XML Metadata Interchange (XMI)

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

Joint Submission

Unisys Corporation	***Fujitsu***
International Business Machines Corporation	***Softeam***
Cooperative Research Centre for Distributed Systems Technology (DSTC)	***Recerca Informatica***
Oracle Corporation	***Daimler-Benz***
Platinum Technology, Inc.	

Supported by:

Cayenne Software	*Ardent*
Genesis Development	*Aviatis*
Inline Software	*ICONIX*
Rational Software Corporation	*Integrated Systems*
Select Software	*Verilog*
Sprint Communications Company	*Telefonica I+D*
Sybase, Inc.	*Universitat Politecnica de Catalunya*
Xerox	*NCR*
MCI Systemhouse	*Nihon Unisys*
Boeing	*NTT*

OMG Document ad/98-10-05
October 20, 1998

Copyright 1998, Unisys Corporation
Copyright 1998, IBM Corporation
Copyright 1998, Cooperative Research Centre for Distributed Systems Technology (DSTC)
Copyright 1998, Oracle Corporation
Copyright 1998, Platinum Technology, Inc.
Copyright 1998, Fujitsu
Copyright 1998, Softeam
Copyright 1998, Recerca Informatica
Copyright 1998, Daimler-Benz

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE,THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any meansógraphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systemsó without the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA, OMG, and Object Request Broker are trademarks of Object Management Group.

Table of Contents

1. Preface 1-9

1.1 Cosubmitting Companies and Supporters 1-9

1.2 Introduction 1-10

1.3 Submission contact points 1-12

1.4 Status of this Document 1-16

1.5 Guide to the Submission 1-16

1.6 Conventions 1-17

2. Proof of Concept 2-19

2.1 Copyright Waiver 2-19

2.2 Proof of Concept 2-19

3. Response to RFP Requirements 3-21

3.1 Mandatory Requirements 3-21

3.1.1 Required Meta-metamodel 3-21

3.1.2 Syntax and Encoding 3-21

3.1.3 Referenced Concepts 3-22

3.1.4 UML Support 3-22

3.1.5 International Codesets 3-22

3.2 Optional Requirements 3-23

3.2.1 Compact Data Representation 3-23

3.2.2 Compatibility with other Metamodels and Interchange Formats 3-23

3.3 Issues for discussion 3-24

3.4 Scope of Revision Task Force 3-26

4. Design Rationale 4-27

4.1 Design Overview 4-27

4.2 XMI and the MOF 4-28

4.2.1 An Overview of the MOF 4-28

4.2.2 The relationship between XMI and MOF . . . 4-31

4.2.3 The relationship between XMI, MOF and UML 4-32

4.3 XMI and XML 4-32

4.3.1 The roots of XML 4-32

4.3.2 Benefits of using XML 4-33

4.3.3 XML and the Computer Industry 4-34

4.3.4 How XML works 4-34

4.3.5 XML and the OMG 4-36

4.3.6 New XML Technologies 4-37

4.4 Major Design Goals and Rationale 4-38

4.4.1 Universally Applicable Solution 4-38

4.4.2 Automatic Generation of Transfer Syntax . . 4-39

4.4.3 Conformance with XML paradigms 4-39

4.4.4 Knowledge of Metamodels 4-40

4.4.5 Complete Encoding of Metadata 4-40

4.4.6 Correctness of MOF MetaModels 4-41

4.4.7 Model Fragments 4-41

4.4.8 Ill-Formed Models 4-41

4.4.9 Model Versions 4-42

4.4.10 Model Extensibility 4-42

4.4.11 MOF as an Information Model 4-43

4.4.12 Status of MOF and UML DTDs 4-43

5. Usage Scenarios 5-45

5.1 Purpose 5-45

5.2 Combining tools in a heterogeneous environment 5-45

5.3 Co-operating with common metamodel definitions 5-46

5.4 Working in a distributed and intermittently connected environment 5-47

5.5 Promoting design patterns and reuse 5-47

6. XMI DTD Design Principles 6-49

6.1 Purpose 6-49

6.2 Use of XML DTDs 6-49

6.2.1 XML Validation of XMI documents 6-50

6.2.2 Requirements for XMI DTDs 6-50
6.3 Basic Principles 6-51
6.3.1 Required XML Declarations 6-51
6.3.2 Metamodel Class Representation 6-51
6.3.3 Metamodel Extension Mechanism 6-52
6.4 XMI DTD and Document Structure 6-52
6.5 Necessary XMI DTD Declarations 6-53
6.5.1 Necessary XMI Attributes 6-53
6.5.2 Common XMI Elements 6-55
6.5.3 XMI 6-56
6.5.4 XMI.header 6-57
6.5.5 XMI.content 6-57
6.5.6 XMI.extensions 6-57
6.5.7 XMI.extension 6-57
6.5.8 XMI.documentation 6-58
6.5.9 XMI.model 6-58
6.5.10 XMI.metamodel 6-59
6.5.11 XMI.metametamodel 6-59
6.5.12 XMI.difference 6-60
6.5.13 XMI.delete 6-60
6.5.14 XMI.add 6-60
6.5.15 XMI.replace 6-60
6.5.16 XMI.reference 6-61
6.5.17 XMI Datatype Elements 6-61
6.6 Metamodel Class Specification 6-64
6.6.1 Fully Qualified Names 6-65
6.6.2 Metamodel Multiplicities 6-65
6.6.3 Class specification 6-66
6.6.4 Inheritance Specification 6-66
6.6.5 Attribute Specification 6-67
6.6.6 Association Specification 6-68
6.6.7 Containment Specification 6-69
6.7 Transmitting Incomplete Metadata 6-69
6.7.1 Interchange of model fragments 6-69
6.7.2 XMI encoding 6-70
6.7.3 Example 6-70
6.8 Linking 6-72
6.8.1 Design principles: 6-72
6.8.2 Linking 6-73
6.8.3 Example from UML 6-74

6.8.4 XMI.reference 6-75
6.9 Transmitting Metadata Differences 6-75
6.9.1 Definitions:. 6-76
6.9.2 Differences 6-76
6.9.3 XMI encoding. 6-76
6.9.4 Example 6-77
6.10 Document exchange with multiple tools 6-78
6.10.1 Definitions:. 6-78
6.10.2 Procedures:. 6-79
6.10.3 Example 6-79
6.11 UML DTD 6-80
6.12 General datatype mechanism 6-81

