atts + name + m
End
If Length(parentAtts) > 0 and Length(atts) > 0) Then
    Set parentAtts := parentAtts + ‘,’
  End
  Return parentAtts + atts
End
```

GetNestedClassLevelAttributes

The GetNestClassLevelAttributes function gets all of the non-derived class Attributes which have classifier scope for the classes of the given package and any packages which it contains.

```
Function GetNestedClassLevelAttributes(in pkg : Package) Returns String
Set rslt := ‘’
For Each classifier-scope Attribute contained in Classes of pkg Do
    Set name := Qualified name of the Attribute
    If the multiplicity of the Attribute is “1..*” Then
      Set m := ‘+’
    Else If the multiplicity of the Attribute is “0..1” Then
      Set  m := ‘?’
    Else If the multiplicity of the Attribute is not “1..1” Then
      Set m := ‘*’
    Else
      Set m := ‘’
    End
    If Length(rslt) > 0 Then
      Set rslt := rslt + ‘,’
```

```
      End
      Set rslt := rslt + name + m
End
  For Each Package of Pkg
      Set childAtts := GetNestedClassLevelAttributes(contained Package)
      If Length(childAtts) > 0 and Length(rslt) > 0 Then
        Set  := rslt + ‘,’
      End
      Set rslt := rslt + childAtts
End
Return rslt
End
```

GetAssociations

The GetAssociations function gets all of the unreferenced, non-derived, non-composition Associations in the given package and any packages from which it is derived.

```
Function GetAssociations(in pkg : Package) Returns String
  If pkg has a parent Package Then
      Set parentAssns := GetAssociations(parent Package)
End
Set assns := ‘’
For Each non-derived, unreferenced, non-composite Association of pkg Do
      Set Temp := Qualified name of the Association
      If Length(assns) > 0 Then
        Set assns := assns + ‘|’
      End
      Set assns := assns + Temp
End
If Length(parentAssns) > 0 and Length(assns) > 0) Then
      Set parentAssns := parentAssns + ‘|’
  End
  Return parentAssns + assns
End
```

GetPackageClasses

The GetPackageClasses function gets all of the non-abstract Classes in the given package and any packages from which it is derived.

```
Function GetPackageClasses(in pkg : Package) Returns String
  If pkg has a parent Package Then
      Set  parentClasses := GetPackageClasses(parent Package)
End
Set classes := ‘’
For Each  non-abstract Class of pkg Do
      Set Temp := Qualified name of the Class
      If Length(classes) > 0 Then
        Set classes := classes + ‘|’
      End
      Set classes := classes + Temp
```

```
End
If Length(parentClasses) > 0 and Length(classes) > 0) Then
      Set parentClasses := parentClasses + ‘,’
   End
   Return parentClasses + classes
End
```

GetContainedPackages

The GetContainedPackages function gets all of the Packages contained in the given package and any packages which it contains.

```
Function GetContainedPackages(this Package)
   If pkg has a parent Package Then
      Set parentPkgs := GetContainedPackages(parent Package)
End
Set pkgs := ‘’
For Each (sub) Package of pkg Do
      Set Temp := Qualified name of the subpackage.
      If Length(pkgs) > 0 Then
        Set pkgs := pkgs + ‘|’
      End
      Set pkgs := pkgs + Temp
End
If Length(parentPkgs) > 0 and Length(pkgs) > 0) Then
      Set parentPkgs := pkgs + ‘|’
   End
   Return parentPkgs + pkgs
End
```

7.3 Rule Set 2: Grouped entities

Although the productions in the previous rule set are very simple, they can result in large DTDs. This is due to the fact that the object contents and any enumerated attlist values are given for not only an object but for all of its supertypes. This means that element definitions might become quite large in a complex MOF metamodel. The set of rules in this section allow for the grouping of the parts of an object into XML ENTITY definitions. These entities may be used in place of the actual listing of the elements. This makes for more compact DTD files.

As in the Simple DTD rule set, The DTD for a MOF-based metamodel consists of a set of DTD definitions for the outermost packages in the metamodel.

7.3.1 Rules

1. DTD

A complete XMI DTD consists of fixed DTD content which is required for any XMI DTD, followed by the various package DTD elements. The document root type required by XML is defined in the fixed content. This root element is the “XMI”

element. The elements defined in the package DTD elements can be placed in the content model of this root element.

Note: In the productions and pseudocode below, the use of 'DTD' as a suffix means a fragment of a DTD, not a complete DTD.

1. DTD ::= FixedContent PackageDTD+

To generate a DTD:

Generate initial fixed XMI definitions common to all MOF metamodel DTDs
Generate the DTD elements for each Metamodel Package for containment in the XMI.content element of XMI.

2. PackageDTD

A PackageDTD is a sequence of DTD elements of various types, reflecting the contents of the package.

2. PackageDTD ::= (2:PackageDTD | 3:ClassDTD | 4:AttributeElementDTD | 13:RoleElementDef | 15:CompositionDTD | 12:AssociationDTD)* 17:PackageElementDef

To Generate a PackageDTD:

For Each Classifier-scope non-derived Attribute of the Classes of the Package Do
  Generate an AttributeElementDTD for the Attribute
End
For Each non-derived Association of the Package which is a composition Do
  Generate the CompositionElementDTD for the Association
End
For Each non-derived Association of the Package which is not a composition and neither of
  whose AssociationEnds is the target of a Reference Do
  Generate the AssociationElementDef for the Association
End
If there are unreferenced, non-derived, non-composition Associations, Then
  Generate RoleElementDefs
End
For Each class (both abstract and non-abstract) of the Package,
  starting at the topmost classes in the inheritance hierarchy(ies) Do
  Generate the ClassDTD for the Class
End
For Each Package of the Package Do
  Generate the PackageDTD for the Package
End
Generate the PackageElementDef for the Package

3. ClassDTD

A ClassDTD is a sequence of DTD fragments for the non-derived instance-scope attributes of the class and the references that it makes, followed by entity definitions that summarize this information. Unless the class is abstract, a ClassElementDef is also generated, in order to provide an XML element for the class data..

3. ClassDTD::= (AttributeElementDTD | ReferenceElementDef)* PropertiesEntityDef? RefsEntityDef? CompsEntityDef? ClassElementDef?

To Generate a ClassDTD:

```
For Each non-derived instance-scope Attribute of the Class Do
   Generate the AttributeElementDTD for the Attribute
End
For Each non-derived Reference of the Class Do
   Generate the ReferenceElementDef  for the Reference
End
If there are non-derived instance-scope Attributes for the Class, Then
   Generate the PropertiesEntityDef for the Class
End
If There are non-derived References for the Class, Then
   Generate the RefsEntityDef for the Class
End
If the Class contains other classes or if it is the containing class in a Composition, Then
   Generate the CompsEntityDef for the Class
End
If the Class is not abstract, Then
   Generate the ClassElementDef for the Class
End
```

4. AttributeElementDTD

An AttributeElementDTD is as sequence of DTD fragments for an attribute. These fragments include entity definitions for enumerated types and the AttributeElementDef items.

4. AttributeElementDTD ::= AttributeEntityDef? AttributeElementDef

To Generate an AttributeElementDTD:

```
If the Attribute has a “type” which is a Boolean or enum Then
   If an AttributeEntityDef for this type name has not previously been produced, Then
      Generate an AttributeEntityDef for this type
   End
End
Generate an AttributeElementDef for this Attribute
```

5. AttributeEntityDef

An AttributeEntityDef is an XML entity which specifies an enumerated set of values which an attribute may have.

5. AttributeEntityDef ::= '<!ENTITY' S '%' S TypeName Q 'XMI.value' '(' enumvalues ')'
'#REQUIRED' Q '>'

To Generate an AttributeEntityDef:

```
Set TypeName := the name of the Attribute type
Set enumvalues := ''
For Each possible value of the enumerated type Do
   If Length(enumvalues) > 0) Then
      Set enumvalues := enumvalues + '|'
   End
   Set enumvalues := enumvalues + the enumerated value
End
Generate the !ENTITY definition using TypeName and enumvalues
```

6. AttributeElementDef

An AttributeElementDef is the XML element definition for an attribute. It gives the name and type (which may be a reference to a Class) for the attribute.

6. AttributeElementDef ::= '<!ELEMENT' S AttribName S AttribContents '>'
('<!ATTLIST' S AttribName S Q AttribAttListItems Q'>')

To Generate an AttributeElementDef:

```
Set AttribName := the qualified name of the Attribute
If the Attribute has a "type" which is a DataType Then
   If the DataType is Boolean or enum Then
      Set AttribContents := 'EMPTY'
      Set TypeName := the name of the enumerated type or Boolean
      Set AttribAttListItems := '%' + TypeName + ';'
   Else If the Data Type is a string or encodable as a string Then
      Set AttribContents := '(#PCDATA | XMI.reference)*'
   Else If the Data Type is a struct Then
      Set AttribContents := '(XMI.struct | XMI.reference)'
   Else If the Data Type is a sequence Then
      Set AttribContents := '(XMI.sequence | XMI.reference)'
   Else If the Data Type is an array Then
      Set AttribContents := '(XMI.array | XMI.reference)'
   Else
      Set AttribContents := '(XMI.any)'
   End
Else If the Attribute has Class type Then
```

Set AttribContents := ‘XMI.reference’
End
Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents and AttribAttlistItems.

7. ReferenceElementDef

The ReferenceElementDef for a Reference in a Class is the XML element definition for the Reference. It gives the name of the Reference and indicates that it is a XMI reference.

7. ReferenceElementDef ::= ‘<!ELEMENT’ S RefName S RefContents ‘>’

To generate a ReferenceElementDef:

Set RefName := The qualified name of the reference
Set RefContents := ‘(‘ + ‘XMI.reference’ + ‘)’
Generate the !ELEMENT definition using RefName and RefContents

8. PropertiesEntityDef

The PropertiesEntityDef for a Class is an entity containing a list of the names and multiplicities of its instance-scope non-derived attributes.

8. PropertiesEntityDef ::= ‘<!ENTITY’ S ‘%’ S PropsEntityName Q PropsList Q ‘>’

To Generate a PropertiesEntityDef:

Set PropsEntityName := the name of the Class + ‘Properties’
Set PropsList := GetInstanceLevelAttributes2 (the Class)
Generate the !ENTITY definition using PropsEntityName and PropsList

9. RefsEntityDef

The RefsEntityDef for a Class is an entity containing a list of the names of its non-derived references.

9. RefsEntityDef ::= ‘<!ENTITY’ S ‘%’ S RefsEntityName Q RefsList Q ‘>’

To Generate a RefsEntityDef:

Set RefsEntityName := the name of the Class + ‘Associations’
Set RefsList := GetReferences2(the Class)
Generate the !ENTITY definition using RefsEntityName and RefsList

10. CompsEntityDef

The CompsEntityDef for a Class is an entity containing a list of the names its contained classes and composition roles..

10. CompsEntityDef ::= '<!ENTITY' S '%' S CompsEntityName Q CompsList Q '>'

To Generate a CompsEntityDef:

```
Set CompsEntityName := the name of the Class + 'Compositions'
Set CompsList := GetComposedRoles2(the Class)
Set Temp := GetContainedClasses2(the Class)
If Length(Temp) > 0 Then
   If Length(CompsList) > 0) Then
      Set CompsList := CompsList + ','
   End
End
Set CompsList := CompsList + Temp
Generate the !ENTITY definition using CompsEntityName and CompsList
```

11. ClassElementDef

The ClassElementDef for a Class is the XML element definition for the Class. It gives the name of the Class and indicates the attributes, contained classes and references of the Class. Here, "contained classes" means, in addition to the classes actually in the Namespace of the class, those classes which are the types of the contained AssociationEnds (roles) of composition Associations which have this class as the containing class.

Whereas the ClassElementDef in the Simple DTD rule set explicitly listed all of the attributes, references and compositions of the class, the ClassElementDef contents in this rule set is a list of the PropertiesEntityDefs, RefsEntityDefs and CompsEntityDefs of its own class and all of the classes from which it is derived..

11. ClassElementDef ::= '<!ELEMENT' S ClassName S ClassContents '>'
'<!ATTLIST' S ClassName S Q ClassAttListItems Q'>'

To Generate a ClassElementDef:

```
Set ClassName := the qualified name of the Class
Set props := GetPropertiesEntities2(this Class, '')
Set refs := GetRefsEntities2(this Class, '')
If Length(refs) > 0 Then
   Set refs := refs + ','
End
Set refs := refs + 'XMI.extension*'
Set comps := GetCompsEntities2(this Class, '')
Set ClassContents to match the pattern:
( atts ) , ( refs ) , ( comps )*
```

Remove empty parentheses and resulting dangling commas from ClassContents
If Length(ClassContents) > 0 Then
    Set ClassContents := ' | (' + ClassContents + ')'
End
Set ClassContents := '(XMI.remoteContent | ' + ClassContents + ')'
  Set ClassAttlistItems := 'XMI.id ID #REQUIRED XMI.remote (true|false) "false"'
  Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents
    and ClassAttlistItems.

12. AssociationDTD

AssociationDTDs are defined only for unreferenced Associations. These are associations for which neither AssociationEnd is the Reference.referencedEnd of any Reference. The AssociationDTD contains only the AssociationElementDef and refers to the 'role1' and 'role2' AssociationEnd place holders..

12. AssociationDTD ::= AssociationElementDef

To generate an AssociationDTD:

Generate the AssociationElementDef

13. RoleElementDef

The RoleElementDef for a role in an unreferenced Association provides a means of holding the reference to the AssociationEnds of the Association. There are two of these in the DTD, one for 'XMI.role1' and one for 'XMI.role2'..

13. RoleElementDef ::= '<!ELEMENT' S RoleName S '(' 'XMI.reference' ')' '>'

To Generate a RoleElementDef:

Set RoleName := 'XMI.role1'
Generate the !ELEMENT definition using RoleName
Set RoleName := 'XMI.role2'
Generate the !ELEMENT definition using RoleName

14. AssociationElementDef

The AssociationElementDef for an Association is the XML element definition for the Association. It gives the name of the association and the roles used in it..

14. AssociationElementDef ::= '<!ELEMENT' S AssnName S AssnContents '>'
('<!ATTLIST' S AssnName S Q
AssnAttListItems Q'>')?

To Generate an AssociationElementDef:

Set AssnName := the qualified name of the Association
Set AssnContents := '(' + 'XMI.role1' + ',' + 'XMI.role2' + ')'
Set AssnAttListItems := 'XMI.id ID #REQUIRED XMI.remote (true|false) "false"'
Generate the !ELEMENT and !ATTLIST definitions using AssnName, AssnContents and AssnAttlistItems

15. CompositionDTD

The CompositionDTD is a set of DTD fragments for an Association which is a composition. It defines the class which is composed by the association directly plus any classes derived from it..

15. CompositionDTD ::= CompositionElementDef

16. CompositionElementDef

The CompositionElementDef for a composition is the XML element definition for the composition. It gives the name of the contained role and the classes which may take that role. It has no Attlist.

16. CompositionElementDef ::= '<!ELEMENT' S RoleName S CompContents '>'

To Generate a CompositionElementDef:

Set RoleName := the name of the role in the Association which is the contained role
Set CompContents := GetClasses(the class which corresponds to RoleName)
Generate the !ELEMENT definition using RoleName and CompContents

17. PackageElementDef

The PackageElementDef gives the name of a package and indicates the contents of the package.

17. PackageElementDef ::= '<!ELEMENT' S PkgName S PkgContents '>'
'<!ATTLIST' S PkgName S Q PkgAttListItems Q'>'

To Generate a PackageElementDef

Set PkgName := the fully qualified name of the Package
Set atts := GetClassLevelAttributes(this Package)
Set atts2 := GetNestedClassLevelAttributes(this Package)
Set assns := GetAssociations(this Package)
Set classes := GetPackageClasses(this Package)
Set pkgs := GetContainedPackages(this Package)
Set PkgContents to match the pattern:

```
    ( atts ) , ( atts2) , ( assns | classes | pkgs ) *
  Remove empty parentheses and any dangling commas from PkgContents
  If Length(PkgContents) > 0 Then
    Set PkgContents := '(' + PkgContents + ')'
  Else
    Set PkgContents := 'EMPTY'
  End
  Set PkgAttlistItems := 'XMI.id ID #REQUIRED XMI.remote (true | false) "false"'
  Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents
    and PkgAttlistItems
```

7.3.2 Auxiliary functions

The following auxiliary functions are used in this rule set. They have a suffix of "2", which indicates that they are introduced in this rule set. Otherwise, the auxiliary functions are the same as in the Simple DTD rule set.

GetInstanceLevelAttributes2

The GetInstanceLevelAttributes2 function gets all of the non-derived class Attributes which have instance scope for the given class (only).

```
Function GetInstanceLevelAttributes2(in cls : Class) Returns String
Set rslt := ''
For Each Non-derived Instance-scope Attribute contained in cls Do
    Set name := Qualified name of the Attribute
    If the multiplicity of the Attribute is "1..*" Then
      Set m := '+'
    Else If the multiplicity of the Attribute is "0..1" Then
      Set m := '?'
    Else If the multiplicity of the Attribute is not "1..1" Then
      Set m := '*'
    Else
      Set m := ''
    End
    If Length(rslt) > 0 Then
      Set rslt := rslt + ','
    End
    Set rslt := rslt + name + m
End
  Return rslt
End
```

GetReferences2

The GetReferences2 function gets all of the non-derived class References for the given class (only). The references may occur in any order.

```
Function GetReferences2(in cls : Class) Returns String
Set rslt := ''
For Each Reference contained in cls Do
    Set name := Qualified name of the Reference.
```

```
        If the multiplicity of the Reference is “1..*” Then
          Set m := ‘+’
        Else If the multiplicity of the Reference is “0..1” Then
          Set  m := ‘?’
        Else If the multiplicity of the Reference is not “1..1” Then
          Set m := ‘*’
        Else
          Set m := ‘’
        End
        If Length(rslt) > 0 Then
          Set rslt := rslt + ‘|’
        End
        Set rslt := rslt + Temp
    End
      Return rslt
    End
```

GetContainedClasses2

The GetContainedClasses2 function returns a string describing the classes contained in a MOF class by means of the “Namespace-Contains-ModelElement” link only. It does not include the list of classes contained by composition.

```
Function GetContainedClasses2(in cls : Class) Returns String
Set rslt := ‘’
For Each  non-abstract Class contained in cls Do
      Set Temp := Qualified name of the Class.
      If Length(rslt) > 0 Then
        Set rslt := rslt + ‘|’
      End
      Set rslt := rslt + Temp
End
  Return rslt
End
```

GetComposedRoles2

The GetComposedRoles2 function returns a string describing the role names of classes “contained” by a MOF Class instance via composition associations. In this form of containment, a Class A is said to contain Class B by composition if there is an Association C for which one role (i.e. AssociationEnd instance), Rx, is an aggregation (composition) of the other role, Ry, and A inherits from the type of Rx and B inherits from Ry.