7. XML DTD Production 7-83

7.1 Purpose. 7-83
7.2 Rule Set 1: Simple DTD. 7-85
7.2.1 Rules. 7-85
7.2.2 Auxiliary functions. 7-93
7.3 Rule Set 2: Grouped entities. 7-107
7.3.1 Rules. 7-107
7.3.2 Auxiliary functions. 7-117
7.4 Rule Set 3: Hierarchical Grouped entities 7-125
7.4.1 Rules. 7-125
7.4.2 Auxiliary functions. 7-135
7.5 Fixed DTD elements 7-145

8. XML Generation Principles 8-153

8.1 Purpose. 8-153
8.2 Introduction 8-153
8.3 Two Model Sources 8-153
8.3.1 Production by Object Containment. 8-154
8.3.2 MOF's Role in XML Production 8-160
8.3.3 Production by Package Extent 8-160
8.4 Distinctions between Approaches in Certain Situations. 8-164
8.4.1 External Links 8-164
8.4.2 Links not Represented by References. 8-165
8.4.3 Classifier-level Attributes. 8-165
8.4.4 Standard Elements 8-165

9. XML Document Production 9-167

9.1 Purpose 9-167

9.2 Introduction 9-167

9.3 Rules Representation 9-167

9.3.1 EBNF Productions 9-168

9.3.2 OCL Rules 9-168

9.4 Production Rules 9-170

9.4.1 Production by Object Containment 9-170

9.4.2 Production by Package Extent 9-172

9.4.3 Object Productions 9-174

9.4.4 Attribute Production 9-176

9.4.5 Reference Productions 9-179

9.4.6 Composition Production 9-181

9.4.7 DataValue Productions 9-181

9.4.8 CORBA-Specific Types 9-189

9.4.9 Helpers 9-199

9.4.10 CORBA-Specific Helpers 9-206

10. Compatibility with Other Standards 10-209

10.1 Introduction 10-209

10.2 XMI and W3C DCD 10-210

10.3 XMI and CDIF 10-210

11. Conformance Issues 11-213

11.1 Introduction 11-213

11.2 Required Compliance 11-213

11.2.1 XMI DTD Compliance 11-213

11.2.2 XMI Document Compliance 11-214

11.2.3 Usage Compliance 11-214

11.3 Optional Compliance Points 11-214

11.3.1 XMI MOF Subset 11-214

11.3.2 XMI DTD Compliance 11-215

11.3.3 XMI Document Compliance 11-215

11.3.4 Usage Compliance 11-215

References References-217

Glossary Glossary-219

Index Index-237

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 response to the Object Analysis & Design Task Force RFP3 - Stream based Model Interchange Format (SMIF):

- Unisys Corporation
- International Business Machines Corporation
- Cooperative Research Centre for Distributed Systems Technology (DSTC)
- Oracle Corporation
- Platinum Technologies, Inc.
- Fujitsu
- Softeam
- Recerca Informatica
- Daimler-Benz

The following companies are pleased to support the XMI specification:

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

- MCI Systemhouse
- Boeing
- Ardent
- Aviatis
- ICONIX
- Integrated Systems
- Verilog
- Telefonica I+D
- Universitat Politecnica de Catalunya
- NCR
- Nihon Unisys
- NTT

1.2 Introduction

The main purpose of XMI is to enable easy interchange of metadata between modeling tools(based on the OMG UML) and metadata repositories (OMG MOF based) in distributed heterogeneous 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, an OMG metamodeling 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 to share object models and other metadata over the Internet.

XMI, together with MOF and UML form the core of the OMG metadata repository architecture as illustrated in Figure 1-1. The UML standard defines a rich, object oriented modeling language that is supported by a range of graphical design tools. The MOF standard defines an extensible framework for defining models for metadata, and providing tools with programmatic interfaces to store and access metadata in a repository. XMI allows metadata to be interchanged as streams or files with a standard format based on XML. The complete architecture offers a wide range of implementation choices to developers of tools, repositories and object frameworks. XMI in particular lowers the barrier to entry for the use of OMG metadata standards.

Key aspects of the architecture include:

- A four layered metamodeling architecture for general purpose manipulation of metadata in distributed object repositories. See the MOF and UML specifications for more details

SMIF (XMI) and OMG Repository Architecture

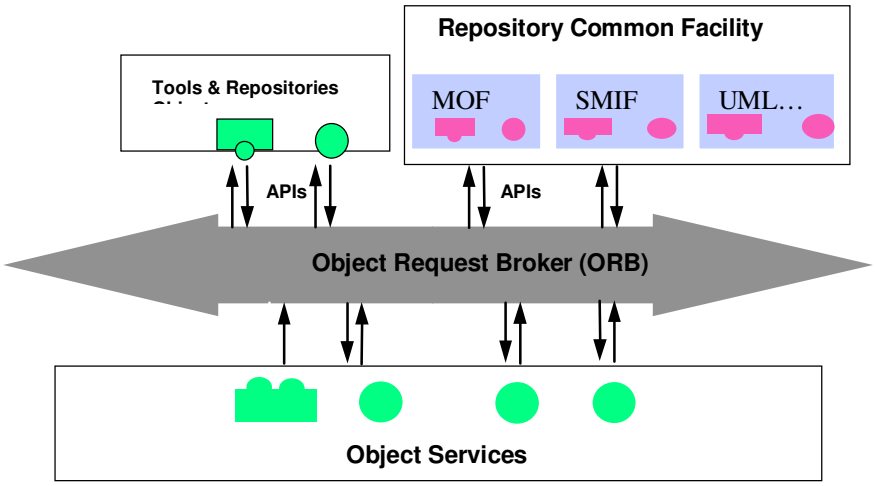

Figure 1-1 The OMG Repository Architecture and the SMIF

- The use of MOF to define and manipulate metamodels programmatically using fine grained CORBA interfaces. This approach leverages the strength of CORBA distributed object infrastructure.
- The use of UML notation for representing models and metamodels
- The use of standard information models (UML) to describe the semantics of object analysis and design models
- The use of SMIF (the current XMI proposal) for stream based interchange of metadata

The OMG ADTF and other task forces have already begun extending this architecture to include data warehouse metadata (Common Warehouse Metadata Interchange RFP) and other metadata by defining MOF compliant metamodels.

This submission mainly consists of:

- A set of XML Document Type Definition (DTD) production rules for transforming MOF based metamodels into XML DTDs
- A set of XML Document production rules for encoding and decoding MOF based metadata
- Design principles for XMI based DTDs and XML Streams
- 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 metadata management and metadata interoperability in distributed object environments in general and in distributed development environments in particular. While this response addresses stream based metadata interoperability in the object analysis and design domain, XMI (in part because it is MOF based) is equally applicable to metadata in many other domains. Examples include metamodels that cover the application development life cycle as well as additional domains such as data warehouse management, distributed objects 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 reduce the plethora of proprietary metadata interchange formats and minimally successful attempts of the Meta Data Coalition (Meta Data Interchange Specification) and Case Data Interchange Format (EIA CDIF) because of widespread adoption of W3C (XML) and OMG (UML, MOF) standards. XMI is also expected to ease the integration of CORBA, XML, Java, and COM based development environments which are evolving towards 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
E-mail: sridhar.iyengar2@unisys.com

Stephen A. Brodsky, Ph.D.
International Business Machines Corporation
555 Bailey Ave., J8RA/F320
San Jose, CA 95141
Phone: +1 408 463 5659
E-mail: SBrodsky@us.ibm.com

Contact information for members of the co-submitting companies is:

Dr. Kerry Raymond
CRC for Distributed Systems Technology
University of Queensland 4072 Australia
Phone: +61 7 3365 4310
E-mail: kerry@dstc.edu.au

Dr. Stephen Crawley
CRC for Distributed Systems Technology
Phone: +61 7 3365 4310
E-mail: crawley@dstc.edu.au

Simon McBride
CRC for Distributed Systems Technology
Phone: +61 7 3365 4310
E-mail: sjm@dstc.edu.au

Tim Grose
International Business Machines Corporation
E-mail: TGrose@us.ibm.com

Dr. Gene Mutschler
Unisys Corporation
E-mail: Gene.Mutschler@unisys.com

GK Khalsa
Unisys Corporation
E-mail: khalsa@objectrad.com

Ashit Sawhney
Unisys Corporation
E-mail: Ashit.Sawhney@unisys.com

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

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

John Clark
Platinum Technology Inc.
E-mail: clark@platinum.com

Jun Ginbayashi
Fujitsu
E-mail: gin@tokyo.se.fujitsu.co.jp

Philippe Desfray
Softeam
E-mail: phd@softeam.fr

Joan M. Moral
Recerca Informatica
E-mail: uol@arrakis.es

Mario Jeckle
Daimler-Benz Research & Technology
E-mail: mario.jeckle@dbag.ulm.daimlerbenz.com

Contact information for the supporting companies is:

Naresh Bhatia
Cayenne Software [Now acquired by Sterling Software]
E-mail: Bhatian@cayennesoft.com

David Frankel
Genesis Development
E-mail: DFrankel@gendev.com

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

Jack Greenfield
Inline Software
E-mail: Jack@inline-software.com

Magnus Christerson
Rational Software Corporation
E-mail: Christerson@rational.com

Lydia Patterson
Select Software
E-mail: Lydiap@selectst.com

Abdul Akram
Sprint Communications Company
E-mail: Abdul.Akram@mail.sprint.com

Andrew Eisenberg
Sybase, Inc.
E-mail: Andrewe@sybase.com

Dennis Vadura
MCI Systemhouse
E-mail: dvadura@shl.systemhouse.com

Cris Kobryn
MCI Systemhouse
E-mail: ckobryn@acm.org

Woody Pidcock
Boeing
E-mail: Woody.Pidcock@pss.boeing.com

Charlie (CQ) Rehberg
Ardent
E-mail:cq.rehberg@ardentsoftware.com

Johannes Ernst
Aviatis
E-mail:jernst@aviatis.com

Doug Rosenberg
ICONIX
E-mail: doug@iconixsw.com

Chandra Prasad
Integrated Systems
E-mail: cprasad@isi.com

Rodolphe Arthaud
Verilog
E-mail: arthaud@tlse.verilog.fr

J. Hierro
Telefonica I+D
E-mail: jhierro@tid.es

Allen Peralta
Universitat Politecnica de Catalunya
E-mail: peralta@lsi.upc.es

Bruce Mclean
NCR
E-mail: bruce.mclean@sandiegoca.ncr.com

Tsuyoshi Hoshina
Nihon Unisys, Ltd.
E-mail: Tsuyoshi.Hoshina@unisys.co.jp

Wataru Takita
NTT Mutlimedia Networks Labs.
E-mail: takita.wataru@na.tnl.ntt.co.jp

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

Don Baisley, Aditya Bansod, Robert Blum, Dan Chang, Dilhar DeSilva, Keith Duddy, Alexander Glebov, Craig Hayman, Kurt Kirkey, Suresh Kumar, Bruce Mclean and Dave Stringer.

1.4 Status of this Document

This document is the final joint submission to the SMIF RFP. Refer to the OMG web site, http://www.omg.org for additional information and the status of the adoption process.

1.5 Guide to the Submission

This proposal is presented in the following chapters:

Chapter 1 Preface

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

Chapter 2 Proof of Concept

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

Chapter 3 Response to RFP Requirements

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

Chapter 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.

Chapter 5 Usage Scenarios

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

Chapter 6 XMI DTD Design Principles

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

Chapter 7 XML DTD Production

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

Chapter 8 XML Generation Principles

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

Chapter 9 XML Document Production

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

Chapter 10 Compatibility with other standards

Discusses how the XMI specification is related to other industry standards

Chapter 11 Conformance Issues

Discusses conformance - mandatory and optional; compliance points in the XMI specification.

References

Lists the references used in this specification

Glossary

Describes a glossary of terms relevant to the XMI specification.

Index

Index to the submission.