```
Function GetComposedRoles2(in cls : Class) Returns String
Set rslt := ‘’
For Each non-derived Composition association for which this cls is the container Do
      Set name := Qualified name of the contained role
      If the multiplicity of the role is “1..*” Then
        Set m := ‘+’
      Else If the multiplicity of the role is “0..1” Then
        Set m := ‘?’
      Else If the multiplicity of the role is not “1..1” Then
```

```
        Set m := '*'
      Else
        Set m := ''
      End
      If Length(rslt) > 0 Then
        Set rslt := rslt + '|'
      End
      Set rslt :=  rslt + name + m
  End
    Return rslt
  End
```

GetPropertiesEntities2

The GetPropertiesEntities2 function collects together a sequence of invocations of the PropertiesEntityDefs for the given class and the classes from which it is derived.

The "previousCls" parameter is used to avoid duplications due to multiple inheritance.

```
Function GetPropertiesEntities2(in cls: Class, inout previousCls : String) Returns String
  If cls appears in previousCls, return ''
  If cls has a parent Class Then
    Set parentProps := GetPropertiesEntities2(the parent Class) + ','
  End
  Set ClassName := the name of cls
  Set props := '%' + ClassName + 'Properties' + ';'
  Add cls to previousCls
  Return parentProps + props
End
```

GetRefsEntities2

The GetRefsEntities2 function collects together a sequence of invocations of the RefsEntityDefs for the given class and the classes from which it is derived.

The "previousCls" parameter is used to avoid duplications due to multiple inheritance.

```
Function GetRefsEntities2(in cls: Class, inout previousCls : String) Returns String
  If cls appears in previousCls, return ''
  If cls has a parent Class Then
    Set parentRefs := GetRefsEntities2(the parent Class) + ','
  End
  Set ClassName := the name of cls
  Set ref := '(' + '%' + ClassName + 'Associations' + ';' + ')*'
  Add cls to previousCls
  Return parentRefs + refs
End
```

GetCompsEntities2

The GetCompsEntities2 function collects together a sequence of invocations of the CompsEntityDefs for the given class and the classes from which it is derived.

The “previousCls” parameter is used to avoid duplications due to multiple inheritance.

```
Function GetCompsEntities2(in cls: Class, inout previousCls : String) Returns String
  If cls appears in previousCls, return ‘’
  If cls has a parent Class Then
    Set parentComps := GetCompsEntities2(the parent Class) + ‘|’
  End
  Set ClassName := the name of cls
  Set comps := ‘%’ + ClassName + ’Compositions’ + ’;’
  Add cls to previousCls
  Return parentComps + comps
End
```

7.4 Rule Set 3: Hierarchical Grouped entities

Although the productions in the previous rule set are more compact than the first, it still means the repetition of a number of entity names in each element definition. The set of rules in this section allows for the grouping of the parts of an object into entity definitions, as in the Grouped Entity rule set and adds the ability to group the usage of these definitions into hierarchies that reflect the generalization hierarchy(s) in the defined metamodel.

A more complete description of the design principles used in this Rule Set can be found in Section 5.7, *Metamodel Class Specification*.

This rule set requires somewhat more computational complexity than the Simple DTD rule set, but not more than in the Grouped Entity rule set. In particular, the DTD generation program must:

- Generate the entities for a class in inheritance order, i.e. starting at the topmost class(es) in any inheritance hierarchy(ies) and proceed downward, and
- Be able to keep a table of generated enumerated type entities in order to re-use them and avoid duplicate entity generation.

As in the Simple DTD and Grouped Entity rule sets, The DTD for a MOF-based metamodel consists of a set of DTD definitions for the outermost packages in the metamodel.

7.4.1 Rules

1. DTD

A complete XMI DTD consists of fixed DTD content which is required for any XMI DTD, followed by at least one set of package DTD elements. The “XMI” element, defined in this fixed content, is the XML document root type for a valid XMI document. The elements defined in the package DTD elements can be placed in the content model of this root element.

Note: In the productions and pseudocode below, the use of 'DTD' as a suffix means a fragment of a DTD, not a complete DTD.

1. DTD ::= FixedContent PackageDTD+

To generate a DTD:

Generate initial fixed XMI definitions common to all MOF metamodel DTDs
Generate the DTD elements for each Metamodel Package for containment in the XMI.content element of XMI.

2. PackageDTD

A PackageDTD is a sequence of DTD fragments of various types, reflecting the contents of the package.

2. PackageDTD ::= (PackageDTD | ClassDTD | AttributeElementDTD | RoleElemenDef | CompositionDTD | AssociationDTD)* PackageElementDef

To Generate a PackageDTD:

For Each Classifier-scope non-derived Attribute of the Classes of the Package Do
  Generate an AttributeElementDTD for the Attribute
End
For Each non-derived Association of the Package which is a composition Do
  Generate the CompositionElementDTD for the Association
End
For Each non-derived Association of the Package which is not a composition and neither of
  whose AssociationEnds is the target of a Reference Do
  Generate the AssociationElementDef for the Association
End
If there are unreferenced, non-derived, non-composition Associations, Then
  Generate RoleElementDefs
End
For Each class (both abstract and non-abstract) of the Package,
    starting at the topmost classes in the inheritance hierarchy(ies) Do
  Generate the ClassDTD for the Class
End
For Each Package of the Package Do
  Generate the PackageDTD for the Package
End
Generate the PackageElementDef for the Package

3. ClassDTD

A ClassDTD is a sequence of DTD fragments for the non-derived instance-scope attributes of the class and the references that it makes, followed by entity definitions

that summarize this information. Unless the class is abstract, a ClassElementDef is also generated, in order to provide an XML element for the class data..

3. ClassDTD ::= (AttributeElementDTD | ReferenceElementDef)* PropertiesEntityDef? RefsEntityDef? CompsEntityDef? ClassElementDef?

To Generate a ClassDTD:

For Each non-derived instance-scope Attribute of the Class Do
  Generate the AttributeElementDTD for the Attribute
End
For Each non-derived Reference of the Class Do
  Generate the ReferenceElementDef for the Reference
End
If there are non-derived instance-scope Attributes for the Class, Then
  Generate the PropertiesEntityDef for the Class
End
If There are non-derived References for the Class, Then
  Generate the RefsEntityDef for the Class
End
If the Class contains other classes or if it is the containing class in a Composition, Then
  Generate the CompsEntityDef for the Class
End
If the Class is not abstract, Then
  Generate the ClassElementDef for the Class
End

4. AttributeElementDTD

An AttributeElementDTD is as sequence of DTD fragments for an attribute. These fragments include entity definitions for enumerated types and the AttributeElementDef items.

4. AttributeElementDTD ::= AttributeEntityDef? AttributeElementDef

To Generate an AttributeElementDTD:

If the Attribute has a “type” which is a Boolean or enum Then
  If an AttributeEntityDef for this type name has not previously been produced, Then
    Generate an AttributeEntityDef for this type
  End
End
Generate an AttributeElementDef for this Attribute

5. AttributeEntityDef

An AttributeEntityDef is an XML entity which specifies an enumerated set of values which an attribute may have.

5. AttributeEntityDef ::= '<!ENTITY' S '%' S TypeName Q 'XMI.value' '(' enumvalues ')'
'#REQUIRED' Q '>'

To Generate an AttributeEntityDef:

```
Set TypeName := the name of the Attribute type
Set enumvalues := ''
For Each possible value of the enumerated type Do
   If Length(enumvalues) > 0) Then
      Set enumvalues := enumvalues + '|'
   End
   Set enumvalues := enumvalues + the enumerated value
End
Generate the !ENTITY definition using TypeName and enumvalues
```

6. AttributeElementDef

An AttributeElementDef is the XML element definition for an attribute. It gives the name and type (which may be a reference to a Class) for the attribute..

6. AttributeElementDef ::= '<!ELEMENT' S AttribName S AttribContents '>'
('<!ATTLIST' S AttribName S Q AttribAttListItems Q'>')?

To Generate an AttributeElementDef:

```
Set AttribName := the qualified name of the Attribute
If the Attribute has a "type" which is a DataType Then
   If the DataType is Boolean or enum Then
      Set AttribContents := 'EMPTY'
      Set TypeName := the name of the enumerated type or Boolean
      Set AttribAttListItems := '%' + TypeName + ';'
   Else If the Data Type is a string or encodable as a string Then
      Set AttribContents := '(#PCDATA | XMI.reference)*'
   Else If the Data Type is a struct Then
      Set AttribContents := '(XMI.struct | XMI.reference)'
   Else If the Data Type is a sequence Then
      Set AttribContents := '(XMI.sequence | XMI.reference)'
   Else If the Data Type is an array Then
      Set AttribContents := '(XMI.array | XMI.reference)'
   Else
      Set AttribContents := '(XMI.any)'
   End
Else If the Attribute has Class type Then
```

Set AttribContents := ‘XMI.reference’
End
Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents and AttribAttlistItems.

7. ReferenceElementDef

The ReferenceElementDef for a Reference in a Class is the XML element definition for the Reference. It gives the name of the Reference and indicates that it is a reference..

7. ReferenceElementDef ::= ‘<!ELEMENT’ S RefName S RefContents ‘>’

To generate a ReferenceElementDef:

Set RefName := The qualified name of the reference
Set RefContents := ‘(‘ + ‘XMI.reference’ + ‘)’
Generate the !ELEMENT definition using RefName and RefContents

8. PropertiesEntityDef

The PropertiesEntityDef for a Class is an entity containing a list of the names and multiplicities of its instance-scope non-derived attributes. It also contains an entity invocation which produces the names of the attributes from the class(es) from which it is derived...

8. PropertiesEntityDef ::= ‘<!ENTITY’ S ‘%’ S PropsEntityName Q PropsList Q ‘>’

To Generate a PropertiesEntityDef:

Set PropsEntityName := the name of the Class + ‘Properties’
Set PropsList := GetInstanceLevelAttributes3 (the Class, True)
Generate the !ENTITY definition using PropsEntityName and PropsList

9. RefsEntityDef

The RefsEntityDef for a Class is an entity containing a list of the names of its non-derived references. It also contains an entity invocation which produces the names of the references from the class(es) from which it is derived..

9. RefsEntityDef ::= ‘<!ENTITY’ S ‘%’ S RefsEntityName Q RefsList Q ‘>’

To Generate a RefsEntityDef:

Set RefsEntityName := the name of the Class + ‘Associations’
Set RefsList := GetReferences3(the Class, True)

Generate the !ENTITY definition using RefsEntityName and RefsList

10. CompsEntityDef

The CompsEntityDef for a Class is an entity containing a list of the names its contained classes and composition roles. It also contains an entity invocation which produces the names of the compositions from the class(es) from which it is derived..

10. CompsEntityDef ::= '<!ENTITY' S '%' S CompsEntityName Q CompsList Q '>'

To Generate a CompsEntityDef:

```
Set CompsEntityName := the name of the Class + 'Compositions'
Set CompsList := GetComposedRoles3(the Class, True)
Set Temp := GetContainedClasses3(the Class)
If Length(Temp) > 0 Then
   If Length(CompsList) > 0) Then
      Set CompsList := CompsList + ','
   End
End
Set CompsList := CompsList + Temp
Generate the !ENTITY definition using CompsEntityName and CompsList
```

11. ClassElementDef

The ClassElementDef for a Class is the XML element definition for the Class. It gives the name of the Class and indicates the attributes, contained classes and references of the Class. Here, "contained classes" means, in addition to the classes actually in the Namespace of the class, those classes which are the types of the contained AssociationEnds (roles) of composition Associations which have this class as the containing class.

Whereas the ClassElementDef in the Simple DTD rule set explicitly listed all of the attributes, references and compositions of the class, the ClassElementDef contents in this rule set is simply three entity invocations which contain all of this information..

11. ClassElementDef ::= '<!ELEMENT' S ClassName S ClassContents '>'
'<!ATTLIST' S ClassName S Q ClassAttListItems Q'>'

To Generate a ClassElementDef:

```
Set ClassName := the qualified name of the Class
Set props := '(' + '%' + ClassName + 'Properties' + ";" + ')'
Set refs := '(' + '%' + ClassName + 'Associations' + ';' + ',' + 'XMI.extension *' + ')'
Set comps := '(' + '%' + ClassName + 'Compositions' + ';' + ')*'
Set ClassContents := '( XMI.remoteContent | (' + props + ',' + refs + ',' + comps + ',' + '))'
Set ClassAttlistItems := 'XMI.id ID #REQUIRED XMI.remote (true|false) "false"'
Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents
```

and ClassAttlistItems.

12. AssociationDTD

AssociationDTDs are defined only for unreferenced Associations. These are associations for which neither AssociationEnd is the Reference.referencedEnd of any Reference. The AssociationDTD contains only the AssociationElementDef and refers to the ‘role1’ and ‘role2’ AssociationEnd place holders..

12. AssociationDTD ::= AssociationElementDef

To generate an AssociationDTD:

Generate the AssociationElementDef

13. RoleElementDef

The RoleElementDef for a role in an unreferenced Association provides a means of holding the reference to the AssociationEnds of the Association. There are two of these in the DTD, one for ‘XMI.role1’ and one for ‘XMI.role2’..

13. RoleElementDef ::= ‘<!ELEMENT’ S RoleName S ‘(‘ ‘XMI.reference’ ‘)’ ‘>’

To Generate a RoleElementDef:

Set RoleName := ‘XMI.role1’
Generate the !ELEMENT definition using RoleName
Set RoleName := ‘XMI.role2’
Generate the !ELEMENT definition using RoleName

14. AssociationElementDef

The AssociationElementDef for an Association is the XML element definition for the Association. It gives the name of the association and the roles used in it..

14. AssociationElementDef ::= ‘<!ELEMENT’ S AssnName S AssnContents ‘>’ (‘<!ATTLIST’ S AssnName S Q AssnAttListItems Q‘>’)?

To Generate an AssociationElementDef:

Set AssnName := the qualified name of the Association
Set AssnContents := ‘(‘ + The name of role 1 + ‘,’ + The name of role 2 + ‘)’
Set AssnAttListItems := ‘XMI.id ID #REQUIRED XMI.remote (true|false) “false”’
Generate the !ELEMENT and !ATTLIST definitions using AssnName, AssnContents and

AssnAttlistItems

15. CompositionDTD

The CompositionDTD is a set of DTD fragments for an Association which is a composition. It defines the class which is composed by the association directly plus any classes derived from it..

15. CompositionDTD ::= CompositionElementDef

16. CompositionElementDef

The CompositionElementDef for a composition is the XML element definition for the composition. It gives the name of the contained role and the classes which may take that role. It has no Attlist..

16. CompositionElementDef ::= '<!ELEMENT' S RoleName S CompContents '>'

To Generate a CompositionElementDef:

Set RoleName := the name of the role in the Association which is the contained role
Set CompContents := GetClasses(the class which corresponds to RoleName)
Generate the !ELEMENT definition using RoleName and CompContents

17. PackageElementDef

The PackageElementDef gives the name of a package and indicates the contents of the package..

17. PackageElementDef ::= '<!ELEMENT' S PkgName S PkgContents '>'
'<!ATTLIST' S PkgName S Q PkgAttListItems Q'>'

To Generate a PackageElementDef

```
Set PkgName := the fully qualified name of the Package
Set atts := GetClassLevelAttributes(this Package)
Set atts2 := GetNestedClassLevelAttributes(this Package)
Set assns := GetAssociations(this Package)
Set classes := GetPackageClasses(this Package)
Set pkgs := GetContainedPackages(this Package)
Set PkgContents to match the pattern:
   ( atts ) , ( atts2) , ( assns | classes | pkgs ) *
Remove empty parentheses and any dangling commas from PkgContents
If Length(PkgContents) > 0 Then
   Set PkgContents := '(' + PkgContents + ')'
Else
   Set PkgContents := 'EMPTY'
```

```
End
Set PkgAttlistItems := 'XMI.id ID #REQUIRED XMI.remote (true | false) "false"'
Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents
  and PkgAttlistItems
```

7.4.2 Auxiliary functions

The following auxiliary functions are used in this rule set. They have a suffix of "3", which indicates that they are introduced in this rule set. Otherwise, the auxiliary functions are the same as in the Simple DTD rule set.

GetParentClassNames3

The GetParentClassNames3 function returns the names of all of the parent classes of a class. This function is used for duplicate elimination in multiple inheritance

```
Function GetParentClassNames3(in cls : Class) Returns String
  set rslt := cls
  For each parent Class of cls Do
    Add GetParentClassNames3(the parent Class) to rslt
  End
  return rslt
End
```

GetInstanceLevelAttributes3

The GetInstanceLevelAttributes3 function gets all of the non-derived class Attributes which have instance scope for the given class and invokes an entity to include the attributes of the classes from which it is derived.

In the case of multiple inheritance, the algorithm falls back on an exhaustive listing of the attributes of the parent classes instead of invoking the parent Properties entity.

```
Function GetInstanceLevelAttributes3(in cls : Class, in needParents : Boolean) Returns
String
  If needParents Then
    If cls has no parent classes Then
      Set parentAtts := ''
    ElseIf cls has exactly one parent class, Then
      Set parentAtts := '%' + parent class name + 'Properties' + ';'
    Else
      Set names := ''
      For Each parent class of cls Do
        Add GetParentClassNames3(parent Class) to names
      End
      Eliminate duplicates from names
      For each class in names Do
        Set parentAtts := parentAtts + GetInstanceLevelAttributes3(class, False)
      End
    End
End
Set atts := ''
```

```
For Each Non-derived Instance-scope Attribute of cls Do
      Set name := Qualified name of the Attribute.
      If the multiplicity of the Attribute is “1..*” Then
        Set m := ‘+’
      Else If the multiplicity of the Attribute is “0..1” Then
        Set m := ‘?’
      Else If the multiplicity of the Attribute is not “1..1” Then
        Set m :=  ‘*’
      Else
        Set m := ‘’
      End
      If Length(atts) > 0 Then
        Set atts := atts + ‘,’
      End
      Set atts := atts + name + m
End
If Length(atts) > 0 Then
      Set parentAtts := parentAtts + ‘,’
End
   Return parentAtts + atts
End
```

GetReferences3

The GetReferences3 function gets all of the non-derived class References for the given class and the classes from which it is derived. An order is imposed: the parent class reference entity is invoked first; the references within the class can occur in any order.

In the case of multiple inheritance, the algorithm falls back on an exhaustive listing of the attributes of the parent classes instead of invoking the parent Properties entity.

```
Function GetReferences3(in cls : Class, in needParents : Boolean) Returns String
   If needParents Then
      If cls has no parent classes Then
        Set parentRefs := ‘’
      ElseIf cls has exactly one parent class, Then
        Set parentRefs := ‘%’ + parent class name + ‘Associations’ + ‘;’
      Else
        Set names := ‘’
        For Each parent class of cls Do
           Add GetParentClassNames3(parent Class) to names
        End
        Eliminate duplicates from names
        For each class in names Do
           Set parentRefs := parentRefs + GetReferences3(class, False)
        End
      End
End
Set refs := ‘’
For Each Reference contained in cls Do
      Set name := Qualified name of theReference.
      If the multiplicity of the Reference is “1..*” Then
        Set m := ‘+’
      Else If the multiplicity of the Reference is “0..1” Then
```

```
        Set  m:= '?'
      Else If the multiplicity of the Reference is not "1..1" Then
        Set m := '*'
      Else
        Set m := ''
      End
      If Length(refs) > 0 Then
        Set refs := refs + '|'
      End
      Set refs := refs + name + m
End
If Length(refs) > 0 Then
      Set parentRefs := parentRefs + ','
      set refs := '(' + refs + ')*'
End
  Return parentRefs + refs
End
```

GetContainedClasses3

The GetContainedClasses3 function returns a string describing the classes contained in a MOF class by means of the "Namespace-Contains-ModelElement" link only. It does not include the list of classes contained by composition. Note that, since GetContainedClasses3 and GetComposedRoles3 are intended to contribute to the same entity, then only GetComposedRoles3 refers to the parent class to obtain the composition entity.