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 XMI cosubmitters 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 metadata repositories, modeling tools, CORBA and the related problems of interchange of metadata across tools in distributed heterogeneous environments. Representative portions of their experience are highlighted below:

- Unisys, IBM, Oracle and Platinum are experienced in the implementation of commercial metadata repositories that have enabled metadata interchange using APIs (proprietary, OMG MOF based, COM based etc.) and file based interchange formats (proprietary, CDIF, MDIS etc.). These metadata repository vendors have already begun prototyping the integration of XMI with their respective products.
- Oracle, Platinum and Select are among the leading modeling and design tool vendors implementing UML and are committing to using XMI as the interchange format for object and data modeling tools and repositories.
- 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. These prototypes include the exporting of UML models from Select Enterprise and importing it into Rational Rose and then exporting the same model into Unisys UREP repository demonstrating model interoperability between tools produced by different vendors.

- Unisys has prototyped and is implementing IDL generation from a MOF based repository and has extended this work to generate both XML DTDs and XML based streams from a MOF 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 that is in the XMI final submission. IBM has also prototyped XMI integration with the TeamConnection repository and the VisualAge for Java Enterprise team support edition.
- 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 stream 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 prototyped with a wide range of realistic meta-models and test cases.
- Oracle has prototyped and is implementing XMI in its design tool and repository products.
- Platinum is working on XMI based interoperability between MOF based repositories and non-MOF repositories.
- The XMI work is based on two key available metadata 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 the MOF model 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 Chapter 6, XMI DTD Design Principles and Chapter 8, XML Generation Principles 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 element identifies the XML version - currently 1.0 as adopted by W3C.

2. The XMI.header element identifies the XMI specification version number - currently 1.0.

3. The XMI.header element 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 the document. Please refer to Chapter 6 XMI DTD Design Principles for details. This submission supports unambiguous identification of all MOF based meta objects using the UUID mechanism.

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 Chapter 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 without loss of information.

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 to XML file to Select Enterprise and IBM TeamConnection to Rational Rose.)

4. Transfer of the complete UML metamodel between tools.

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 extended Unicode) standard.

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. Integration of CDIF and XMI is discussed in Chapter 10, Compatibility With Other Standards.

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. Integration of CDIF and XMI is discussed in Chapter 10, Compatibility with other standards.

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. Integration of CDIF and XMI is discussed in Chapter 10, Compatibility with other standards.

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.

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.

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. UOL is being developed jointly by Recerca Informàtica, Universitat Politècnica de Catalunya and Daimler-Benz Research and Technology.

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 MODL nor UOL is suitable as an interchange format for models in general. The issue of a Human Readable Textual Notation for object models is currently being investigated by the OMG Business Object Domain Task Force.

OCL, which is an optional part of 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 W3C 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.

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 defines 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 application 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 based 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 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 rules for 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 is 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 appropriate. 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.

3.4 Scope of Revision Task Force

The following items are specifically in scope of the XMI revision task force:

- Changes to support revisions to OMG standards and metamodels (MOF, UML, CWM in the future)
- Modifications to take advantage of upcoming XML standards and technology
- Modifications for comprehensive support of data types.

Design Rationale 4

4.1 Design Overview

This submission proposes that the OMG's Stream-based Model Interchange Format for exchanging metadata be based on the W3C's Extensible Markup Language (XML). The XML-based Metadata Interchange (XMI) proposal has two major components:

- The ***XML DTD Production Rules*** for producing XML Document Type Definitions (DTDs) for XMI encoded metadata are specified in Chapters 6 and 7. XMI DTDs serve as syntax specifications for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents.
- The ***XML Document Production Rules*** for encoding metadata into an XML compatible format are specified in Chapters 8 and 9. The production rules can be applied in reverse to decode XMI documents and reconstruct the metadata.

The XMI proposal supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information. The proposal supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

XML is gaining widespread acceptance as the de facto standard for representing structured information in the context of the world-wide web and beyond. Basing the proposed OMG SMIF on XML means that XMI can be used for metadata interchange with and between non-CORBA based metadata repositories and tools.

The XML language is defined by the W3C's "Extensible Markup Language (XML) Recommendation 1.0" document [REC-xml-19980210]. This definition includes a specification of XML in Extended Backus-Naur Form (EBNF) notation. XML is LL(1) parsable.

4.2 XMI and the MOF

XMI is an interchange format for metadata that is defined in terms of 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 defining metadata and representing it as CORBA objects. In this proposal, ***metadata*** is a general term for data that in some sense describes information. The information so described may be information represented in a computer system; e.g. in the form of files, databases, running program instances and so on. Alternatively, the information may be embodied in some system, with the metadata being a description of some aspect of the system such as a part of its design.

The MOF supports any kind of metadata that can be described using Object Modelling techniques. This metadata may describe any aspect of a system and the information it contains, and may describe it to any level of detail and rigour depending on the metadata requirements.

The designers envisaged that the MOF-based metadata will be used in a wide range of CORBA related applications. For example:

- metadata repositories and tools will support the process of analysis, design and development of CORBA-based software,
- metadata repositories will support infrastructure services such as COS Trading, COS Events and ultimately the CORBA Interface Repository itself,
- metadata repositories will support data warehousing, data mining and database interoperability, and
- metadata will be used to describe free-text data sources such as on-line document collections and the world-wide web.

The term ***model*** is generally usually used to denote a description of something, typically something in the real world. The concept of a model is highly fluid, and depends on one's point of view. To someone who is concerned with building or understanding an entire system, a model would include all of the metadata for the system. On the other hand, most people are only concerned with certain components (e.g. programs A and B) or certain kinds of detail (e.g. wiring diagrams) of the system.

In the MOF context, the term ***model*** has a broader meaning. Here, a model is any collection of metadata that is related in the following ways:

- The metadata describes information that is itself related in some way.
- The metadata all conforms to rules governing its structure and consistency; i.e. it has a common ***abstract syntax***.
- The metadata has meaning in a common (often implied) semantic framework.

(Note that a MOF model is not necessarily a model in the usual sense of the word. It does not necessarily describe something in the real world, and it does not necessarily describe things in a way that is interesting to modellers.)