```
Function GetContainedClasses3(in cls : Class) Returns String
Set rslt := ''
For Each  non-abstract Class contained in cls Do
      Set Temp := Qualified name of the Class.
      If Length(rslt) > 0 Then
        Set rslt := rslt + '|'
      End
      Set rslt := rslt + Temp
End
  Return rslt
End
```

GetComposedRoles3

The GetComposedRoles3 function returns a string describing the role names of classes "contained" by a MOF Class instance via composition associations. In this form of containment, a Class A is said to contain Class B by composition if there is an Association C for which one role (i.e. AssociationEnd instance), Rx, is an aggregation (composition) of the other role, Ry, and A inherits from the type of Rx and B inherits from Ry.

```
Function GetComposedRoles3(in cls : Class, in needParents : Boolean) Returns String
  If needParents Then
      If cls has no parent classes Then
        Set parentComps := ''
      ElseIf cls has exactly one parent class, Then
```

```
        Set parentComps := '%' + parent class name + 'Compositions' + ';'
      Else
        Set names := ''
        For Each parent class of cls Do
          Add GetParentClassNames3(parent Class) to names
        End
        Eliminate duplicates from names
        For each class in names Do
          Set parentComps := parentComps + GetComposedRoles3(class, False)
        End
      End
End
Set comps := ''
For Each non-derived Composition association for which this cls is the container Do
      Set name := Qualified name of the contained role
      If the multiplicity of the role is "1..*" Then
        Set m := '+'
      Else If the multiplicity of the role is "0..1" Then
        Set m:= '?'
      Else If the multiplicity of the role is not "1..1" Then
        Set m := '*'
      Else
        Set m := ''
      End
      If Length(comps) > 0 Then
        Set comps := comps + '|'
      End
      Set comps := comps + Temp
End
If Length(comps) > 0 Then
      Set parentComps := parentComps + '|'
End
   Return parentComps + comps
End
```

7.5 Fixed DTD elements

There are some elements of the DTD which are fixed, constituting a form of "boilerplate" necessary for every MOF DTD. These elements are described in this section. They should be included in the generated DTD file in the locations indicated. Though, as elements, these need not be at the beginning of the DTD, the convention is to place them there.

The use of these fixed content elements means that any DOCTYPE declaration in an XMI-conformant transfer text should reference "XMI" as its root element. The "XMI" element includes the "XMI.content" element, which contains the actual transferred data. The content model of "XMI.content" then allows the transferred data to have any element as its effective root element.

Only the DTD content of the fixed elements is given here. For a complete description of the semantics of these elements, see Section 5.6, *Common XMI DTD Declarations*.

The FixedContent elements are:

```
<!ELEMENT XMI (XMI.header, XMI.content, XMI.extensions*) >
<!ATTLIST XMI
        xmi-version CDATA #FIXED "1.0"
        timestamp   CDATA #IMPLIED
        verified (true | false) #IMPLIED >


<!ELEMENT XMI.header (XMI.documentation?, XMI.metamodel+) >

<!ELEMENT XMI.documentation (#PCDATA |
                XMI.owner | XMI.contact |
                XMI.long_description |
                XMI.short_description | XMI.exporter |
                XMI.exporter_version | XMI.notice)* >

<!ELEMENT XMI.owner ANY >
<!ELEMENT XMI.contact ANY >
<!ELEMENT XMI.longDescription ANY >
<!ELEMENT XMI.shortDescription ANY >
<!ELEMENT XMI.exporter ANY >
<!ELEMENT XMI.exporterVersion ANY >
<!ELEMENT XMI.exporterID ANY >
<!ELEMENT XMI.notice ANY >

<!ELEMENT XMI.metamodel ANY>
<!ATTLIST XMI.metamodel
         name    CDATA #REQUIRED
         version CDATA #REQUIRED
         href    CDATA #IMPLIED >
<!ELEMENT XMI.content ANY >
<!ELEMENT XMI.extensions ANY >

<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
        target IDREF #IMPLIED
        href   CDATA #IMPLIED
        expectedType CDATA #IMPLIED >

<!ELEMENT XMI.field ANY >
<!ELEMENT XMI.struct (field)+ >

<!ELEMENT XMI.seqItem ANY >
<!ELEMENT XMI.sequence (seqItem)* >

<!ELEMENT XMI.arrayLen ANY >
<!ELEMENT XMI.arrayItem ANY >
<!ELEMENT XMI.array (XMI.arrayLen, XMI.arrayItem*) >

<!ELEMENT XMI.enum (#PCDATA) >

<!ELEMENT XMI.discrim ANY >
<!ELEMENT XMI.union (XMI.discrim, XMI.field*) >

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
```

```
        type CDATA #IMPLIED >

<!ELEMENT XMI.extension ANY >

<!ELEMENT XMI.remoteContent (XMI.reference) >
```

XML Generation Principles 8

8.1 Purpose

This section describes the manner in which XML Documents are generated to represent models. The subsequent section specifies the specific rules that XMI uses in this generation process.

8.2 Introduction

XMI defines the manner in which a model will be represented as an XML document. For a given model, each XMI-conforming implementation will produce an equivalent XML document.

XML document production is defined as a set of rules, which when applied to a model or model elements, produce an XML document. These rules can be applied to any model whose metamodel can be described by the MOF. This section provides an informal description of the production of XML documents from models. Although it may appear from this description that XML production should be performed using certain algorithms, interfaces, or facilities, any implementation which produces XML equivalent to the XML produced by the application of the specified production rules complies with XMI. The specific rules, and the specification of XML document equivalence is provided in Section 9, *XML Document Production* on page 109.

8.3 Two Model Sources

XMI can be applied to any model whose metamodel can be described by the MOF. However, the MOF meta-metamodel does not require any specific construct or mechanism to be used to define, in a metamodel, what will constitute a model. This approach allows metamodelers greatest flexibility. XMI is not able to identify, for any metamodel, what will constitute a model. Therefore XMI, to provide greater flexibility in exchanging model information, provides two distinct methods of specifying the modeling elements which are used to generate an XML document.

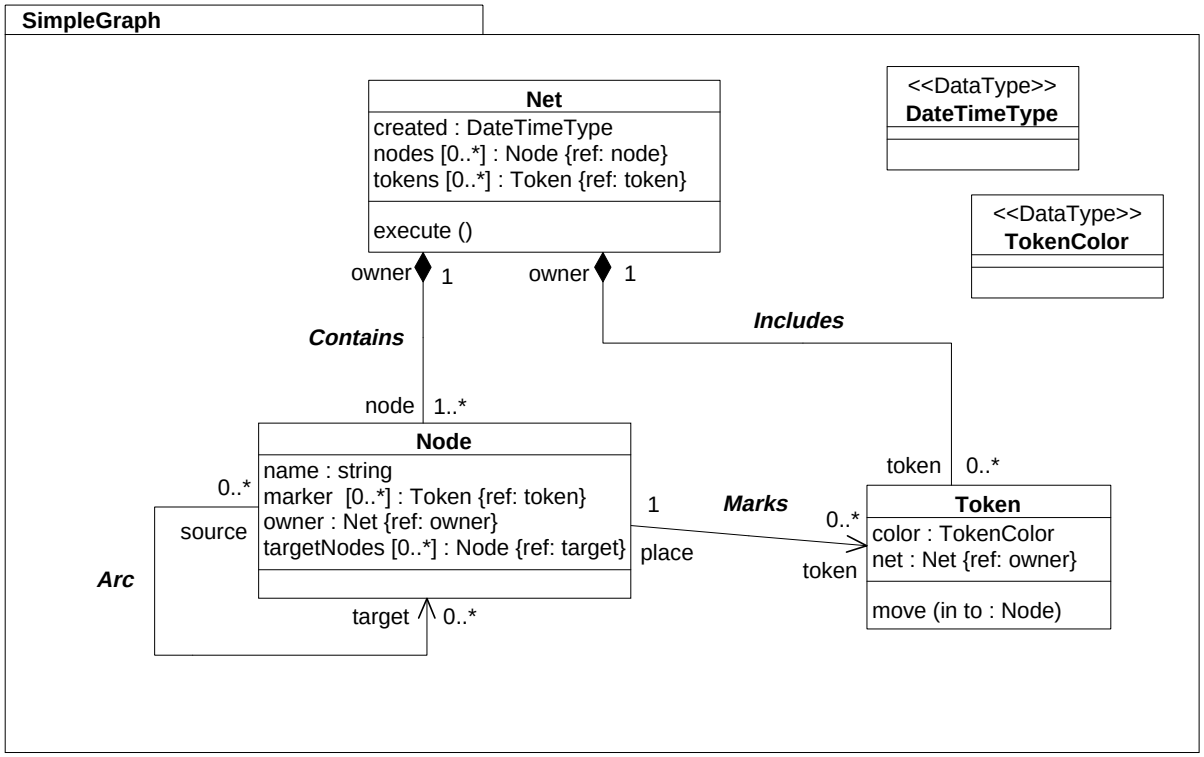

Figure 8-1 A very simple metamodel for graph modeling

8.3.1 Production by Object Containment

Most metamodels are characterized by a composition hierarchy. Modeling elements of some type are composed of other modeling elements. In UML, a Model is composed of Classes, UseCases, Packages, etc. Those elements in turn of composed of other elements. This composition is defined in metamodels using the MOF's composite form of Association. This composition must obey strict containment – an element cannot be contained in multiple compositions. To support models and model fragments as compositions, XMI provides for XML document production by object containment. Given a composite object, XMI's rules define the XML document which represents the composite object and all the contained objects in the composition hierarchy.

Consider a simple example. A very simple metamodel defines a language or set of constructs for developing graphs. The modeling elements Net, Node, Arc, and Token, and a supporting data type are defined. Figure 8-1 on page 94 shows this metamodel in UML notation. The metamodel is defined using the MOF Model. The MOF Model instances which compose the SimpleGraph metamodel are shown in Figure 8-2 on page 95 (which much detail omitted).

Since this metamodel is expressed via the MOF, its model instances can be represented in XML using the XMI generation rules. A simple model is shown in some net

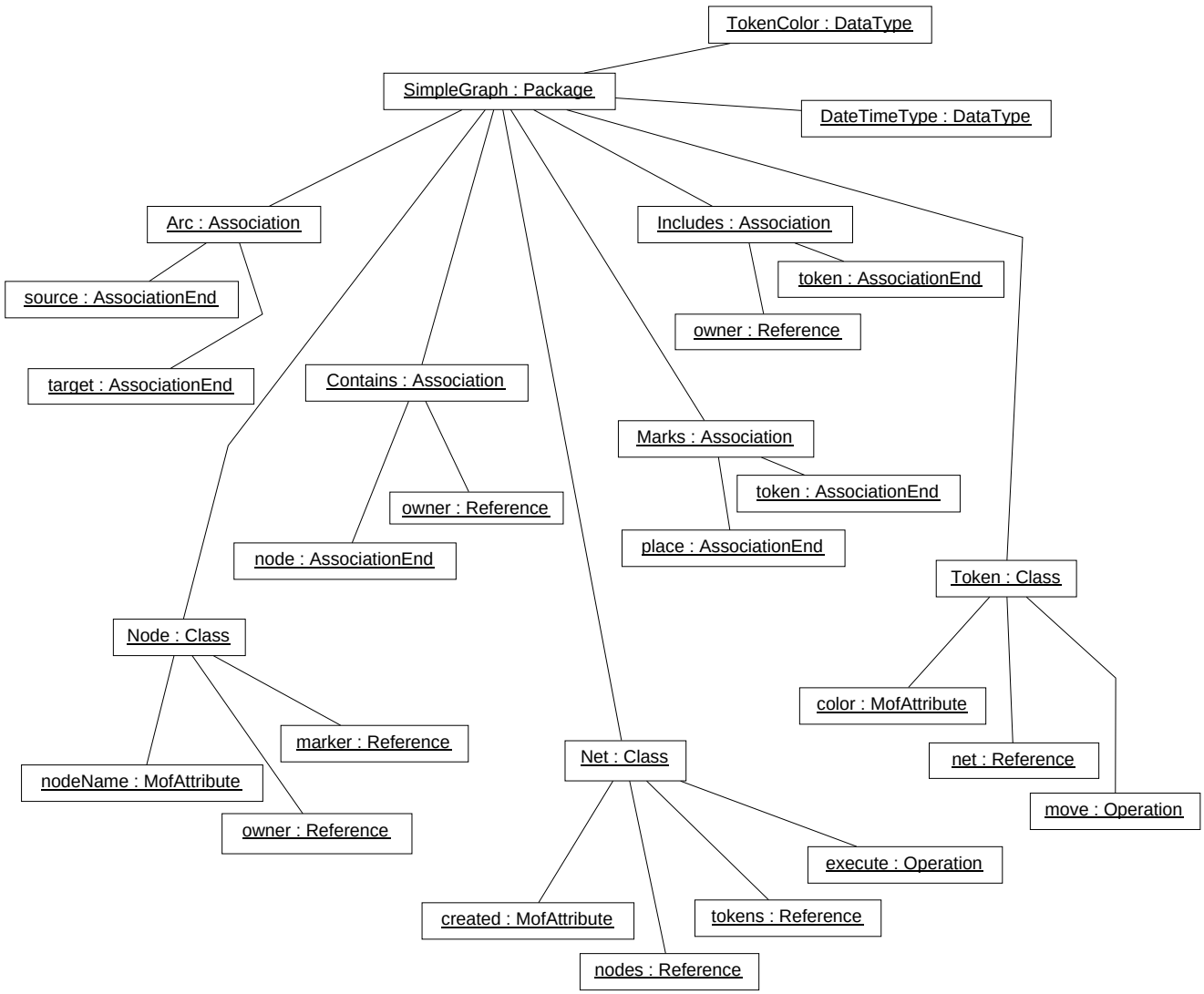

Figure 8-2 Object diagram showing simple metamodel as an instance of the MOF Model

notation in Figure 8-3 on page 96. As instances of the metamodel elements, the same model would form the object diagram in Figure 8-4 on page 96.

The XML production rules for Production by Object Containment are applied to a single root object of a composition. In this example, the rules are applied to the Net instance, to form the XML document representing this model. The rules are applied throughout the composition hierarchy by navigating through the composition links. In addition, the rules make use of the model's metamodel to represent the types of the values.

Each generated XML document begins with a prologue and the standard enclosing XML element's start tag. This part of the generation process is Specified in Section 9,

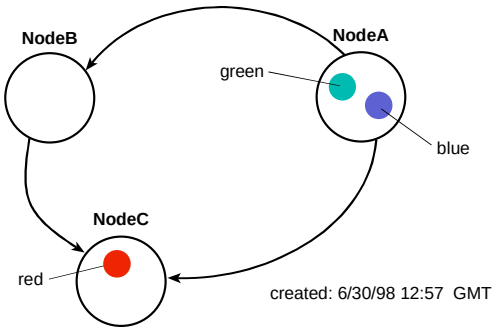

Figure 8-3 Example Net as a model of the SimpleGraph metamodel

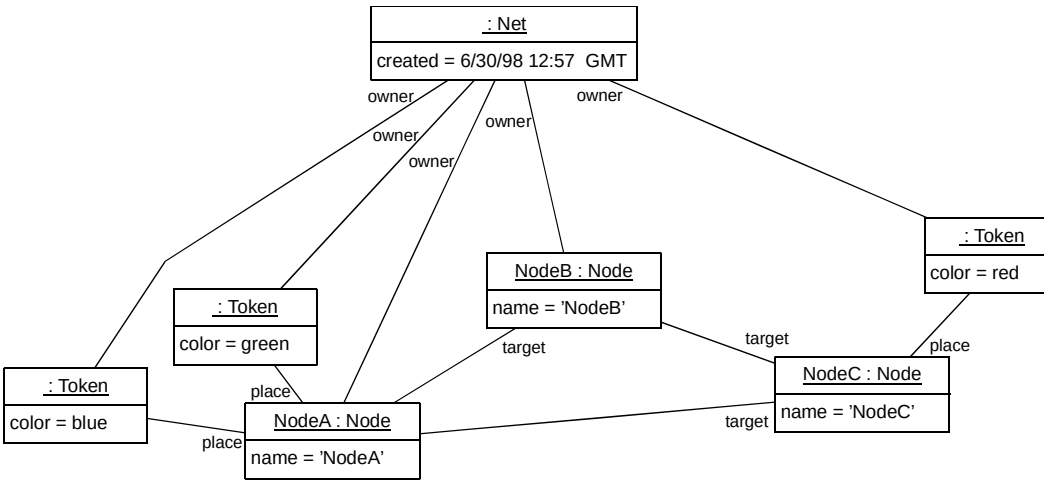

Figure 8-4 Objects forming the example SimpleGraph model

XML Document Production on page 109. Section 9.6, *Necessary XMI DTD Declarations* on page 105 describes the standard elements placed in the front of each XMI document Next comes the actual model, starting with the root object. For each object, including this root object, the element start tag is generated from the object's metaclass name. In this example, it is:

```
<Net XMI.id='a1'>
```

The element attribute XMI.id provides a unique identifier with the document for this element. Note that it may not be possible to simply use the metaclass name as the element type name. The element type's namespace is flat, while the metamodel namespace (as defined by the MOF), is hierarchical. Optionally, the element type is

defined using the qualified name via a dot notation. In the face of element type name collisions, this qualified name must be used. So the element tag could also be:

```
<SimpleGraph.Net XMI.id='a1'>
```

Next each attribute of the current object is used to generate XML. The attribute is enclosed in an element, defined by the name of the attribute, as found in the metamodel:

```
<created>
```

Next the attribute value is written out as XML. In the example, the attribute is of type DateTimeType, as defined in the metamodel. The details of that datatype were not shown above. DateTimeType is a struct with two fields, time, of type long, and timezone, of type string. The representation of struct values uses field tags as delimiters:

```
<field>1873852</field>
<field>GMT</field>
```

Then the attribute is completed with the corresponding end tag:

```
</created>
```

In the case of this object, each component, or contained, object is included. A component object is an object linked to the composite object via a link defined in the metamodel as a composite association, with the composite end corresponding to link end of the composite object. In this example, there is a total of three Node objects and three Token objects contained by the Net object. Similar to how attribute values are represented, contained objects are represented as the contents of reference elements.

The MOF supports the use of References in defining metamodels. A reference provides the object's navigability to linked objects. In this example, the Net's nodes reference will produce:

```
<nodes>
```

to indicate the referenced nodes it contains. Then, for each Node, the process of producing XML to represent an object is repeated. For the example, the Node with the name NodeA is written out in XML, starting with the element start tag:

```
<Node XMI.id='a2'>
```

the value of the element attribute id can be any unique value which is XML-compliant. Just as before, all the attribute values are written out first. The node class defines the attribute name; for this Node instance, the XML is:

```
<name>NodeA<name>
```

Next the non-composite, non-component references are written out. These are the references defined by Associations which are not defined as composites at either end. Since the Node class defines the Reference marker, and NodeA has markers, the XML generated is:

```
<markers>
```

```
    <XMI.reference target='a5' />
    <XMI.reference target='a6' />
  </markers>
```

The two target element attribute values correspond to two elements representing Token objects which have not yet been written. When the XLink/XPointer work becomes a W3C recommendation, the alternate href attribute may be used instead of target. See Section 6.6.10, *XMI.reference* on page 43, for a discussion on references.

Next, the value of the Node's targetNodes reference is written out as XML:

```
  <targetNodes>
    <XMI.reference target='a3' />
    <XMI.reference target='a4' />
  </targetNodes>
```

Finally, for NodeA, any contained objects are written out. But since The Node class does not define Node as a composite, this step is skipped. The XML for NodeA is complete:

```
</Node>
```

This process is repeated for the other values of the Net's nodes reference, NodeB and NodeC:

```
<Node XMI.id='a3'>
  <name>NodeB<name>
  <targetNodes>
    <XMI.reference target='a3' />
  </targetNodes>
</Node>
<Node XMI.id='a4'>
  <name>NodeC<name>
  <markers>
    <XMI.reference target='a7' />
  </markers>
</Node>
```

Notice that for NodeB, the markers reference element is omitted. When there are no values for a Reference or Attribute, the element tag may be omitted. Likewise the target reference element is absent for NodeC. The composite reference nodes is now fully represented, as is completed in the XML with a corresponding end tag:

```
</nodes>
```

Next the Token objects contained via the tokens Reference of Net are written out as XML:

```
<tokens>
```

Each Token object is written out as the other objects, starting with the attributes. Although not shown in the example, the TokenColor data type is an enumeration. Attributes whose types are enumerations or boolean are represented is a special

manner. Their value is represented as an element attribute value, to increase XML parser validation.

```
<Token XMI.id='a5'>
  <color XMI.value='green' />
</Token>
```

Since the value of the attribute is encoded in the tag of the empty element, separate end tag is not used.The Token class is defined with the single attribute. If were derived from a supertype, the values of attributes and references defined in the supertype would also be written out as XML. Like the Node class, the Token class has no composite references. The single reference defined for token provides the value of the owner, the Net object acting as the component in the composite link. These kinds of references are not written out to the XML document.

The remaining Tokens from the Net's tokens reference yield:

```
<Token XMI.id='a6'>
  <color XMI.value='blue' />
</Token>
<Token XMI.id='a7'>
  <color XMI.value='red' />
</Token>
</tokens>
```

At this point, all the values that make up the model have been written out as XML. The Net object is completed with the end tag:

```
</Net>
```

All this XML will be embedded in the standard XML element, as described later. Also, sometimes object links will not be represented via references, and need to be represented in XML after the root element. For this simple model though, no unrepresented links remain.

8.3.2 MOF's Role in XML Production

The specific generation rules rely on a MOF definition of the model's metamodel. It would simply not be possible to define meaningful production rules that would work on any arbitrary model, regardless of its metamodel. The single meta-metamodel provides the commonality among models, allowing the metamodel information to be uniformly represented. In addition, the MOF defines standard interfaces for the model elements of instances of MOF-defined metamodels. These interfaces – from the MOF's Reflective module – provide for access to an object's metaclass, attribute values, and reference values, among other capabilities. The operations of these interfaces provide an unambiguous means of specifying the access of model elements' metamodel and values.

In order for a metamodel to have its models interchanged through XMI, that metamodel must be representable through the MOF, as an instance of the MOF Model. However, this submission does not actually require an implementation to make use of a MOF, the MOF-defined Reflective interfaces, or even have metamodels represented

as instances of the MOF model. The implementation must, however, conform to the generation rules. These rules are based on the metamodels defined via the MOF and the use of the operations in the Reflective interfaces.

8.3.3 Production by Package Extent

It may not always be possible or useful to represent a desired set of modeling elements through a composition hierarchy. For this reason, XMI defines a second set of rules for generating XML from modeling elements.

The MOF provides the Package element in support of metamodel development. At the metamodel level, Package objects are always the top-most (uncontained) elements. A Package will contain Classes and Associations, directly and possibly through nested Packages. In the IDL generated from a MOF metamodel, interfaces represent specific features of these Packages, Classes, and Associations, in the use of model development. For each Package, there is a corresponding subtype of RefPackage, an interface in the MOF's Reflective module. Likewise, for each Class, there is a corresponding subtype of RefObject, and for each Association, a corresponding subtype of RefAssociation.

These interfaces define a structure which mirrors the metamodel structure. So the RefPackage subtype corresponding to the top-level Package in the metamodel contains all the other RefPackages, RefObjects, and RefAssociations. Each RefObject subtype object can provide all of the current objects of the class it represents; each RefAssociation subtype object can provide all the links corresponding to the Association it represents. The Package Extent, then, is the top-level RefPackage subtype object, all the RefPackage, RefObject and RefAssociation subtype objects it contains, and all the objects and links associated with them.

In this example, the IDL generation creates interfaces SimpleGraphPackage, NetClass, NodeClass, TokenClass, and Arc. Figure 8-5 on page 101 shows some of the interfaces generated for the example SimpleGraph metamodel. Suppose two different Nets were modeled, with an Arc crossing from one net to the next, as shown in Figure 8-6 on page 102. These nets are shown in Figure 8-7 on page 103, as instances of the SimpleGraph metamodel. The dashed lines in that figure represent the extent the NetClass, NodeClass, and TokenClass. The extent of the SimpleGraphPackage includes those extents.

The rules for XML Production by Package Extent act upon the uncontained RefPackage instance, producing an XML document which represents all the elements in the extent of that RefPackage. In the example, the rules are applied to the SimpleGraphPackage instance.

The same XML document prologue and enclosing element is required as was for Production by Object Containment. Then, the SimplePackageClass is traversed. For each RefObject instance, the extent is examined. Any object which is not participating as a component in a composition link becomes the starting point for generating XML. For instance, from the NodeClass, all Node instances can be accessed. But since all are at the component end of a composition link, none are used in XML production. When the NetClass is accessed, though, each of the two objects in its extent are uncontained

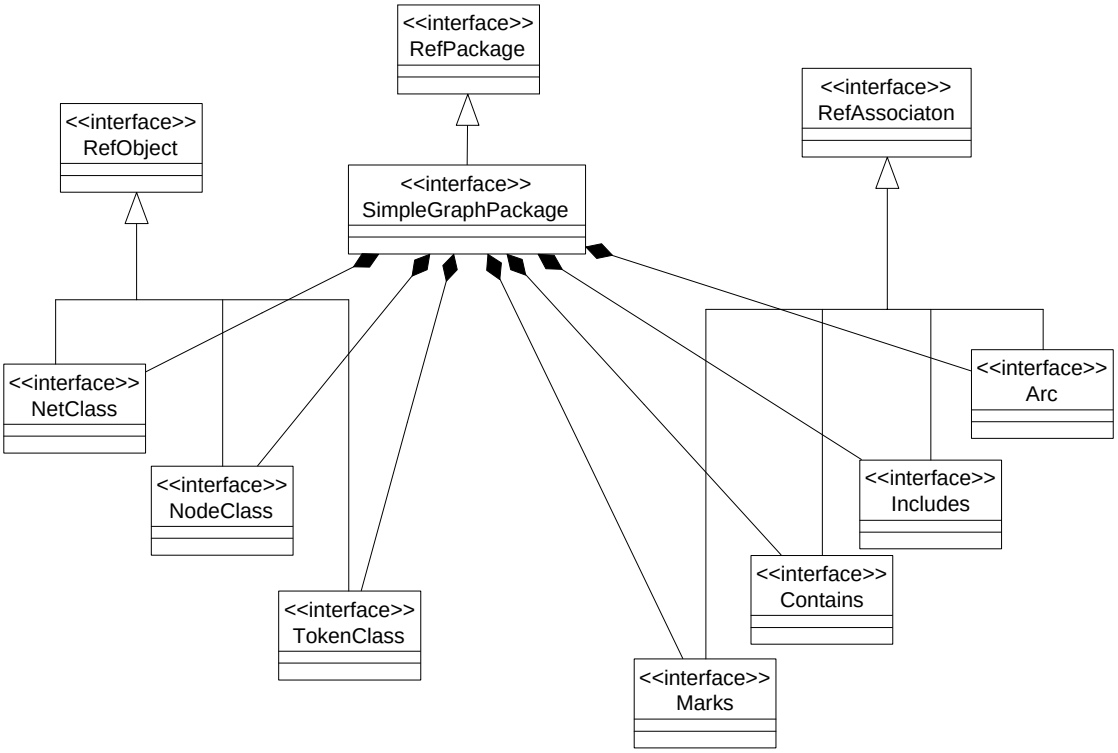

Figure 8-5 Generated interfaces from the SimpleGraph metamodel

– not on the component end of a composition link. So, within one Net instance, XML is produced in the same manner as described before:

```
<Net XMI.id=’a1’>
  <created>
    <field>1868128</field>
    <field>GMT</field>
  </created>
  <nodes>
    <Node XMI.id='a2'>
      <name>NodeX<name>
      <targetNodes>
        <XMI.reference target='a3' />
      </targetNodes>
    </Node>
    <Node XMI.id='a3'>
      <name>NodeW<name>
      <targetNodes>
        <XMI.reference target='a6' />
      </targetNodes>
    </Node>
  </nodes>
</Net>
```

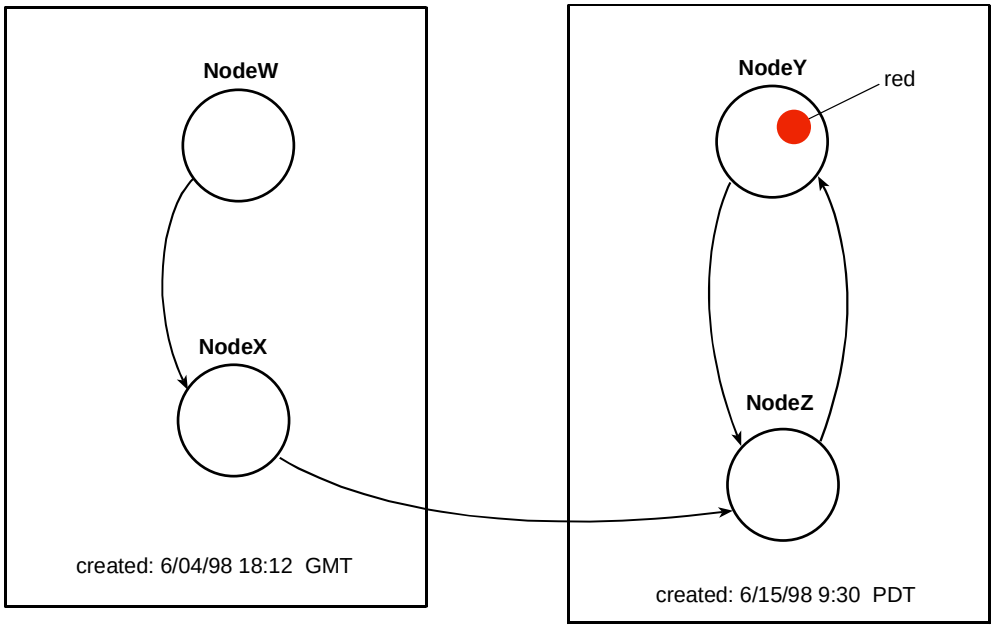

Figure 8-6 Example of two nets with a connecting arc

Similarly, the second Node in the NodeClass extent is used to produce the following XML:

```
<Net XMI.id=’a4’>
  <created>
    <field>1872537</field>
    <field>GMT</field>
  </created>
  <nodes>
    <Node XMI.id='a5'>
      <name>NodeY<name>
      <targetNodes>
        <XMI.reference target='a6' />
      </targetNodes>
    </Node>
    <Node XMI.id='a6'>
      <name>NodeW<name>
      <targetNodes>
        <XMI.reference target='a5' />
      </targetNodes>
      <markers>
        <XMI.reference target='a7' />
```

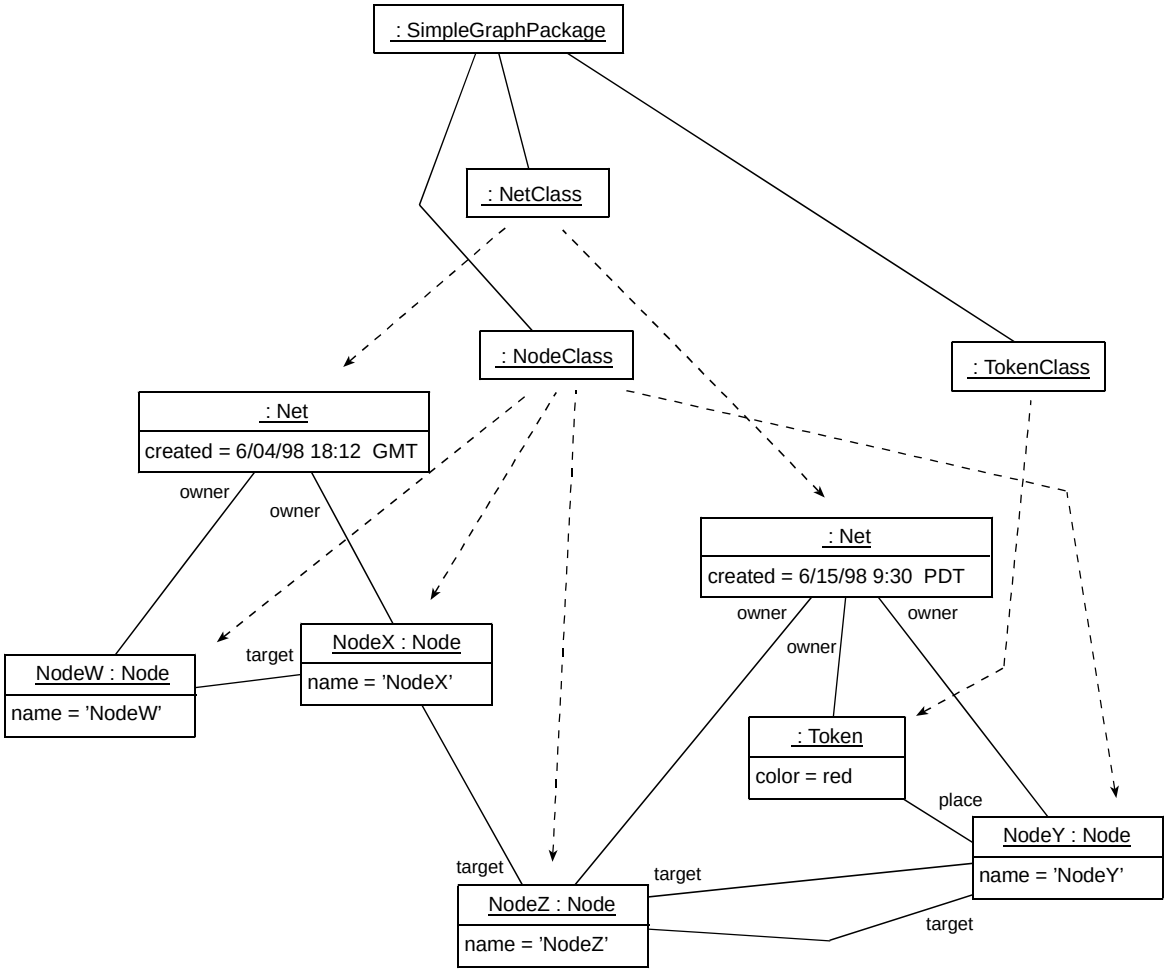

Figure 8-7 Objects representing multiple Nets and instances of RefPackage and RefObject subtypes

```
      </markers>
    </Node>
  </nodes>
  <tokens>
    <Token XMI.id='a7'>
      <color XMI.value='red' />
    </Token>
  </tokens>
</Net>
```

The Production by Package Extent is not unlike writing out an entire workspace, environment, or database. This approach is more desirable when:

- more than one containment hierarchy needs to be exchanged;
- there are interconnections among separate containment hierarchies that need to be replicated; or

- it is necessary to insure that values of classifier-level attributes are preserved, even when no instances of that class exist.

Conversely, creating XML using Production by Object Containment provides:

- finer granularity of the units of interchange; and
- rules definition less dependent upon the RefPackage, RefAssociation, and RefObject features.

8.4 Distinctions between Approaches in Certain Situations

The examples above used very simple models. Some more complex models create situations in which the use each of the two approaches has different consequences.

8.4.1 External Links

Each of the Reference links in the examples referred to an XML element within the XML document. But references can also refer to objects without a representative XML element in the document. Consider the two nets in the second example above. If Production by Object Containment is used to produce XML representing the Net which contains NodeW and NodeX, then the reference of NodeX to NodeZ must be an external link. Since NodeZ is not part of the Net which is used to produce the XML, it will not be represented in the generated document. Instead a URN will be used, which can be resolved to navigate to a representation of the NodeZ object.

This distinction means that, for that example, result of Production by Package Extent would be different than applying Production by Object Containment to the two Net instances. In the latter approach, two XML documents are produced.

8.4.2 Links not Represented by References

On the example metamodel, each Association had a a corresponding Reference defined for the class at one end. However, it is possible, and sometimes desirable or necessary, to define associations without a reference associated with either Association End. For instance, suppose in the SimpleGraph metamodel that the targetNodes Reference was not defined in the Nodes class. Under both approaches, the XML Node elements will not contain any references to the target Nodes. Instead, the links corresponding to the Arc association would be represented as Arc elements. These elements would be contained by the standard XMI.content element.

For Production by Package Extent, after the XML is produced from each of the uncontained objects (and their contents), each of the RefAssociation instances are examined for links in their extent which are not represented in the document. These links would be defined by Associations no Reference is defined for either end.

For Production by Object Containment, the RefAssociation instances are also examined. However, the only links written out are those links not already represented by references (or composition) in which the objects at both ends are in the containment hierarchy.

8.4.3 Classifier-level Attributes

The MOF supports the definition of classes with classifier-level attributes. At the time of model development, within a MOF, these attributes are part of and managed by the RefObject instances contained by the RefPackage (the class proxies). For Production by object containment, the values of a classifier-level attribute will only be made part of the XML document when the containment hierarchy includes at least one instance of that class. Conversely, in Production by Package Extent, any classifier-level attribute not included with an object is represented in XML with an XML element corresponding to the RefObject instance (the class proxy). This again highlights the distinction between the approaches. In programming languages classifier-level attributes, in the form of class variables or static members, are most often considered part of the programming environment. For instance, serialization techniques usually do not serialize these attributes.

XML Document Production 9

9.1 Purpose

This section specifies the production of an XML document from a model. It is essential for model interchange that this specification be complete and unambiguous.

9.2 Introduction

XML document production is defined as a set of rules, which when applied to a model, produce an XML document. These production rules are provided as a specification of the XML production from models and modeling elements. These rules are not a prescription for any specific implementation.

9.3 Rules Representation

The XML produced by XMI is represented here in Extended Backus Nour Form (EBNF). Although this grammar provides a definition of conforming XMI documents, it does not specify how a model is transformed into a document. The Object Constraint Language (OCL) is employed to provide that specification. OCL is a formal language which can specify side-effect free expressions. OCL was introduced as part of the definition of UML, and was used to specify constraints in support of the definition of UML. It was also used in the specification of the MOF. Although intended for the specification of constraints, it is useful in an object-oriented environment for a broader range of specification.

The OCL expressions make use of both the MOF operation definitions, and some operations defined for the Reflective Interfaces. Because these operations are well defined in the MOF specification, their use does not dimish the rigor of these rules.

Although OCL is side-effect free, it is impractical to represent the complete behavior of XML production from a model without retaining some state information. Therefore, a simple OCL class, Producer, is introduced to support this specification. OCL

provides no means of assigning values to objects or their attributes. For this specification, the following notation and semantics are used:

```
Attribute-Expression ← OCL-Expression;
```

where the Attribute-Expression represents an attribute of the Producer class, the symbol "←" represents assignment, with the value of the OCL-Expression replacing the current value of the Producer attribute. In addition to maintaining state during the XML production, Producer attributes are used to represent additional input into the XMP production. In addition to the model itself, other information is needed. As described in Section 8.3, *Two Model Sources* on page 93, there are two separate methods of producing XML from model. Figure 9-1 shows the input and output for the XML production rules using object containment as the model source..

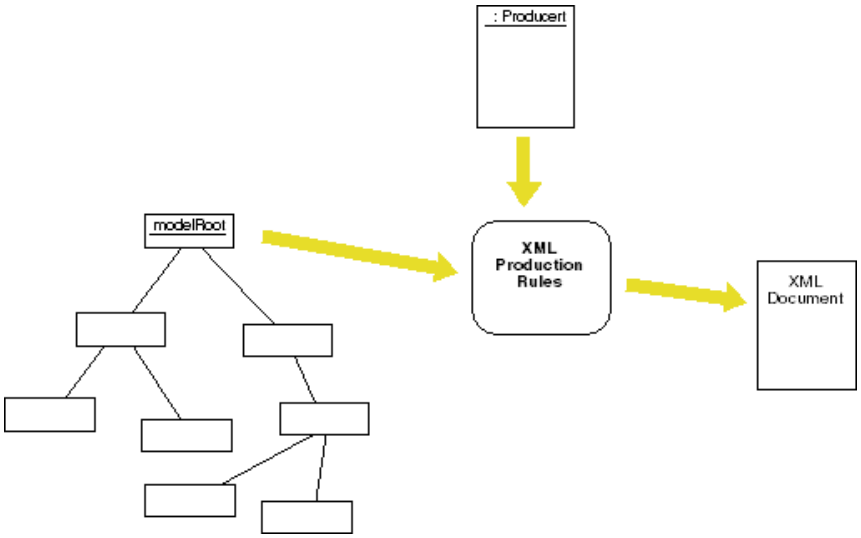

Figure 9-1 XML Production using Object Containment

For each EBNF expression, a corresponding OCL expression is defined. The OCL expression is a query, returning either a string or a sequence of strings. In OCL, a sequence of sequences evaluates to a sequence. For instance,

```
Sequence{ 'aa', Sequence{ 'bb', 'cc' }, 'dd'}
```

evaluates to

```
Sequence{ 'aa', 'bb', 'cc''dd'}.
```

So OCL queries defined as a sequence of other OCL expressions, all returning sequences of strings, will return a simple sequence of strings. For this specification, there is no distinction in the resulting XML between a string and an equivalent sequence of strings.

9.4 Production Rules

The two production schemes, by object containment and by package extent, require two different EBNF expressions, and two different OCL queries. However, both schemes share the bulk of the EBNF expressions and OCL queries. The XMI Document is represented with the EBNF extpression:.

XMIDocment ::= (RootAsDocument | ExtentAsDocument)

showing that an XMI Document is produced either from the root object of a containment heirarchy, or from the extent of a RefPackage subtype.

9.4.1 Production by Object Containment

The following EBNF expressions and OCL queries are specific to the object by containment production scheme.

9.4.1.1 RootAsDocument

The RootAsDocument production produces an XML document representing a root object and its entire contents. The containment hierarchy defines a tree, so the initial object is considered the root. The document is a sequence defined by the DocumentPrologue (containing the information about the metamodel, versions, etc., and the closing tag), the ObjectAsElement (the element and its contents produced from the root object), the OtherLinks (links that were not previously represented in the document), and the DocumentEnd (including the matching enclosing tag).

RootAsDocument ::= DocumentPrologue ContentsFromRoot ContentsAsExtensions? DocumentEnd

The corresponding OCL query describes the production of the document, as a sequence of other queries.

RootAsDocument(root : RefObject) : Sequence(string)

```
RootAsDocument(root) =
  Sequence{ DocumentPrologue(),
            ContentsFromRoot(root),
            ContentsAsExtensions(root),
            DocumentEnd()
          }
```

9.4.1.2 ContentsFromRoot

The ContentsFromRoot production generates the XMI.contents element and the element contents from the root object of the model. The document is a sequence defined by the DocumentPrologue – containing the information about the metamodel, versions, etc., and the enclosing tag, the ObjectAsElement – the element and its

contents produced from the root object, the OtherLinks – links that were not previously represented in the document, and the DocumentEnd – including the matching end tag.

ContentsFromRoot ::= ContentsElementStartTag ObjectAsElement OtherLinks? ContentsElementEndTag

In the OCL operation, the OtherLinks() operation is always evaluated. However, as shown in that operation, an empty sequence may be returned.

ContentsFromRoot(root : RefObject) : Sequence(string)

```
ContentsFromRoot(root) =
  Sequence{ ContentsElementStartTag(),
            ObjectAsElement(root),
            (if linksRemain() then
               OtherLinks(root)
             else
               Sequence{ }
             endif),
            ContentsElementEndTag()
          }
```

9.4.2 Production by Package Extent

These expressions define the production of an XML document using a Package Extent (See Section 8.3.3, *Production by Package Extent* on page 100).

9.4.2.1 Extent as Document

The top level EBNF expression produces the XML document as the document prologue, the contents of the XMI.contents element obtained from the package extent, contents represented as values corresponding to metamodel extensions, and the document end tags.t

ExtentAsDocument ::= DocumentProlog ContentsFromExtent ContentsAsExtensions? DocumentEnd

The corresponding OCL query describes the production of the document, as a sequence of other queries.

RootAsDocument(pkgProxy : RefPackage) : Sequence(string)

```
ExtentAsDocument(pkgProxy) =
  Sequence{ DocumentPrologue(),
            ContentsFromExtent(pkgProxy),
            ContentsAsExtensions(pkg),
            DocumentEnd()
          }
```

9.4.2.2 ContentsFromExtent

The contents of the XML document are enclosed in an XMI.content element.

ContentsFromExtent ::= ContentsElementStartTag ObjectAsElement* ClassAttributes* OtherExtentLinks? ContentsElementEndTag

This OCL operation accepts a RefPackage, a package proxy corresponding to an uncontained Package instance from the MOF Model The operation produces the XMI.content element including the start tag, the XML produced starting with each uncontained object in the extent, the previously unrepresented classifier-level attributes, the previously unrepresented links, and the XMI.content end tag.

ExtentAsDocument(pkgProxy : RefPackage) : Sequence(string)

```
ExtentAsDocument(pkgProxy) =
  Sequence{ ContentsElementStartTag(),
            AllUncontainedObjects(pkgProxy)->collect(obj |
               ObjectAsElement(obj)),
            AllClassProxies(pkgProxy)->collect(c | ClassAttributes(c)),
            OtherExtentLinks(pkgProxy),
            ContentsElementEndTag(),
          }
```

9.4.3 Object Productions

The rest of the expressions in this document are not specific to either the object containment or package extent productions. The object productions define expressions producing XML from objects.

9.4.3.1 ObjectAsElement

An object is represented as an element by producing an element start tag, then the object as the contents of the element, followed by the element end tag..

ObjectAsElement ::= ObjectStartTag ObjectContents ElementEndTag

In the OCL qurey, the metaObject operation is the operation from the Reflective interface, RefObject. In these rules, this operation always returns an instance of the MOF Class type. So, even though the signature of the metaObject operation defines a

return type more abstract (RefObject), the explicit cast is not shown ith the OCL operations.

ObjectAsElement(obj : RefObject) : Sequence(string)

```
ObjectAsElement(obj) =
  Sequence{ ObjectStartTag(obj.metaObject(), obj),
            ObjectContents(obj.metaObject(), obj),
            ElementEndTag(obj.metaObject())
          }
```

9.4.3.2 ObjectContents

The Object-Contents operation produces XML to represent the contents of an object – its state (attributes and references). This is separated from ObjectAsElement expression above to allow representation of objects which are not individually accessible, such as attribute values, which require no element identifier.

To produce XML from the input object, requires three steps; produce the XML for the object's attribute values, then produce the XML for the object's non-composite, non-component references, and finally produce the XML for the objects' component objects. The composite references, the references which may refer to the object's container (at most one could be present) are omitted. Component references are omitted, since the component is directly contained within the composite element in the XML document. Only the non-composite, non-component references need to be represented via XLink elements..

ObjectContents ::= **AttributeAsElement* ReferenceAsElement* CompositeAsElement***

The value operation is provided in the MOF's reflective operations. For a single-valued attribute or reference (multiplicity.upper equals one), a single value is returned; for a multivalued attribute or reference, a sequence of values is returned. The findElementsByTypeExtented operation, defined in the MOF, return all the contained elements of a specified type. When invoked on a Class, it returns inherited features as well.

ObjectContents(metaClass : MofClass, obj : RefObject) : Sequence(string)

```
ObjectContents(metaClass, obj) =
  Sequence{ metaClass.findElementsOfTypeExtended(MofAttribute, false)->
              collect(attr | AttributeAsElement(attr, obj.value(attr))),

            metaClass.findElementsOfTypeExtended(Reference, false)->
               select(ref | ref.exposedEnd.aggregation <> composite and
                  ref.referencedEnd.aggregation <> composite)->
                     collect(r | ReferenceAsElement(r, obj.value(r))),

            metaClass.findElementsOfTypeExtended(Reference, false)->
               select(ref | ref.exposedEnd.aggregation = composite)->
                 collect(r | CompositeAsElement(r, obj.value(r)))
          }
```

9.4.3.3 EmbeddedObject

An alternative is provided to the ObjectAsElement production, for producing elements of objects that will not be referenced.

EmbeddedObject ::= ElementStartTag ObjectContents ElementEndTag

The extract_Object operation is an operation defined for the CORBA Any type.

EmbeddedObject(value : Any) : Sequence(string)

```
EmbeddedObject(value) =
  Sequence{ ElementStartTag
              (value.extract_Object().oclAsType(RefObject).metaObject()),
            ObjectContents
              (value.extract_Object().oclAsType(RefObject).metaObject(),
               value.extract_Object().oclAsType(RefObject)),
            ElementEndTag
              (value.extract_Object().oclAsType(RefObject).metaObject())
          }
```

9.4.3.4 AnyValue

The AnyValue expression produces the XMI fragment representing any value which cannot participate in a Link (typically an attribute value or part of a complex data type value). The value is either an object value or a data value..

AnyValue ::= (EmbeddedObject | DataValue)

The AnyValue operation accepts a value of type Any, and a type indicator in the form of a TcKind value, returning the XMI representation of that value. The value may be an object, a simple data value, or a value of a complex type, such as a sequence or structure

AnyValue(value : Any, kind : TcKind) : Sequence(string)

```
AnyValue(value, kind) =
  if kind = tk_objref then
    EmbeddedObject(value)
  else
    DataValue(attr, kind)
  endif
```

9.4.4 AttributeProduction

Each object attribute value is represented in XML in enclosing start and end tags which identify the attribute. If the attribute value is a datatype, the value can be represented directly, without enclosing tags defining the value's type. For object

values, the object type is represented by the enclosing element tag representing the object's class. The Any type must be treated special. An attribute of type Any will always have its value enclosed in tags which identify its type.

9.4.4.1 AttributeAsElement

An AttributeAsElement is either a classifier-level attribute or an instance-level attribute. Care is taken to insure that classifier-level attributes are only represented once.

AttributeAsElement ::= AttibuteValue?

In the OCL operation, the classifier-level or instance-level property is determined by the scope attribute of the MOFAttribute value.

AttributeAsElement(attr : MofAttribute, value : Any) : Sequence(string)

```
AttributeAsElement(attr, value) =
  if attr.scope = instance_level then
    AttributeValue(attr, value)
  else
    if not self.classiferAttributesCovered->includes(attr) then
      AttributeValue(attr, value)
    else
      Sequence{ }
    endif
  endif
```

In addition to the OCL operation, the following modification is made to the Producer object. Doing so supports identifying any links not accessible through a reference.

```
self.classiferAttributesCovered ← self.classiferAttributesCovered->append(attr)
```

9.4.4.2 AttributeValue

An AttributeValue is either AttributeContents or MvAttributeContents – a potential collection of contents – enclosed in an element start and end tag.

AttributeValue ::= ElementStartTag
(AttributeContents | MvAttributeContents)
ElementEndTag

In the OCL operation, the selection between treating the contents as a single attribute value or a collection of attribute values is determined by the multiplicity in the attribute definition. The cardinality of the actual attribute value is not considered.

AttributeValue(attr : MofAttribute, value : Any) : Sequence(string)

```
AttributeValue(attr, value) =
  if value.oclIsOfType(Sequence) and not value->isEmpty then
    Sequence{ ElementStartTag(attr),
              (if attr.multiplicity.upper < 2 and
                   attr.multiplicity.upper <> unbound then
                 AttributeContents(attr, value)
               else
                 MvAttributeContents(attr, value)
               endif),
              ElementEndTag(attr)
            }
  else
    Sequence{ }
  endif
```

9.4.5 *AttributeContents*

The contents of a single valued attribute is either an object or a data value.

AttributeContents ::= (EmbeddedObject | DataValue)

In the OCL operation, different queries are used for object values and data values.

AttributeContents(attr : MofAttribute, value : Any) : Sequence(string)

```
AttributeContents(attr, value) =
  if attr.type().oclIsOfType(Class) then
    EmbeddedObject(value)
  else
    DataValue(value, Dealias(attr.type().typeCode()).kind())
  endif
```

9.4.5.1 *MvAttributeContents*

For a multivalued attribute's contents, multiple attribute values may be present. Because data values do not have enclosing element tags, each value is delimited with a sequence item tag.

MvAttributeContents ::= (EmbeddedObject* |
(SeqItemStartTag DataValue SeqItemEndTag)*)

In the OCL operation, the ExtractSequence operation is a convenience qurey for transforming the value of the Any type into a sequence.

MvAttributeContents(attr : MofAttribute, value : Any) : Sequence(string)

```
MvAttributeContents(attr, value) =
  if attr.type().oclIsOfType(Class) then
    ExtractSequence(value)->collect( obj | EmbeddedObject(obj))
  else
    ExtractSequence(value)->collect(item |
           Sequence{ SeqItemStartTag(),
                     DataValue(item, Dealias(attr.type().typeCode()).kind()),
                     SeqItemEndTag()
                   } )
  endif
```

9.4.6 Reference Productions

In the MOF, the object-to-object navigability via links is supported through the definition of References in the Classes. XMI makes use of this feature in representing objects. Two kinds of references are treated special. The first is the reference to a component (or part) object. This condition is signified when the reference definition's exposed end has an aggregation of value composite. The second special reference is to the component (container) of an object. This condition is signified when the reference definition's referenced end has an aggregation of value composite.

9.4.6.1 ReferenceAsElement

A reference is represented as an element with a start end tag enclosing one or more reference contents.

ReferenceAsElement ::= ElementStartTag ReferenceContents+ ElementEndTag

ReferenceAsElement(ref : Reference, value : Any)

```
ReferenceAsElement(ref, value) =
  Sequence{ ElementStartTag(ref),
            (if attr.multiplicity.upper < 2 and
                  attr.multiplicity.upper <> unbound then
               ReferenceContents(value.extract_Object().oclAsType(RefObject))
             else
               ExtractSequence(value)->collect (item |
                  ReferenceContents(item.extract_Object().oclAsType(RefObject)))
             endif),
            ElementEndTag(ref)
          }
```

In addition to the OCL operation, the following modification is made to the Producer object. Doing so supports identifying any links not accessible through a reference.

```
self.coveredLinkDefinitions ← self.coveredLinkDefinitions->append(ref)
self.coveredLink ← self.coveredLinks->append(value)
```

9.4.6.2 ReferenceContents

ReferenceContents :: '<reference' ((' id="' IdOfObject(obj) '"') | (' href="' ExternalIdOfObject(obj) '"'))

ReferenceContents(obj : RefObject) : Sequence(string)

```
ReferenceContents(obj) =
  Sequence{ '<reference',
            (if IdOfObject(obj) <> '' then
                Sequence{ ' idref="', IdOfObject(obj), '"' }
             else
                Sequence{ ' href="', ExternalIdOfObject(obj), '"' }
             endif),
            '/>'
          }
```

9.4.7 Composition Production

9.4.7.1 CompositionAsElement

CompositeAsElement ::= ElementStartTag ObjectAsElement+ ElementEndTag

CompositeAsElement(ref : Reference, value : Any)

```
CompositeAsElement(ref, value) =
  Sequence{ ElementStartTag(ref),
             (if attr.multiplicity.upper < 2 and
                   attr.multiplicity.upper <> unbound then
                ObjectAsElement(value.extract_Object().oclAsType(RefObject))
              else
                ExtractSequence(value)->collect (item |
                   ObjectAsElement(item.extract_Object().oclAsType(RefObject)) )
              endif),
            ElementEndTag(ref)
          }
```

In addition to the OCL operation, the following modification is made to the Production object, to support identifying any links not accessible through a reference.