Metadata is itself a kind of information, and can accordingly be described by other metadata. In MOF terminology, metadata that describes metadata is called ***meta-metadata***, and a model that consists of a meta-metadata is called a ***metamodel***.

One kind of metamodel plays a central role in the MOF. A ***MOF metamodel*** defines the abstract syntax of the metadata in the MOF representation of a model. Since there are many possible kinds of metadata in a typical system, the MOF framework needs to support many different MOF metamodels. The MOF integrates these metamodels by defining a common abstract syntax for defining metamodels. This abstract syntax is called the ***MOF Model*** and is model for metamodels; i.e. a meta-metamodel. The MOF metadata framework is typically depicted as a four layer architecture as shown in Table 1 below.

Meta-level	MOF terms	Examples
M3	meta-metamodel	The “MOF Model”
M2	meta-metadata metamodel	UML Metamodel, CWMI Metamodel(s), etcetera
M1	metadata model	UML Models, Warehouse Schemas, etcetera
M0	data	Modelled systems, Warehouse databases, etcetera

Table 1: A Typical OMG Metadata Architecture

A couple points on the OMG / MOF metadata terminology:

- To make things easier to understand, we often describe things in terms of their level in the meta-stack; e.g. the MOF Model is an M3-level model in a 4 level stack.
- The “meta-” prefix should be viewed in a relative rather than absolute sense. Similarly, the numbering of meta-levels is not absolute.
- While there are typically 4 layers in a MOF based metadata stack, the number of layers can more or less than this.

The MOF specification has three core parts; i.e. the specification of MOF Model, the MOF IDL Mapping and the MOF’s interfaces.

The MOF Model

The "MOF Model" is the MOF's built-in meta-metamodel. One can think of it as the "abstract language" for defining MOF metamodels. This is analogous to the way that the UML metamodel is an abstract language for defining UML models. While the MOF and UML are 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. (The alignment of the two models is close enough to allow UML notation to be used to express MOF-based metamodels!)

The three main metadata modelling constructs provided by the MOF are the Class, Association and Package. These are similar to their counterparts in UML, with some simplifications:

- 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 other significant MOF Model constructs 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 elements in a MOF metamodel. 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's "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, and indeed it is largely irrelevant to the problem of model interchange. Instead, we will simply note 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 precise detail so that different vendor implementations of the MOF can generate compatible repository interfaces from a given MOF metamodel. Similarly, the semantic specification of the mapped interfaces allows metadata objects be interoperable.

In addition to the metamodel specific interfaces for the metadata (defined by the IDL mapping), MOF metadata objects share a common set 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 CORBA 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 not of interest to the meta-modeller who will typically 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. In the MOF specification, the MOF Model is defined using the MOF Model as its own modelling language; i.e. it is the "fixed point" of the metadata stack Conceptually, the MOF Model is M3 level metadata conforming to an M4 level metamodel that is isomorphic to the MOF Model. 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 SMIF is to allow the interchange of models in a serialised form. Since the MOF is the OMG's adopted technology for representing metadata, it is natural that the XMI proposal should focus on the interchange of MOF metadata; i.e. metadata that conforms to a MOF metamodel. In fact, XMI is really a pair of parallel mappings

between MOF metamodels and XML DTDs, and between MOF metadata and XML documents.

From the viewpoint users of MOF-based metadata repositories, XMI represents a new 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 new form of metadata interchange that significantly enhances the usefulness of the MOF.

In the wider context, XMI can be viewed as a common metadata interchange format that is independent of middleware technology. Any metadata repository or tool that can encode and decode XMI streams can exchange metadata with other repositories or tools with the same capability. There is no need for to implement the MOF defined CORBA interfaces, or even to “speak” CORBA at all.

XMI provides a possible route for interchange of metadata with repositories whose metamodels are not MOF based. This interchange can be realised by ad hoc mappings between an XMI document and the repository’s native metamodel. Alternatively it can be based on mapping at the meta-metamodel level. For example, interoperability with CDIF-based repositories can be based on a mapping between the MOF Model and the CDIF meta-metamodel.

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 the modelling concepts of UML. This allows the UML graphical notation to be used to express MOF meta-models. The increasing popularity of UML modelling should make an SMIF based on the MOF more accessible than an SMIF based on other meta-modelling concepts.

The second point is that the adopted OMG UML specification defines the UML meta-model as a MOF meta-model. This means that the XMI proposal will lead directly to a model interchange format for UML.

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. Today, HTML (HyperText Markup Language) is the predominant language for expressing web pages. An HTML document consists of the textual content of the document embedded in matched display tags which specify the visual presentation of the content. A well designed HTML document is visually interesting to a human viewer when displayed in a web browser. However, the automatic extraction of information from HTML documents is difficult since HTML tags are designed to express presentation rather than semantic information. This makes HTML a less than ideal medium for general electronic interchange in the Internet.

HTML is a specific tailoring of the more powerful SGML (Standard Generalized Markup Language), a sophisticated tag language which separates view from content and data from metadata. Due to SGML's complexity, and the complexity of the tools required, it has not achieved widespread uptake.

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

XML is a subset of SGML that maintains the important architectural aspects of contextual separation while removing nonessential features. The XML document format embeds the content within tags that express the structure. XML also provides the ability to express rules for the structure (i.e. grammar) of a document. These two features allow automatic separation of data and metadata, and allow generic tools to validate an XML document against its grammar.

Unlike HTML, an XML document does not include presentation information. Instead, an XML document may be rendered for visual presentation 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.

4.3.2 *Benefits of using XML*

There are many advantages in basing an OMG metadata interchange format on XML. These include the following:

- XML is already an open, platform independent and vendor independent standard.
- 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.
- The XML standard itself is programming language-neutral and API-neutral. A range of XML APIs are available, giving the programmer a choice of access methods to create, view, and integrate XML information. Leading XML APIs include DOM, SAX, and Web-DAV.
- The cost of entry for XML information providers is low. XML documents can currently be created by hand using any text editor. In the future, XML-based WYSYWIG editors with support for XSL rendering will allow creation of XML documents. XML's tag structure and textual syntax make it as easy to read as HTML, and is clearly superior for conveying structured information.
- The cost of entry for automatic XML document producers and consumers is low. A growing set of tools is available for XML development. This includes a complete, free, commercially unrestricted XML parser written in Java available from one of the submitting companies (IBM). A variety of other XML support tools including implementations of the XML APIs are available on the Internet.

The XML approach to structured data interchange has been validated through the wide experience with XML itself and with other the members of the XML family: SGML, used in high-end document processing, and HTML, the predominant language of the web.

4.3.3 XML and the Computer Industry

XML is widely believed to be the next step in the evolution of the Web. This is demonstrated by announcements by Netscape and Microsoft that upcoming versions of the leading web browsers Netscape Navigator and Internet Explorer will incorporate XML support. This kind of high profile uptake will enhance the ability of XMI documents based on XML to be integrated into the information Web of the Internet.

While XML is still in its infancy, there are many well documented applications of XML. Example application domains include web commerce, publishing, repositories, modelling, 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**.**

Widespread public interest in XML has lead to a substantial number of books being written. Amazon.com lists 28 books on XML as published in the last year, 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. More advanced XML features are described in sections of the submission which use them.

XML Structure elements

XML documents are tree-based structures of matched tag pairs 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 Document Type Declaration or DTD.

In the simple case, an XML ***tag*** consists of a ***tag name*** enclosed by less-than (‘<‘) and greater-than (‘>’) characters. Tags in an XML document always come in pairs consisting of an opening tag and a closing tag. The closing tag in a pair has the name of the opening tag preceded by a slash symbol. Formally, a balanced tag pair is called an ***element***, and the material between the opening and closing tags is called the element’s ***content***. The following example shows a simple element:

```
<Dog>a description of my dog</Dog>
```

The content of an element may include other elements which may contain other elements in turn. However, at all levels of nesting, the closing tag for each element must be closed before its surrounding element may be closed. This requirement to

balance the tags is what provides XML with its tree data structure and is a key architectural feature missing from HTML.

XML Example

This is a simple example document describing a Car. (New lines and indentation have no semantic significance in XML. They are included here simply to highlight the structure of the example document.)

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

The Car element contains five nested elements which describe it more detail: Make, Model, Year, Color, and Price. The content of each of the nested elements encodes a value in some agreed format.

XML Attributes

In addition to contents, an XML element may contain ***attributes***. Element attributes are expressed in the opening tag of the element as a list of name value pairs following the tag name. For example:

```
<Class xmi.label="c1"> </Class>
```

XML defines a special attribute, the ID, which can be used to attach a unique identifier to an element in the context of a document. These IDs can be used to cross-link the elements to express meaning that cannot be expressed in the confines of XML's strict tree structure. The ID attribute is discussed in detail in the section on XMI Linking, 6.5.1.

Document Type Definitions

A Document Type Definition or DTD is XML's way of defining the syntax of an XML document. An XML DTD defines the different kinds of elements that can appear in a valid document, and the patterns of element nesting that are allowed.

A DTD for the Car example above could contain the following declaration:

```
<!Element Car (Make, Model, Year, Color, Price)>
```

This indicates that for a Car 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 also define the attributes that can be included in an element using an ATTLIST. For example, the following DTD component specifies that every Class element has an optional xmi.label XML attribute and that the xmi.label consists of a character data string: (The #IMPLIED directive indicates that the attribute is optional.)

<!ATTLIST Class xmi.label CDATA #IMPLIED >

While a DTD can be embedded in the document whose syntax it defines, DTDs are typically stored in external files and referenced by the XML document using a Universal Resource Identifier (URI) such as

"http://www.xmi.org/car.dtd"

or

"file:car.dtd"

XML Document Correctness

There are three levels of correctness associated with XML document; well-formedness, validity and semantic correctness:

- A "well-formed" XML document is one where the elements are properly structured as a tree with the opening and closing tags correctly nested. Well-formed documents are essential for information exchange.
- A "valid" XML document is one which is well-formed and that conforms to the structure defined by a DTD. A valid document will only contain elements and attributes defined in the DTD. Similarly, the element contents and attribute values will conform to the DTD. While the DTD need not be specified in an XML document, and an consumer need not to use the DTD when decoding the document, the DTD is essential for checking validity.
- The highest level of document correctness ("semantic correctness") is beyond the scope of XML and DTDs as they are currently defined. Only a XML document consumer with deep domain knowledge can check that the information in an XML document makes sense. In the Car example, this might include a check that a particular Color was available for a given combination of Make, Model, and Year.

4.3.5 XML and the OMG

There is strong synergy between the OMG technologies and XML in a number of areas. OMG defines CORBA as the medium for interchange of data between objects which have (inter-)network connectivity. XML represents a potential alternative interchange medium for cases where ORB to ORB connectivity is not possible. Furthermore, XML presents a possible medium for interchange of data between CORBA based systems and other systems.

The OMG's MOF specification defines a common framework for representing metadata. At the moment, the MOF is restricted is providing metadata for CORBA based systems since the only defined way to interchange MOF metadata is to use the

CORBA interfaces produced by the MOF's IDL mapping. XML (in the form of XMI) provides a way to lift this restriction.

OMG can use the MOF and XMI to expand the significance of the current OMG activities which are producing Domain Service specifications. If a Domain Service specification includes a normative MOF-based metamodel, XMI can then be used to generate a XML DTDs for these metamodels. These DTDs would allow interchange of metadata between and beyond CORBA-based systems, increasing relevance for the Domain Service specifications. There is considerable scope for duplicating this pattern for data interchange.

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 New XML Technologies

The XML family of standards is currently undergoing rapid development. This section gives capsule summaries of important new XML technologies which are in the process of being standardized by the W3C and other organizations. While the XMI submission is designed to be upwards compatible with these technologies, it is rather difficult to use them in this submission. In the future when the technology has stabilized and been standardized it may well be feasible to revise XMI make use of them. XMI has been designed to be upwards compatible with these upcoming XML technologies and provide facilities for their use where possible.

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 in the XMI Linking section 6.8.

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 information based on the entity-relationship model. 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. The latter is particularly significant to XMI, and future incorporation would be of great benefit. XML-Data has been superseded by DCD (Document Content Definition) - a proposal to provide data type support and a new syntax for DTDs.

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 on-line, distributed XML access and modification.