```
self.coveredLinkDefinitions ← self.coveredLinkDefinitions->append(ref)
self.coveredLink ← self.coveredLinks->append(value)
```

9.4.8 DataValue Productions

9.4.8.1 DataValue

DataValue ::= **(StructValue|SequenceValue|ArrayValue|UnionValue|StringValue|CharacterValue|OctetValue|IntegerValue|RealValue | TypeCodeValue | CorbaAnyValue)**

DataValue(value : Any, kind : TCKind) : Sequence(string)

```
DataValue(value, kind) =
  if kind = tk_struct then
    StructValue(value)
  else
    if kind = tk_sequence then
      SequenceValue(value)
    else
      if kind = tk_array then
        ArrayValue(value)
      else
        if kind = tk_union then
          UnionValue(value)
        else
          if kind = tk_string or kind = tk_wstring then
            StringValue(value)
          else
            if kind = tk_char or kind = tk_wchar then
              CharacterValue(value)
            else
              if kind = tk_enum then
                EnumValue(value)
              else
                if kind = tk_boolean then
                  BooleanValue(value)
                else
                  if kind = tk_octet then
                    OctetValue(value)
                  else
                    if Set{ tk_short, tk_ushort, tk_long, tk_ulong,
                            tk_longlong, tk_ulonglong }->includes(kind) then
                      IntegerValue(value)
                    else
                      if kind = tk_float or kind = tk_double or
                         kind = tk_longdouble or kind = tk_fixed then
                        RealValue(value)
                      else
                        if kind = tk_TypeCode then
                          TypeCodeValue(value)
                        else
                          if kind = tk_any then
                            CorbaAnyValue(value)
                          else
                              Sequence {}  -- should never be the case
                          endif
                        endif
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  endif
```

9.4.8.2 StructValue

StructValue ::= **(FieldStartTag AttributeContents FieldEndTag)***

StructValue(value : Any) : Sequence(string)

```
StructValue(value) =
  Sequence{ 0..value.typeCode().member_count() - 1 }->collect(index |
     Sequence{ FieldStartTag(),
               AnyValue(FieldValue(value, index),
                        Dealias(value.typeCode().member_type(index))),
               FieldEndTag()
             })
```

9.4.8.3 SequenceValue

SequenceValue ::= **(SeqItemStartTag AttributeContents SeqItemEndTag }***

SequenceValue(value : Any) : Sequence(string)

```
SequenceValue(value) =
  ExtractSequence(value)->collect( seqItem |
     Sequence{ SeqItemStartTag(),
               AttributeContents
                 (seqItem,
                  Dealias(Dealias(value.typecode()).content_type()).kind()),
               SeqItemEndTag()
             } )
```

9.4.8.4 ArrayValue

ArrayValue ::= **(SeqItemStartTag AttributeContents SeqItemEndTag }***

ArrayValue(value : Any) : Sequence(string)

```
ArrayValue(value) =
  ExtractSequence(value)->collect( seqItem |
     Sequence{ SeqItemStartTag(),
               AttributeContents
                 (seqItem,
                  Dealias(Dealias(value.typecode()).content_type()).kind()),
               SeqItemEndTag()
             } )
```

9.4.8.5 UnionValue

UnionValue ::= **TBD**

UnionValue(value : Any) : Sequence(string)

```
UnionValue(value) = TBD
```

9.4.8.6 StringValue

StringValue ::= **EncodedString**

StringValue(value : Any) : Sequence(string)

```
-- update for wstring
StringValue(value) = EncodedString(value.extract_string())
```

9.4.8.7 CharacterValue

CharacterValue ::= **EncodedCharacter**

CharacterValue(value : Any) : string

```
-- update for wchar
CharacterValue(value) = EncodedCharacter(value.extract_char())
```

9.4.8.8 OctetValue

OctetValue ::= **OctetAsString**

OctetValue(value : Any) : string

```
OctetValue(value) = OctetAsString(value.extract_octet())
```

9.4.8.9 IntegerValue

IntegerValue ::= **IntegerAsString**

IntegerValue(value : Any) : string

```
IntegerValue(value) =
  IntegerAsString(
     if value.typeCode().kind() = tk_short then
       value.extract_short()
     else
       if value.typeCode().kind() =  tk_ushort then
         value.extract_ushort()
       else
         if value.typeCode().kind() =  tk_long then
           value.extract_long()
         else
           if value.typeCode().kind() =  tk_ulong then
             value.extract_ulong()
           else
             if value.typeCode().kind() =   tk_longlong then
               value.extract_longlong()
             else
               if value.typeCode().kind() =  tk_ulonglong then
                 value.extract_ulonglong()
               else
                  -- undefined
               endif
             endif
           endif
         endif
       endif
     endif
   )
```

9.4.8.10 RealValue

RealValue ::= **RealAsString**

RealValue(value : Any) : string

```
RealValue(value) =
  RealAsString(
     if value.typeCode().kind() = tk_float then
       value.extract_float()
     else
       if value.typeCode().kind() =  tk_double then
         value.extract_double()
       else
          -- undefined
       endif
     endif
   )
```

9.4.9 CORBA-Specific Types

9.4.9.1 TypeCodeValue

TypeCodeValue ::= **TypeCodeState**

Producer object Modifications:

```
self.constructedTcList ← Sequence{ }
```

TypeCodeValue(value : Any) : Sequence(string)

```
TypeCodeValue(value) =
  TypeCodeState(value.extract_TypeCode())
```

9.4.9.2 CorbaAnyValue

CorbaAnyValue ::= **(CorbaAnyObject | CorbaAnyData)**

CorbaAnyValue(obj : Any) : Sequence(string)

```
CorbaAnyObject(obj) =
  if DeAlias(value.typeCode()).kind() = tk_objref then
    CorbaAnyObject(value)
  else
    CorbaAnyData(value)
  endif
```

9.4.9.3 CorbaAnyObject

CorbaAnyObject ::= CorbaAnyStartTag ObjectValue CorbaAnyEndTag

CorbaAnyObject(obj : Any) : Sequence(string)

```
CorbaAnyObject(obj) =
  Sequence{ CorbaAnyStartTag(DeAlias(obj.typeCode()).kind(),
                             obj.typeCode().id()),
            EmbeddedObject(obj),
            AnyEndTag()
          }
```

9.4.9.4 CorbaAnyData

CorbaAnyData ::= CorbaAnyStartTag DataValue CorbaAnyEndTag

CorbaAnyData(value : Any) : Sequence(string)

```
CorbaAnyData(value) =
  Sequence{ (if Set{ tk_objref, tk_struct, tk_union, tk_enum, tk_alias,
                     tk_except }->includes(obj.typeCode().kind()) then
               CorbaAnyStartTag(DeAlias(obj.typeCode()).kind(),
                                obj.typeCode().id(),
                                obj.typeCode().name())
             else
               CorbaAnyStartTag(DeAlias(obj.typeCode()).kind(), '', '')
             endif),
            DataValue(value, DeAlias(obj.typeCode()).kind()),
            CorbaAnyEndTag()
          }
```

9.4.9.5 CorbaAnyStartTag

CorbaAnyStartTag ::= '<XMIdata.CorbaAny' CorbaType CorbaId? CorbaName? '>'

CorbaAnyStartTag(kind : TcKind, id : string, name : string) : Sequence(string)

```
CorbaAnyData(kind, id, name) =
  Sequence{ '<XMIdata.CorbaAny',
            CorbaType(kind),
           (if id <> '' then
              CorbaId(id)
            else
              Sequence{ }
            endif),
           (if name <> '' then
              CorbaName(name)
            else
              Sequence{ }
            endif),
           '>'
          }
```

9.4.9.6 CorbaAnyEndTag

CorbaAnyEndTag ::= '</XMIdata.CorbaAny>'

CorbaAnyEndTag() : string

```
CorbaAnyEndTag() =  '</XMIdata.CorbaAny>'
```

9.4.9.7 TypeCodeState

We have to define the mechanism for representing typecodes, since they are Corba datatypes

TypeCodeState ::= **TcStartTag (TcAlias | TcStruct | TcSequence | TcObjRef | TcEnum | TcUnion | TcExcept | TcString | TcSimple) TcEndTag**

TypeCodeState(tc : TypeCode) : Sequence(string)

```
TypeCodeState(tc) =
  Sequence{ TcStartTag(),
            (if tc.kind() = tk_alias then
               TcAlias(tc)
             else
               if tc.kind() = tk_struct then
                 TcStruct(tc)
               else
                 if tc.kind() = tk_sequence then
                   TcSequence(tc)
                 else
                   if tc.kind() = tk_objref then
                     TcObjRef(tc)
                   else
                     if tc.kind() = tk_enum then
                       TcEnum(tc)
                     else
                       if tc.kind() = tk_union then
                         TcUnion(tc)
                       else
                         if tc.kind() = tk_except then
                           TcExcept(tc)
                         else
                           if tc.kind() = tk_string then
                             TcString(tc)
                           else
                             TcSimple(tc)
                           endif
                         endif
                       endif
                     endif
                   endif
                 endif
               endif
             endif),
            TcEndTag()
          }
```

9.4.9.8 TcAlias

TcAlias ::= **TcAliasStartTag TypeCodeState TcAliasEndTag**

TcAlias(tc : TypeCode) : Sequence(string)

```
TcAlias(tc) =
  Sequence{ TcAliasStartTag(tc.name(), tc.id()),
            TypeCodeState(tc.alias()),
            TcAliasEndTag()
          }
```

9.4.9.9 TcStruct

TcStruct ::= **TcStructStartTag (TcFieldStartTag TypeCodeState TcFieldEndTag)* TcStructEndTag**

Producer Object Modifications

```
self.constructedTcList ← self.constructedTcList->append(tc)
```

TcStruct(tc : TypeCode) : Sequence(string)

```
TcStruct(tc) =
  Sequence{ TcStructStartTag(tc.name(), tc.id()),
            Sequence{ 0..(tc.member_count - 1) }->collect(i |
               TcFieldStartTag(tc.member_name(i)),
               TypeCodeState(tc.member_type(i)),
               TcFieldEndTag() )
```

9.4.9.10 TcSequence

TcSequence ::= **TcSequenceStartTag(TypeCodeState|TcRecursiveLink) TcSequenceEndTag**

TcSequence(tc : TypeCode) : Sequence(string)

```
TcSequence(tc) =
  Sequence{ TcSequenceStartTag(tc.length(), tc.id())
            if self.constrcutedTcList->includes(tc.content_type())
              TcRecursiveLink(tc)
            else
              TypeCodeState(tc.content_type())
            endif
            TcSequenceEndTag()
          }
```

9.4.9.11 TcObjRef

TcObjRef ::= TcObjRefTag

TcObjRef(tc : TypeCode) : Sequence(string)

```
TcObjRef(tc) =
  Sequence{ TcObjRefTag(tc.name(), tc.id()) }
```

9.4.9.12 TcEnum

TcEnum ::= **TcEnumStartTag TcEnumLabelTag* TcEnumEndTag**

TcEnum(tc : TypeCode) : Sequence(string)

```
TcEnum(tc) =
  Sequence{ TcEnumStartTag(tc.name(), tc.id()),
            Sequence{ 0..(tc.member_count-1) }->collect(i |
               TcEnumLabelTag(tc.member_name(i)) ),
            TcEnumEndTag()
          }
```

9.4.9.13 TcUnion

TcUnion ::= **TcUnionStartTag TcUnionDiscrimStartTag TypeCodeState TcUnionDiscrimEndTag (TcFieldStartTag TypeCodeState AnyValue TcFieldEndTag)* TcUnionEndTag**

TcUnion(tc : TypeCode) : Sequence(string)

```
TcUnion(tc) =
  Sequence{ TcUnionStartTag(tc.name(), tc.id()),
             TcUnionDiscrimStartTag(tc.default_index()),
             TypeCodeState(tc.discriminator_type(),
             TcUnionDiscrimEndTag,
             Sequence{ 0..(tc.member_count-1) }->collect(i |
                TcFieldStartTag(tc.member_name()),
                TypeCodeState(tc.member_type(i)),
                AnyValue(tc.member_label(i)),
                TcFieldEndTag() },
             TcUnionEndTag()
          }
```

9.4.9.14 TcExcept

TcExcept ::= **TcExceptStartTag (TcFieldStartTag TypeCodeState TcFieldEndTag)* TcFieldEndTag**

TcExcept(tc : TypeCode) : Sequence(string)

```
TcExcept(tc) =
  Sequence{ TcExceptStartTag(tc.name(), tc.id()),
            Sequence{ 0..(tc.member_count-1) }->collect(i |
               TcFieldStartTag(tc.member_name(i)),
               TypeCodeState(tc.member_type(i)),
               TcFieldEndTag() ),
            TcExceptEndTag()
          }
```

9.4.9.15 TcString

TcString ::= **TcStringTag**

TcString(tc : TypeCode) : Sequence(string)

```
TcString(tc) =
  TcStringTag(tc.length())
```

9.4.9.16 TcSimple

TcSimple ::= **(TcShortTag | TcLongTag | TcUshortTag | TcUlongTag | TcFloatTag | TcDoubleTag | TcBooleanTag | TcCharTag | TcOctetTag | TcAnyTag | TcTypeCodeTag | TcPrincipalTag | TcNullTag | TcVoidTag)**

TcSimple(tc : TypeCode) : Sequence(string)

```
TcSimple(tc) =
  if tc.kind() = tk_short then
    TcShortTag()
  else
    if tc.kind() = tk_long then
      TcLongTag()
    else
      if tc.kind() = tk_ushort then
        TcUshortTag()
      else
        if tc.kind() = tk_ulong then
          TcUlongTag()
        else
          if tc.kind() = tk_float then
            TcFloatTag()
          else
            if tc.kind() = tk_double then
              TcDoubleTag()
            else
              if tc.kind() = tk_boolean then
                TcBooleanTag()
              else
                if tc.kind() = tk_char then
                  TcCharTag()
                else
                  if tc.kind() = tk_octet then
                    TcOctetTag()
                  else
                    if tc.kind() = tk_any then
                      TcAnyTag()
                    else
                      if tc.kind() = tk_TypeCode then
                        TcTypeCodeTag()
                      else
                        if tc.kind() = tk_Principal, then
                          TcPrincipalTag()
                        else
                          if tc.kind() = tk_null then
                            TcNullTag()
                          else
                            if tc.kind() = tk_void then
                              TcVoidTag()
                            else
                              -- undefined (mnot expected)
                            endif
                          endif
                        endif
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  endif
```

9.4.9.17 CorbaType

CorbaType ::= **' tcType=' "'"** *tkName* **"'"**

CorbaType(kind : TcKind) : Sequence(string)

```
CorbaType(kind) =
  Sequence{ ' tcType=', '\'',
            -- return a string corresponding to the kind value
            (if kind = tk_objref then
               'tk_objef'
             else
               if kind = tk_short then
                'tk_short'
               else
                 -- the other cases omitted
               endif
             endif),
            '\''
          }
```

9.4.9.18 CorbaId

CorbaId ::= ' tcId=' "'" *repositoryId* **"'"**

CorbaId(id : string) : Sequence(string)

```
CorbaId(id) =
  Sequence{ ' tcId=', '\'', id, '\''}
```

9.4.9.19 CorbaName

This expression provides a name originating from a CORBA TypeCode. The

CorbaName ::= ' tcName=' "'" *typeName* **"'"**

CorbaName(name : string) : Sequence(string)

```
CorbaaName(name) =
  Sequence{ ' tcName=', '\'', name, '\''}
```

9.4.9.20 TcStartTag

TcStartTag ::= **'<XMIdata.CorbaTypeCode>'**

TcStartTag() : string

```
TcStartTag() =  '<XMIdata.CorbaTypeCode>'
```

9.4.9.21 TcEndTag

TcEndTag ::= **'</XMIdata.CorbaTypeCode>'**

TcEndTag() : string

```
TcEndTag() =  '</XMIdata.CorbaTypeCode>'
```

9.4.9.22 TcAliasStartTag

TcAliasStartTag ::= **'<XMIdata.CorbaTcAlias' CorbaName CorbaId? '>'**

TcAliasStartTag(name : string, id : string) : Sequence(string)

```
TcAliasStartTag() =
  Sequence{ '<XMIdata.CorbaTcAlias',
            CorbaName(name),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.23 TcAliasEndTag

TcAliasEndTag ::= **'</XMIdata.CorbaTcAlias>'**

TcAliasEndTag() : string

```
TcAliasEndTag() =  '</XMIdata.CorbaTcAlias>'
```

9.4.9.24 TcStructStartTag

TcStructStartTag ::= '<XMIdata.CorbaTcStruct' CorbaName CorbaId? '>'

TcStructStartTag(name : string, id : string) : Sequence(string)

```
TcStructStartTag(name, id) =
  Sequence{ '<XMIdata.CorbaTcStruct',
            CorbaName(name),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.25 TcStructEndTag

TcStructEndTag ::= '</XMIdata.CorbaTcStruct>'

TcStructEndTag() : string

```
TcStructEndTag() =  '</XMIdata.CorbaTcStruct>'
```

9.4.9.26 TcUnionStartTag

TcUnionStartTag ::= '<XMIdata.CorbaTcUnion' CorbaName CorbaId? '>'

TcUnionStartTag(name : string, id : string) : Sequence(string)

```
TcUnionStartTag(name, id) =
  Sequence{ '<XMIdata.CorbaTcUnion',
            CorbaName(name),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.27 TcUnionEndTag

TcUnionEndTag ::= '</XMIdata.CorbaTcUnion>'

TcUnionEndTag() : string

```
TcUnionEndTag() =  '</XMIdata.CorbaTcUnion>'
```

9.4.9.28 TcSequenceStartTag

TcSequenceStartTag ::= '<XMIdata.CorbaTcSequence' CorbaLength? CorbaId? '>'

TcsequenceStartTag(length : long, id : string) : Sequence(string)

```
TcSequenceStartTag(kength, id) =
  Sequence{ '<XMIdata.CorbaTcSequence',
            (if length <> 0 then
               CorbaLength(length)
             else
               Sequence{ }
             endif),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.29 TcSequenceEndTag

TcSequenceEndTag ::= '</XMIdata.CorbaTcSequence>'

```
TcSequenceEndTag() : string
TcSequenceEndTag() =  '</XMIdata.CorbaTcSequence>'
```

9.4.9.30 TcObjRefStartTag

TcObjRefStartTag ::= '<XMIdata.CorbaTcObjRef' CorbaName CorbaId? '>'

```
TcObjRefStartTag(name : string, id : string) : Sequence(string)
TcObjRefStartTag(name, id) =
  Sequence{ '<XMIdata.CorbaTcObjRef',
            CorbaName(name),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.31 TcObjRefEndTag

TcObjRefEndTag ::= '</XMIdata.CorbaTcObjRef>'

```
TcObjRefEndTag() : string
TcObjRefEndTag() =  '</XMIdata.CorbaTcObjRef>'
```

9.4.9.32 TcFieldStartTag

Only the default field of a CORBA Union TypeCode nnot have its name represented.