4.4 Major Design Goals and Rationale

This section describes the major design goals that the XMI developers are aiming to meet, and explains some of the more significant design choices that we have made.

4.4.1 Universally Applicable Solution

Design Goal: The XMI submission shall provide the means of interchanging metadata for any MOF metamodel.

The XMI proposal defines DTD generation and stream production rules that can be used to transfer any models described by a MOF metamodel; i.e. any metamodel that is defined in the “abstract language” of the MOF Model. Since the MOF Model is itself described as a MOF metamodel, the proposal also allows interchange of metamodels and even the MOF Model itself.

Table 2 below shows how XMI artifacts fit into the OMG’s four layer metadata architecture. An (M1 level) XMI document is used to transfer an (M1 level) model. This is described by an (M2 level) XML DTD that corresponds to the (M2 level) MOF metamodel for the metadata. For example, a UML model would be encoded against a UML DTD which corresponds to the UML metamodel.

Meta-Level	Metadata	XMI DTDs	XMI Documents
M3	The MOF Model	MOF DTD	
M2	UML MetaModel (and others)	UML DTD (and others)	MOF MetaModel Documents
M1	UML Models (and others)		UML Model Documents (and others)
M0	Instances		

Table 2: XMI and the OMG Metadata Architecture

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 can be encoded against the (M3 level) XML DTD for the (M3 level) MOF Model.

4.4.2 Automatic Generation of Transfer Syntax

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

The classical way of defining a data interchange format is to create a specification document which describes the syntax in BNF or a similar notation, and includes a natural language description of non-syntactic aspects. The problem with this approach is that errors and omissions inevitably creep into the specification. The result is that the person responsible for coding import and export modules needs to "interpret" the specification. Divergence in people's interpretations of a specification often leads to cases where data cannot be exchanged successfully.

The XMI specification is designed to allow the automated generation of XML DTDs based on the original MOF specification of a metamodel. Such DTDs are pretty much guaranteed to be a faithful reflection of the original metamodel. The XMI specification also contains rules for stream production based on the MOF metamodel. These rules can be used to automatically generate XML import and export tools for instances of a metamodel, removing a source of errors and reducing the cost of developing the software needed to support a new metamodel.

4.4.3 Conformance with XML paradigms

Design Goal: As far as is possible, the XMI proposal shall follow XML's established principles for document design.

For example, XML deems it to be "good practice" to produce a DTD that defines the syntax of a document. This allows generic XML tools to validate documents without any hard-wired knowledge of the validity rules for the document. The XMI proposal therefore specifies how XML DTDs which allow validation can be produced.

Similarly, XML's tree-based element structure emphasizes nesting over linkage. The XMI proposal follow's XML's lead by rendering the instances of "contained" Attributes and References in a MOF metamodel as nested XML elements. (The alternative of rendering all Class instances independently and using links to represent all relationships is not the XML pattern of doing things.)

Design Decision: The XMI proposal does not map all MOF DataTypes onto distinct elements in the XMI DTD.

Encoding of MOF DataTypes (i.e. CORBA data types) in XMI presents us with a tricky problem. It is arguable that XMI should map data types so that the XMI DTDs allow full validation. However, if XMI were to do this, there is a substantial risk that future integrate with the XML proposals being developed by W3C would be problematical. XMI therefore optionally encodes most CORBA data types using "boilerplate" DTDs. It is anticipated that this decision will be revisited in the future.

4.4.4 Knowledge of Metamodels

Design Decision: An XMI document consumer or producer needs "knowledge" of the MOF metamodel for the metadata.

An XMI DTD for defines the grammar rules for an XMI document. Unfortunately, XMLs DTD language can only express a subset of the structure and consistency rules contained in a MOF metamodel. For example:

- XML DTDs cannot express the full richness of multiplicities on MOF Attributes and Associations.
- XML DTDs cannot express arbitrary consistency rules as expressed in MOF Constraints.
- The XMI DTD generation rules do not fully render data types.

One consequence of this is that a consumer of an XMI document may need knowledge of the metamodel that is not conveyed in the DTD. This knowledge is needed,

- to check semantic correctness of the document, and
- to reconstruct the metadata in its original form; e.g. with the correct CORBA data types.

Similarly, an XMI document producer needs to be aware to the correct way to encode metadata in areas not covered by the DTD, and needs knowledge of the additional semantic constraints.

For a consumer or producer implemented in the context of a fully functioned MOF framework, this "knowledge" can be obtained by exchanging the MOF metamodel for the metadata and data types.

4.4.5 Complete Encoding of Metadata

Design Goal: Assuming that a consumer has full knowledge of the metamodel, it shall be possible for it to recover all of the source metadata from the XMI document alone.

Since XMI allows interchange of MOF metamodels, this means that it will be 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.)

Full functioned MOF-compliant repositories will be able to use XMI as their sole means of interchange of metadata and meta-metadata. If a MOF repository sends the XMI files for both a model and its metamodel, a receiving MOF repository has sufficient information to fully reconstruct the model, even if it had no prior knowledge of the metamodel. In theory, no other shared infrastructure is necessary.

4.4.6 Correctness of MOF MetaModels

Design Decision: The XMI proposal assumes semantic correctness of the MOF metamodels for the metadata that is to be transmitted.

MOF metamodels expressed in UML notation have a tendency to be under-specified in certain respects that impact on XMI DTD and document production. For example, the names of Associations and AssociationEnds which are mandatory in a MOF metamodel are often omitted from UML class diagrams. Rather than trying to address these issues in the XMI proposal, we make the assumption that any metamodel that is a source or target for XMI interchange is fully compliant to the MOF Model.

4.4.7 Model Fragments

Design Goal: The XMI proposal shall support the interchange of model fragments as well as complete models.

The closure of an entire model often consists of many more model elements than are required by a stream consumer. A consumer may already have many or most of the elements, or alternatively may have no interest in them. In these circumstances, the production, transmission and consumption of redundant or unwanted metadata can be a substantial burden to all parties.

The flexible generation of XMI DTDs, and XMI's use of XML linking makes it possible to use XMI to exchange arbitrary model fragments. XMI's differential metadata interchange (an optional compliance point) is another way to reduce the volume of transmitted metadata. However, it should be noted that any schemes for partial model interchange implicitly relies on the producer and consumer agreeing on what needs to be transmitted. This typically entails some form of user input.