TcFieldStartTag ::= '<XMIdata.CorbaTcField' CorbaName? '>'

TcFieldStartTag(name : string) : Sequence(string)

```
TcFieldStartTag(name) =
  Sequence{ '<XMIdata.CorbaTcField',
            (if name <> '' then
               CorbaName(name)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.33 TcFieldEndTag

TcFieldEndTag ::= '</XMIdata.CorbaTcField>'

TcFieldEndTag() : string

```
TcFieldEndTag() =  '</XMIdata.CorbaTcField>'
```

9.4.9.34 TcEnumStartTag

TcEnumStartTag ::= '<XMIdata.CorbaTcEnum' CorbaName CorbaId? '>'

TcEnumStartTag(name : string, id : string) : Sequence(string)

```
TcEnumStartTag(name, id) =
  Sequence{ '<XMIdata.CorbaTcEnum',
            CorbaName(name),
            (if id <> '' then
               CorbaId(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.35 TcEnumEndTag

TcEnumEndTag ::= '</XMIdata.CorbaTcEnum>'

TcEnumEndTag() : string

```
TcEnumEndTag() =  '</XMIdata.CorbaTcEnum>'
```

9.4.9.36 TcUnionDiscrimStartTag

TcUnionDiscrimStartTag ::='<XMIdata.CorbaTcUnionDiscrim' CorbaDefaultIndex? '>'

TcUnionDiscrimStartTag(defaultIndex : long) : Sequence(string)

```
TcUnionDiscrimTag(defaultIndex) =
  Sequence{ '<XMIdata.CorbaTcUnionDiscrim',
            (if defaultIndex <> -1 then
               CorbaDefaultIndex(id)
             else
               Sequence{ }
             endif),
            '>'
          }
```

9.4.9.37 TcUnionDiscrimEndTag

TcUnionDiscrimEndTag ::='</XMIdata.CorbaTcUnionDiscrim>'

TcUnionDiscrimEndTag() : string

```
TcUnionDiscrimEndTag() =  '</XMIdata.CorbaTcUnionDiscrim>'
```

9.4.9.38 TcEnumLabelTag

TcEnumLabelTag ::= '<XMIdata.CorbaTcEnumLabel' CorbaName '/>'

TcEnumLabelTag(name : string) : Sequence(string)

```
TcEnumLabelTag(name) =
  Sequence{ '<XMIdata.CorbaTcEnumLabel',
            CorbaName(name),
            '/>'
          }
```

9.4.9.39 TcStringTag

TcStringTag ::= **'<XMIdata.CorbaTcString' CorbaLength? '/>'**

TcStringTag(length : long) : Sequence(string)

```
TcStringTag(length) =
  Sequence{ '<XMIdata.CorbaTcString',
            CorbaLength(length),
            '/>'
          }
```

9.4.9.40 TcShortTag

TcShortTag ::= **'<XMIdata.CorbaTcShort/>'**

TcShortTag() : string

```
TcShortTag() = '<XMIdata.CorbaTcShort/>'
```

9.4.9.41 TcLongTag

TcLongTag ::= **'<XMIdata.CorbaTcLong/>'**

TcLongTag() : string

```
TcLongTag() = '<XMIdata.CorbaTcLong/>'
```

9.4.9.42 TcUlongTag

TcUlongTag ::= **'<XMIdata.CorbaTcUlong/>'**

TcUlongTag() : string

```
TcUlongTag() = '<XMIdata.CorbaTcUlong/>'
```

9.4.9.43 TcUshortTag

TcUshortTag ::= **'<XMIdata.CorbaTcUshort/>'**

TcUshortTag() : string

```
TcUshortTag() = '<XMIdata.CorbaTcUshort/>'
```

9.4.9.44 TcFloatTag

TcFloatTag ::= **'<XMIdata.CorbaTcFloat/>'**

TcFloatTag() : string

```
TcFloatTag() = '<XMIdata.CorbaTcFloat/>'
```

9.4.9.45 TcDoubleTag

TcDoubleTag ::= **'<XMIdata.CorbaTcDouble/>'**

TcDoubleTag() : string

```
TcDoubleTag() = '<XMIdata.CorbaTcDouble/>'
```

9.4.9.46 TcBooleanTag

TcBooleanTag ::= **'<XMIdata.CorbaTcBoolean/>'**

TcBooleanTag() : string

```
TcBooleanTag() = '<XMIdata.CorbaTcBoolean/>'
```

9.4.9.47 TcCharTag

TcCharTag ::= **'<XMIdata.CorbaTcChar/>'**

TcCharTag() : string

```
TcCharTag() = '<XMIdata.CorbaTcChar/>'
```

9.4.9.48 TcOctetTag

TcOctetTag ::= **'<XMIdata.CorbaTcOctet/>'**

TcOctetTag() : string

```
TcEnumStartTag() = '<XMIdata.CorbaTcOctet/>'
```

9.4.9.49 TcAnyTag

TcAnyTag ::= **'<XMIdata.CorbaTcAny/>'**

TcAnyTag() : string

```
TcAnyTag() = '<XMIdata.CorbaTcAny/>'
```

9.4.9.50 TcTypeCodeTag

TcTypeCodeTag ::= **'<XMIdata.CorbaTcTypeCode/>'**

TcTypeCodeTag() : string

```
TcTypeCodeTag() = '<XMIdata.CorbaTcTypeCode/>'
```

9.4.9.51 TcPrincipleTag

TcPrincipleTag ::= **'<XMIdata.CorbaTcPrinciple/>'**

TcPrincipleTag() : string

```
TcPrincipleTag() = '<XMIdata.CorbaTcPrinciple/>'
```

9.4.9.52 TcNullTag

TcNullTag ::= **'<XMIdata.CorbaTcNull/>'**

TcNullTag() : string

```
TcNullTag() = '<XMIdata.CorbaTcNull/>'
```

9.4.9.53 TcVoidTag

TcVoidTag ::= **'<XMIdata.CorbaTcVoid/>'**

TcVoidTag() : string

```
TcVoidTag() = '<XMIdata.CorbaTcVoid/>'
```

9.4.9.54 TcLongLongTag

TcLongLongTag ::= **'<XMIdata.CorbaTcLongLong/>'**

TcLongLongTag() : string

```
TcLongLongTag() = '<XMIdata.CorbaTcLongLong/>'
```

9.4.9.55 TcUlongLongTag

TcUlongLongTag ::= **'<XMIdata.CorbaTcUlongLong/>'**

TcUlongLongTag() : string

```
TcUlongLongTag() = '<XMIdata.CorbaTcUlongLong/>'
```

9.4.9.56 TcLongDoubleTag

TcLongDoubleTag ::= **'<XMIdata.CorbaTcLongDouble/>'**

TcLongDoubleTag() : string

```
TcLongDoubleTag() = '<XMIdata.CorbaTcLongDouble/>'
```

9.4.9.57 TcFixedTag

TcFixedTag ::= **'<XMIdata.CorbaTcFixed/>'**

```
TcFixedTag() : string
TcFixedTag() = '<XMIdata.CorbaTcFixed/>'
```

9.4.10 Document Prologue

9.4.10.1 Document Prologue

Each XMI document produced by XMI has the same root element

DocumentProlog ::= **XMLDecl? doctypedecl XmiElementStartTag XmiHeader XmiContentStartTag**

The XML Recommendation provides the definitions of **XMLDecl** and **doctypedecl**.

The two operations, XMLDecl() and doctypedecl(), are not individually specified. XMLDecl must produce a string or sequence of strings which conforms to the definition of the XMLDecl expression in the XML Recommendation. Likewise, doctypedecl() must produce a string or sequence of strings which conforms to the definition of doctypedecl in the XML Recommendation. The doctypedecl expression represents the Document Type Declaration, which specifies the Document Type Definition (DTD), by reference, by embedding the markup in the declaration, or by a combination of the two.

```
DocumentProlog() : Sequence(string)
DocumentProlog() =
  Sequence{ XMLDecl(),
            (if self.dtdUsage <> none then
               doctypedecl()
             else
               Sequence{ }
             endif),
            XmiElementStartTag(),
            XmiHeader()
          }
```

9.4.10.2 XmiElementStartTag

Each XMI document produced by XMI has the same root element, <XMI>.

XmiElementStartTag ::= '<XMI' XmiVersion? XmiTimestamp? xmiVerified? '>'

In the Production Class, the showVerified attribute indicates whether the version attribute will be explicitly included in the document. Absence of the DTD requires the specification of the version. The Production's showTimestamp attribute determines the production of a timestamp. The showVerified attribute determines whether the verification status should be included.

XmiElementStartTag() : Sequence(string)

```
XmiElementStartTag() =
  Sequence{ '<XMI',
            (if self.explicitVersion = true or self.dtdUsage = none then
               XmiVersion()
             else
               Sequence{ }
             endif),
            (if self.showTimestamp = true then
               XmiTimestamp()
             else
               Sequence{ }
             endif),
            (if self.showVerify = true then
               XmiVerified()
             else
               Sequence{ }
             endif)
          }
```

9.4.10.3 XmiVersion

When a DTD is present, either embedded or by reference, the version element attribute may be omitted. The DTD, as specificity in the DTD Production Rules, will define the version attribute of the XMI Element Type as fixed, with the value of '1.0'. Future versions of XMI are expected to be represented with different version numbers, although a specific numbering scheme is not currently specified.

XmiVersion ::= ' xmi-version=' "'" '1.0' "'"

XmiVersion() : Sequence(string)

```
XmiVersion() =
  Sequence{ ' xmi-version=', '\'', '1.0', '\'' }
```

9.4.10.4 XmiTmestamp

The XMI Document can optionally provide a timestamp. The timestamp expression is not defined here. It conforms to [ISO8601]. We recommend the timestamp, when generated, should conform to the profile found in the [DATETIME] W3C Note used in the specification of HTML 4.0 [HTML40].,

XmiTimestamp ::= **' xmi-version=' "'" *timestamp* "'"**

The OCL operation, timestamp, is not separately specified. It returns a string or sequence of strings conforming to the above description.

XmiTimestamp() : Sequence(string)

```
XmiVersion() =
  Sequence{ ' timestamp=', '\'', timestamp() '\'' }
```

9.4.10.5 XmiVerified

The XMI Document can optionally provide a indicate whether the source model has been verified. The specific definition of a verified model will vary, based on the meta-model. If a metamodel does not provide a definition of model verification, then the corresponding models should not use this element attribute in the document production.,

XmiVerified ::= **' verified=' "'" ('true' | 'false') "'"**

XmiVerified() : Sequence(string)

```
XmiVersion() =
  Sequence{ ' verified=',
            '\'',
            (if self.modelVerified = true then
               'true'
             else
               'false'
             endif),
            '\'' }
```

9.4.10.6 XmiHeader

XmiHeader ::= **XmiDocumentation? XmiMetaModel+**

XmiVerified() : Sequence(string)

```
XmiVersion() =
  Sequence{ ' verified=',
            '\'',
            (if self.modelVerified = true then
               'true'
             else
               'false'
             endif),
            '\'' }
```

9.4.11 Terminals

9.4.11.1 EnumValue

EnumValue ::= **ElementEmptyStartTag ' value="'** *enumValue* **'"/>'**

ElementValue(attr : MofAttribute, value : Any) : Sequence(string)

```
EnumAsElement(attr, value) =
  Sequence{ ElementEmptyStartTag(attr.type()),
            ' value="',
            value.typeCode().element_name(ExtractEnumValue(value)),
            '"/>'
          }
```

9.4.11.2 BooleanAsElement

BooleanAsElement ::= ElementEmptyStartTag, ' value="' ('true' | 'false') '">'

BooleanValue(value : Any) : Sequence(string)

```
BooleanValue(value) =
  Sequence{ ElementEmptyStartTag(attr.type()),
            ' value="',
            (if value.extract_boolean() then
               'true'
             else
               'false'
             endif),
            '"/>'
          }
```

9.4.11.3 ObjectStartTag

ObjectStartTag ::= **'<' metaClassName, ' id="', IdOfObject(obj), '">'**

```
ObjectStartTag(metaObject : ModelElement, obj : RefObject) :
   Sequence(string)
```

```
ObjectStartTag(metaObject, obj) =
  Sequence{ '<', MetaObjectName(metaObject, obj),
            ' id="', IdOfObject(obj), '">'
          }
```

9.4.11.4 metaObjectName

metaObjectName ::= *identifier* **| (** *identifier* **('.'** *identifier***)+)**

```
metaObjectName(metaObject : ModelElement, obj : RefObject) : string
```

```
metaObjectName(metaObject, obj) =
  if self.shortenedNames then
    if AllMetaObejcts()->forAll(obj | obj.name() <> metaObject.name()) then
      metaObject.name()
    else
      DotNotation(metaObject.qualifiedName())
  else
      DotNotation(metaObject.qualifiedName())
  endif
```

9.4.11.5 ElementEndTag

ElementEndTag ::= **'>'**

```
ElementEndTag() : string
```

```
ElementEndTag() = '>'
```

9.4.11.6 SeqItemStartTag

SeqItemStartTag ::= **'<XMIdata.seqItem>'**

SeqItemStartTag() : string

SeqItemStartTag() = ’<XMIdata.seqItem>’

9.4.11.7 SeqItemEndTag

SeqItemEndTag ::= ’</XMIdata.seqItem>’

SeqItemEndTag() : string

SeqItemEndTag() = ’</XMIdata.seqItem>’

9.4.11.8 FieldStartTag

FieldStartTag ::= ’<XMIdata.field>’

FieldStartTag() : string

FieldStartTag() = ’<XMIdata.field>’

9.4.11.9 FieldEndTag

FieldEndTag ::= ’</XMIdata.field>’

FieldEndTag() : string

FieldEndTag() = ’</XMIdata.field>’

9.4.11.10 FieldStartTag

FieldStartTag ::= ’<XMIdata.field>’

FieldStartTag() : string

FieldStartTag() = ’<XMIdata.field>’

9.4.11.11 FieldEndTag

FieldEndTag ::= **'</XMIdata.field>'**

FieldEndTag() : string

```
FieldEndTag() = '</XMIdata.field>'
```

9.4.12 Helpers

9.4.12.1 ExtractEnumValue

ExtractEnumValue(value : Any) : long

```
ExtractEnumValue(value) =
  value.create_input_stream().read_long()
```

9.4.12.2 DeAlias

DeAlias(tc : TypeCode) : TypeCode

```
DeAlias(tc) =
  if tc.kind() = tkalias then
    DeAlias(tc.content_type())
  else
    tc
  endif
```

9.4.12.3 IdOfObject

IdOfObject(obj : RefObject) : string

```
  if Sequence{ 1..(self.objectInventory->size) }->select(i |
       self.objectInventory->at(i) = obj)->isEmpty then
    if InScope(obj) then
      NewObjectId(obj)
    else
      ''
  else
    self.objectIds.at(Sequence{ 1..(self.objectInventory->size) }->select(i |
       self.objectInventory->at(i) = obj)->first)
  endif
```

9.4.12.4 DotNotation

DotNotation(names : Sequence(string)) : string

```
DotNotation(names) =
  substring(names->iterate(s : string, answer : string = '' |
              string.contat(s).concat('.')),
            names->iterate(s : string, answer : string = '' |
              string.contat(s).concat('.')).size)
```

Compatibility with Other Standards 10

10.1 Introduction

The XMI specification addresses the metadata interchange requirement of the OMG repository architecture which is described in the OMG MOF specification (ad/97-10-02, Section 1.3) and corresponds to the 'Data Interchange' component of the architecture. The XMI specification conforms to the following standards:

- XML, the Extensible Markup Language, is a new data format for electronic interchange designed to bring structured information to the web. XML is an open technology standard of the W3C (www.w3c.org), the standards group responsible for maintaining and advancing HTML. XML is used as the concrete syntax and transfer format for OMG MOF compliant metadata.

  There are several benefits of basing metamodel interchange on XML. XML is an open standard, platform and vendor independent. XML supports the international character set standards of extended ISO Unicode. XML is metamodel-neutral and can represent metamodels compliant with OMG's meta-metamodel, the MOF. XML is programming language-neutral and API-neutral. XML APIs are provided in additional standards, giving the user an open choice of several access methods to create, view, and integrate XML information. Leading XML APIs include DOM, SAX, and WEB-DAV.

- MOF, the Meta Object Facility is an OMG (www.omg.org) metadata interface standard that can be used to define and manipulate a set of interoperable metamodels and their instances (models). The MOF also defines a simple meta-metamodel (based on the OMG UML - Unified Modeling Language) with sufficient semantics to describe metamodels in various domains starting with the domain of object analysis and design. The XMI specification uses MOF as the meta-metamodel to ensure transfer of any MOF compliant metamodel (such as UML) and instances of these metamodels - the models themselves.

- UML, the Unified Modeling Language is an OMG (www.omg.org) standard modeling language for specification, construction, visualization and documentation of the artifacts of a software system. The XMI can be used to exchange UML models between tools and between tools and repositories.
- The CORBA interfaces specified in the MOF (ad/97-10-02, ad/97-10-03) can be used to internalize and externalize XML streams of MOF based metamodels. (See the interface MOF::Package in ad/97-10-02) for more details. In this sense, the XMI together with the MOF conforms to the OMA and can be used as the foundation for developing web based distributed development environments.

In summary the XMI supports W3C XML, OMG MOF, UML and OMA standards. There are no dependencies on any other standards.

It is anticipated that additional work is required to provide a migration path from existing metadata interchange standards (such as EIA CDIF - Electronics Industry Associates Case Data Interchange Format) to XMI should such a market requirement exist. The submitters believe that such a migration path is possible based on

1. Implementation experience on CDIF
2. The MOF meta-metamodel has all the modeling concepts needed to represent the CDIF meta-metamodel and provide appropriate transformation algorithms from CDIF to MOF and vice-versa.

Such a migration path could result from potential collaboration between XMI and CDIF experts.

Conformance Issues 11

11.1 Introduction

This section describes the required and optional points of compliance with the XMI specification. The term “XML recommendation” refers to technical recommendations by the W3C for XML version 1.0 and later [XML reference] [W3C reference].

11.2 Required Compliance

11.2.1 XMI DTD Compliance

XMI DTDs are required to conform to the following points:

- The XMI DTD(s), both internal and external, must be “valid” and “well-formed” as defined by the XML recommendation [XMI reference]
- The determination of compliance on a DTD is made in the “expanded form” where all entity information is expanded out. Many variations of entity declarations result in the same “expanded form” DTD, each variation having have identical compliance.
- The expanded form of an XMI DTD must follow the processing and fixed element declarations of Section 6.3.2, *Requirements of XMI DTDs*, Section 6.5, *XMI DTD and Document Structure*, and Section 6.6, *Common XMI DTD Declarations*.
- An expanded form XMI DTD must have the “same” set of elements as those which are created in expanded form using one of the rule sets from Chapter 6. The definition of “same” for two DTDs is that there is an exact one to one correspondence between the elements in each DTD, each correspondence identical in terms of element name, element attributes (name, type, and default actions), element content specification, content grammar, and content multiplicities.

11.2.2 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 [XMI reference], whether used with or without the document's corresponding XMI DTD(s). Although it is optional not to transmit and/or validate a document with its XMI DTD(s), the document must still conform as if the check had been made.
- The XMI document must contain the XML declarations and processing instructions as defined in Section 6.5, *XMI DTD and Document Structure*.
- The XMI document must contain one or more XMI root elements that together contain all other XMI information within the document as defined in Section 6.6, *Common XMI DTD Declarations*.
- The XMI document must be the "same" as a document following the document production rules of Section 9. The definition of "same" for two documents is that there is an exact 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, XMI.extension, and XMI.extensions elements are excepted.

11.2.3 Usage Compliance

The XMI documents must be used under the following conditions:

- The XML parsers, browsers, or other tools used to input and/or output XMI information must conform to the XML recommendation [XMI reference]. Note that early releases of many tools are not fully XML version 1.0 compliant.

11.3 Optional Compliance Points

11.3.1 XMI DTD Compliance

XMI DTDs optionally conform to the following points:

- The definition of XML entities within DTDs are suggested to follow the design rules in Section 6.3, Section 6.4, Section 6.6, Section 6.7, Section 7.3, and Section 7.4.

11.3.2 XMI Document Compliance

XMI Documents optionally conform to the following points:

- The guidelines for using the XMI.extension and XMI.extensions elements are suggested in Section 6.6 and Section 6.8. In general, tools should place their extended information within the designated extension areas, declare the nature of the extension using the standard XMI elements where applicable, and preserve the extensions of other tools where appropriate.

11.3.3 Usage Compliance

The XMI documents are optionally used under the following conditions:

- The XML parsers, browsers, or other tools used to input and/or output XMI information should conform to standard APIs for the XML recommendation [XMI reference]. These APIs include, but are not limited to, DOM [DOM reference], SAX [SAX reference], and Web-DAV [Web-DAV reference].
- Note that the early releases of many tools are not fully XML version 1.0 compliant. Check for updated versions of the tools or use the references as a guide for locating compliant tools.

References

[ISO8601] “Data elements and interchange formats -- Information interchange -- Representation of dates and times”, ISO 8601:1988

[HTML40] "HyperText Markup Language Specification Version 3.0", Dave Raggett, September 1995.

[XML] XML, a technical recommendation standard of the W3C. http://www.w3.org/TR/REC-xml

[NAMESP] Namespaces, a working draft of the W3C. http://www.w3.org/TR/WD-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

[XPointer] XPointer, working draft of the W3C. http://www.w3.org/TR/WD-xptr

[RDF] RDF, a working draft of the W3C. http://w3c.org/RDF/

[RDFSCHEM] RDF-Schema, a working draft of the W3C. http://www.w3.org/TR/WD-rdf-schema

[XMLDATA] XML-Data, a note for discussion purposes to the W3C. http://www.w3.org/TR/1998/NOTE-XML-data

[XSL] XSL, a working draft of the W3C. http://www.w3.org/Style/XSL/

[DOM] DOM, a working draft of the W3C. http://www.w3.org/DOM/

[SAX] SAX, a standard of the XML-DEV mailing list. http://www.microstar.com/XML/SAX/

[WEBDAV] Web-DAV, a working draft of the IETF. http://www.ietf.org/html.charters/webdav-charter.html

[UML] UML, an adopted standard of the OMG. http://www.omg.org

[MOF] MOF, an adopted standard of the OMG. http://www.omg.org

[XMLJAVA] XML for Java, a free, complete, commercial XML parser written in Java by IBM. http://www.alphaworks.ibm.com/formula/xml

The following is the XML specification's reference to its character set standards:

[ISO10646] ISO (International Organization for Standardization). ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

The following is the XML specification's reference to its character set standards:

[Unicode] The Unicode Consortium. The Unicode Standard, Version 2.0. Reading, Mass.: Addison-Wesley Developers Press, 1996.

Glossary

This glossary defines the terms that are used to describe the XMI specification. The glossary includes concepts from the Meta Object Facility (MOF) as well as key concepts of the Unified Modeling Language (UML) for completeness. The rationale for including key MOF and UML terms is to be consistent in the definition and usage of fundamental object modeling as well as meta modeling constructs and to provide a baseline for creating a common glossary for all OMG OA&DTF modeling and metadata related technologies. This glossary builds on the UML 1.1 and MOF 1.1 glossaries.

In addition to MOF and UML specific terminology it includes related terms from OMG standards, W3C standards, object-oriented analysis and design methods as well as the domain of object repositories and meta data managers. Glossary entries are organized alphabetically. The new glossary entries have been marked (XMI) and mainly consist of Extensible markup Language (XML) related terminology. For a more comprehensive description of XML, please refer to www.w3c.org.

Scope

This glossary includes terms from the following sources:

- Meta Object Facility 1.1 specification which has been adopted by the OMG
- Appendix M1 from the UML 1.1 specification which has been adopted by the OMG
- Object Management Architecture object model [OMA]
- CORBA 2.0 [CORBA]
- Object Analysis & Design RFP-1 [OA&D RFP]
- W3C XML 1.0 specification [XML]

Notation Conventions

The entries in the glossary usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.

When brackets enclose one or more words in a multi-word term, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.

The following conventions are used in this glossary:

- *Contrast: <term>*. Refers to a term that has an opposed or substantively different meaning.
- *See: <term>*. Refers to a related term that has a similar, but not synonymous meaning.
- *Synonym: <term>*. Indicates that the term has the same meaning as another term, which is referenced.
- *Acronym: <term>*. This indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.

The glossary is extensively cross-referenced to assist in the location of terms that may be found in multiple places.

Terms

abstract class — A class that cannot be directly instantiated.

abstraction — The essential characteristics of an entity that distinguish it from all other kind of entities. An abstraction defines a boundary relative to the perspective of the viewer.

actual parameter — Synonym: argument

aggregate [class] — A class that represents the "whole" in an aggregation (whole-part) relationship. See: .aggregation

aggregation — A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. composition

analysis — The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses on what to do, design focuses on how to do it.

analysis time — Refers to something that occurs during an analysis phase of the software development process.

architecture — The organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts.

argument	A specific value corresponding to a parameter. Synonym: actual parameter.
artifact	A piece of information that is used or produced by a software development process. An artifact can be a model, a description or software.
association	A relationship that describes a set of .
association class	A modeling element that has both association and class properties. An association class can be seen as an association that also has class , or as a class that also has association properties.
association role	The role that a type or class plays in an .
attribute	A named property of a type. Synonym: attribute [OMA].
behavior	The observable effects of an operation or event, including its results. Synonym:
binary association	An between two classes. A special case of an .
boolean	An whose values are true and false.
boolean expression	An that evaluates to a boolean value.
CDATA Section (XMI)	A part of an XML document in which markup (apart from that indicating the end of CDATA section) is not interpreted as such, but is passed to the application as is.
cardinality	The number of elements in a set.
class	A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class is an implementation of type.
class diagram	A that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.
client	A type, class, or component that requests a service from another type, class or component.
compile time	Refers to something that occurs during the compilation of a software module.
component	An executable software module with identity and a well-defined interface.
component diagram	A that shows the organizations and dependencies among .
composite [class]	A class that is related to one or more classes by a composition relationship. See: .
composite aggregation	
composite state	A state that consists of substates.
composition	A form of with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive. Synonym: composite aggregation.
concrete class	A class that can be directly instantiated.

constraint A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extendibility mechanisms in UML. See: , Textual Constraint, Sequence Constraint.

container 1. An object that exists to contain other objects, and that provides operations to access or iterate over its contents. For example, arrays, sets, dictionaries. 2. A component that exists to contain other components.

ontainment hierarchy Within the MOF model, the containment hierarchy is the acyclic graph defined by the Namespaces and ModelElements participating in the Namespace-Contains-ModelElement association. For any Namespace instance, its containment hierarchy is the ModelElements and Links defined by the transitive closure of its contents reference.

context A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

defining model (1.1) Each repository is based on a model, which it considers its defining model. Any number of repositories can have the same defining model.

delegation The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance.

dependency A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

derived element A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design The part of the software development process whose primary purpose is to decide how the system will be implemented. During design, strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time Refers to something that occurs during a design phase of the software development process. See: .

development process A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the following diagrams:

document element (XMI) See root element.

ocument Type Definition (XMI) See DTD.

domain An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic classification A of generalization in which an object may change type or role.

DTD (XMI)	A set of rules governing the element types that are allowed within an XML document and rules specifying the allowed content and attributes of each element type. The DTD also declares all the external entities referenced within the document and the notations that can be used.
element	An atomic constituent of a A logical unit of information in a XML document. (XMI)
element attributes (XMI)	The name-value pairs assigned within an element's start-tag.
element content (XMI)	The elements or text that is contained between an element's start-tag and end-tag.
element type (XMI)	A particular type of element, such as a paragraph in a document or a class in a metamodel. The element type is indicated by the name that occurs in its start-tag and end-tag.
mpty string	A string with zero characters.
end tag (XMI)	A tag that marks the end of an element, such as </Model>. Also see start-tag.
enumeration	A list of named values used as the range of a particular attribute type. For example, Color = {Red, Green, Blue}.
event	A significant occurrence. An event has a location in time and space and may have parameters. In the context of , an event is an occurrence that can trigger a state .
export	In the context of packages, to make an element visible outside its enclosing . See: . Contrast: , .
expression	A string that evaluates to a value of a particular type. For example, the expression "(7 + 5 * 3)" evaluates to a value of type number.
extends	A relationship from one to another, specifying how the behavior defined for the first use case can be inserted into the behavior defined for the second use case. See refines.
formal parameter	
framework	A micro-architecture that provides an extensible template for applications within a specific domain.
generalizable element	A model element that may participate in a generalization relationship. See: .
generalization	A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: .
HTML (XMI)	Hyper Text Markup Language. An encoding scheme for displaying and hyperlinking pages of information on the World Wide Web. HTML is an application of SGML.
implementation	A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.
implementation inheritance	The inheritance of the implementation of a more specific element. Includes inheritance of the interface.

import	In the context of , a that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: , .
inheritance	The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See .
instance	An individual member described by a type or a class. Usage note: According to a strict interpretation of the metamodel an individual member of a type is an instance and a member of a class is an object. In less formal usage it is acceptable to refer to a member of a class as an object or an instance. See: .
interaction	A behavioral specification that comprises a set of message exchanges among a set of objects within a particular context to accomplish a specific purpose. An interaction may be illustrated by one or more scenarios.
interface	The use of a type to describe the externally visible behavior of a class, object, or other entity. In the case of a class or object, the interface includes the of the operations. See: .
interface inheritance	The inheritance of the interface of a more specific element. Does not include inheritance of the implementation.
layer	A specific way of grouping in a at the same level of abstraction.
link	A semantic connection among a tuple of objects. An instance of an association. See: .
link role	An instance of an association role. See: .
list	A whose contents are ordered. An ordered collection. The ordering is not proscribed by the list, but the ordering is preserved, typically with operations that allow placing an element in a desired locations within a list. See: Set, Array , Unique list.
markup (XMI)	Information that is intermingled with the text of an XML document to indicate its logical and physical structure.
member	A part of a type or class denoting either an or an .
message	A communication between objects that conveys information with the expectation that activity will ensue. The receipt of a message is normally considered an .
metaclass	A class whose instances are classes. Metaclasses are typically used to construct .
metainterface	A metatype which is restricted in its use of meta constraints. A metainterface can be refined to support interface definitions in object systems such as CORBA or DCOM. See interface.
meta-metamodel	A model that defines the language for expressing a . The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a .
metamodel	A model that defines the language for expressing a . An instance of a .
metaobject	A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

metatype	A type whose instances are types. Metatypes are typically used to construct .
method	The implementation of an operation. The algorithm or procedure that effects the results of an operation.
model	A semantically closed abstraction of a system. See: .
MOF 1.1 model.	A package which fulfills its role as a complete representation of the intended subject, without being nested in or imported by another package. Typically a model is a top-level package. However, a package can both be a model for one purpose, yet be nested in another package in support of another package, in which case it would not be a top-level package. A top-level package may not be a model, but intended to support packages which import it.
model aspect	A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the emphasizes the structural qualities of the metamodel.
odel elaboration (1.1)	The process of generating a repository type from a published model. Includes the generation of interfaces and implementations which allows repositories to be instantiated and populated, based on, and in compliance with, the model elaborated.
model element	An element that is an abstraction drawn from the system being modeled. Analogous to metaobject.
modeling time	Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note: When discussing object systems it is often important to distinguish between modeling-time and run-time concerns.
module	A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See: .
multiple classification	A of generalization in which an object may belong directly to more than one class. See: .
multiple inheritance	A of generalization in which a type may have more than one supertype.
multiplicity	A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers.
ulti-valued (1.1)	A ModelElement with multiplicity defined is called multi-valued when its MultiplicityType::upper attribute is set to a number greater than one. The term multi-valued does not pertain to the number of values held by an attribute, parameter, etc., at any point in time. Contrast single-valued.
n-ary association	An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes.
name	A string used to identify a model element. In some systems names are first class objects to support a richer name service. Within an XML document (Name - note the capitalization), consists of a letter or underscore followed by zero or more name characters. (XMI)

namespace	A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: .
node	A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Run-time objects and components may reside on nodes.
note	A comment attached to an element or a collection of elements. A note has no semantics.
object	An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class.
operation	A service that can be requested from an object to effect behavior. An operation has a , which may restrict the actual parameters that are possible.
package	A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages. A system may be thought of as a single high-level package, with everything else in the system contained in it. Contrast with Model, MetaModel.
parameter	The specification of a variable that can be changed, passed or returned. A parameter may include a name, type and direction. Parameters are used for operations, messages and events. Synonyms: , formal parameter. Contrast: argument.
parameterized class	The descriptor for a class with one or more unbound parameters. Synonym: template.
participates	A relationship that indicates the role that an instance plays in a modeling element. For example, a class participates in an association, an actor participates in a use case. Contrast: .
persistent object	An object that exists after the process or thread that created it.
postcondition	An that must be true at the completion of an operation.
powertype	A metatype whose instances are subtypes of another type. For example, TreeSpecies is a powertype on the Tree type. The subtypes of Tree (e.g., Ash, Birch, Cherry) are therefore all instances of TreeSpecies.
precondition	An that must be true when an operation is invoked.
primitive type	A predefined basic type, such as an integer or a string.
product	The artifacts of development, such as models, code, documentation, work plans.
projection	A mapping from a set to a subset of it.
property	A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See .
ublished model	A model which has been frozen, and becomes available for instantiating repositories and for the support in defining other models.

qualifier	An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.
receive [a message]	The handling of a message passed from a sender object. See: sender, receiver.
receiver [object]	The object handling a message passed from a sender object. Constrast: sender.
reference	A denotation of a model element.
refinement	A relationship that represents the fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class. Example: the refines association in Meta Feature.
relationship	A semantic connection among model elements. Examples of relationships include and .
requirement	A desired feature, property, or behavior of a system.
responsibility	A contract or obligation of a type or class.
reuse	The use of a pre-existing .
role	The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association role) or dynamic (e.g., a collaboration role).
root element (XMI)	The single element that contains all other elements and character data that comprises an XML document. Also referred to as Document element.
run time	The period of time during which a computer program executes.
semantic variation	A particular interpretation choice for a semantic variation point. See: .
semantic variation point	A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics. See: .
send [a message]	The passing of a message from a sender object to a receiver object.
sender [object]	The object passing a message to a receiver object.
sequence constraint	A constrained defined on an ordered set of operations. A typical example of this constraint is the definition of pre-amble and post-amble operations that are executed before and after a given operation. See constraint, textual constraint.
et (1.1)	An unordered collection where each element is unique.
SGML (XMI)	Standard Generalized Markup Language. An International Standard (ISO 8879:1986) that describes a generalized markup scheme for representing the logical structure of documents in a system-independent and platform independent manner.
signal	A named that can be explicitly invoked ("raised"). Signals may have parameters. A signal may be broadcast or directed toward a single object or a set of objects.
signature	The name and parameters of an operation, message, or event. Parameters may include an optional returned parameter.
single inheritance	A of generalization in which a type may have only one supertype.

single-valued (1.1)	A odelElement with ultiplicity defined is called single-valued when its MultiplicityType::upper attribute is set to one. The term single-valued does not pertain to the number of values held by an attribute, parameter, etc., at any point in time, since a single-valued attribute, for instance, with a multiplicity lower bound of zero may have no value. Contrast: Multi-valued.
specification	A declarative description of what something is or does.
Standard Generalized Markup Language (XMI)	See SGML
start tag (XMI)	A tag that marks the beginning of an element, such as <Model>. Also see end-tag.
state	A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
state diagram	A diagram that shows a state machine. See: state machine.
state machine	A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.
static classification	A of generalization in which an object may not change type or may not change role.
stereotype	A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extendibility mechanisms in UML.
string	A sequence of text characters. The details of string representation depends on implementation, and may include character sets that support international characters and graphics.
structural model aspect	A that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes and operations.
subclass	In a generalization relationship the specialization of another class, the superclass. See: .
substate	A state that is part of a composite state. A substate can either be a concurrent or disjoint substate. See: concurrent state, disjoint state.
subsystem	A subordinate system within a larger system. In the UML a subsystem is modeled as a package of .
subtype	In a generalization relationship the specialization of another type, the supertype. See: .
superclass	In a generalization relationship the generalization of another class, the subclass. See: .
supertype	In a generalization relationship the generalization of another type, the subtype. See: .
supplier	A type, class or component that provides services that can be invoked by others.
system	A collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints.

tagged value	The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extendibility mechanisms in UML.
template	
textual constraint	A textual representation of a constraint. While not rigorous, it presents a convenient mechanism to represent most constraints. The constraint string is interpreted or somehow encoded by a tool or service. See Sequence constraint.
time	A value representing an absolute or relative moment in time.
transient object	An object that exists only during the execution of the process or thread that created it.
transition	A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state the transition is said to fire.
transitive closure (1.1)	for a type t and a reference r, the transitive closure on r is given by: if a is in t.r, then a is in the transitive closure; if a is in t.r, and b is in a.r then b is in the transitive closure; and nothing else is in the transitive closure unless it so follows from (1) or (2). (adapted from Hopcroft & Ullman).
type	A description of a set of instances that share the same operations, abstract attributes and relationships, and semantics. A type may define an operation specification (such as a signature) but not an operation implementation (such as a method). Usage note: Type is sometimes used synonymously with interface, but it is not an equivalent term.
type expression	An expression that evaluates to a reference to one or more types.
uninterpreted	A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See: .
Unique list	A list in which the same element may not appear more than once (duplicate elements are not allowed).
use case [class]	A class that defines a set of use case instances.
use case instance	A sequence of actions a system performs that yields an observable result of value to a particular actor. Usually scenarios illustrate prototypical use case instances. An instance of a use case class. See: use case class.
uses	A relationship from a concrete use case to an abstract use case in which the behavior defined for the concrete use case employs the behavior defined for the abstract use case.
value	An element of a type domain. An instance of a data type. Contrast: .
view	A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.
visibility	An whose value (public, protected, private, or implementation) denotes how the model element to which it refers may be seen outside its enclosing name space.

W3C (XMI)	The World Wide Web Consortium. A standards organization (www.w3c.org) that standardized popular Internet standards such as HTML, XML etc.
well-formed XML document (XMI)	An XML document that consists of a single element containing properly nested subelements. All entity references within the document must refer to entities that have been declared in the DTD, or be one of a small set of default entities.
XLink (XMI)	A syntax for identifying links to external documents. See Xpointer.
XML (XMI)	Extensible Markup Language. A profile, or simplified subset of SGML. W3C standard for representing web metadata contained in XML Documents.
XML Declaration (XMI)	A processing instruction at the start of an XML document, which asserts that the document is an XML document.
XML Document (XMI)	An XML document consists of an optional XML declaration, followed by an optional document type declaration, followed by a document element.
XPointer (XMI)	A syntax for identifying the element, range of elements, or text within an XML document that is the target resource of a link. (XML-Link 6)