4.4.8 Ill-Formed Models

Design Goal: The XMI proposal shall not require a model to be fully validated as a precondition for metadata interchange.

It would be too restrictive to require a modeller to make a model fully well-formed before it can be transmitted using XMI. Ideas often need to be shared before all the details of a model can filled in. However, a minimum level of correctness is necessary to allow metadata interchange. A model needs all metadata that is necessary to allow a compliant MOF implementation to (re-)construct metaobjects from the XMI document. In particular:

- values of all mandatory attributes must be present,
- implicit constraints on the types of attributes and links must be satisfied,
- maximum cardinality and uniqueness constraints on multi-valued attributes and associations must be satisfied, and
- all "immediate" Constraints in the metamodel must be satisfied.

The above rules are sufficient to ensure that the resulting XMI document (if correctly encoded according to this specification) will be valid according to the XMI DTD for the metamodel. However, the reverse is not true.

This proposal also includes an optional compliance point which supports interchange of incomplete metadata. This is done by relaxing the "multiplicity" specifications in the XMI DTDs to make mandatory elements optional. This feature is provided to make it easier to support models under development and non-MOF tools that do not fully implement a metamodel; e.g. UML tools that implement UML version 0.9 rather than UML 1.1. However, since incomplete models are might create problems in MOF-based repositories and their associated tools, both production and consumption of incomplete metadata documents is an optional XMI compliance point.

4.4.9 Model Versions

Design Goal: The XMI proposal shall support versions of models.

The XMI proposal allows model and metamodel version information to be included in the XMI header. However, it is up to the producers and consumers of XMI streams to manage the allocation of version numbers, and issues associated with compatibility between versions and model lifecycles. It is our recommendation that these issues should be addressed in a future MOF-related RFP.

4.4.10 Model Extensibility

Design Goal: The XMI 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. An XMI document consists of two parts. The first part contains metadata that conforms to a particular 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 in separate, self-contained sections of the XMI document, simplifying its management.

Note however that XMI places no requirement on an XMI document consumer to do anything sensible with metadata corresponding to metadata extensions. A compliant implementation is free to totally discard such metadata if it so desires. However, to support round-trip exchange between heterogeneous tools, (Section 6.10) these sections should be preserved if the document is intended to be shared.

4.4.11 MOF as an Information Model

Design Goal: The XMI proposal shall be capable of being used to transmit operational data as well as metadata.

As was explained earlier, while the typical use of the MOF involves a four layer metadata architecture, there are situations in which only three layers are required. In such cases, the meta-layers are shifted and MOF Model effectively becomes the metamodel for operational data. XMI can then be used as an data interchange medium. Note that this is only appropriate when the MOF Model is suitable for modelling operational data.

4.4.12 Status of MOF and UML DTDs

Design Decision: The MOF and UML DTDs in the appendices of this document are normative for the adopted versions of MOF and UML only.

The SMIF RFP called for proposals to include examples showing how MOF metamodels and UML models can be interchanged. Accordingly, this proposal includes sample XMI DTDs for MOF metamodels and for UML models. The DTDs in this (final) submission have been automatically generated from the MOF Model and the UML metamodel respectively using the normative processes for doing this.

Both the MOF and UML revision task forces are currently active, with final reports due by April 1st 1999. Both task forces have revision issues before them that involve non-trivial changes to their respective metamodels. In the UML case, these include the formal adoption of proposed fixes that make the published UML metamodel a fully MOF compliant metamodel.

In the long term, we propose that the responsibility for producing definitive metamodel and the corresponding DTDs should rest with the groups who propose and maintain the specifications that need them. In the meantime, there is a pressing short term need for MOF and UML metadata interchange. The XMI proposal therefore defines the MOF and UML DTDs provided as appendices as the interim normative XMI DTDs for MOF version 1.1 and UML version 1.1 respectively.

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 modeling 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

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 a brief description of some basic principles, which includes a short description of each XML attribute and XML element defined by XMI. Those descriptions are followed by more complete descriptions that provide examples illustrating the motivation for the XMI DTD design in the areas of metamodel class specification, transmitting incomplete metadata, linking, transmitting metadata differences, and exchanging documents between tools. This chapter concludes by describing the UML DTD included in Appendix A.

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.

6.2 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 valid XMI-transmissible 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.

6.2.1 XML Validation of XMI documents

XML validation can determine whether the XML elements required by this specification are present in the XML document containing 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.

6.2.2 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.

It is permissible for users of XMI to generate a DTD which relaxes the multiplicities described in Section 6.6, “Metamodel Class Specification” to enable incomplete models to be transmitted according to this specification. See Section 6.7, “Transmitting Incomplete Metadata” below for further details.

6.3 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 this chapter.

6.3.1 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.

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 some attributes that must be defined according to this specification. Every XML element that corresponds to a metamodel class must have attributes that enable the XML element to act as a proxy for a local or remote XML element. These attributes are used to associate an XML element with another XML element.

Most of the XML attributes defined by this specification have the prefix "xmi."; however, the XML attributes of XMI elements defined by this specification do not, in general, have that prefix.

6.3.2 Metamodel Class Representation

Every 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 ends relating to 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 (both with and without containment) 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. The content model of the XML element that represents the container class has an XML element with the name of the role at the association end, with the multiplicity defined for its association end. The XML element representing the role has a content model that allows XML elements representing the associated class and any of its subclasses to be included.

6.3.3 Metamodel Extension Mechanism

Every XMI DTD contains a mechanism for extending a metamodel class. Any number of **XMI.extension** elements are 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. Another use might be to transmit data that represents extensions to a metamodel.

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 **XMI.extension** or **XMI.extensions** elements must be declared in either the internal DTD or external DTD.

6.4 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 extended Unicode) standard. Example: <? XML version="1.0" ENCODING=”UCS-2” ?>.
- Any other valid XML processing instructions.
- The required XMI declarations specified in Section 6.5.
- Declarations for a specific metamodel.
- Declarations for differences.
- 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.

6.5 Necessary XMI DTD Declarations

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

6.5.1 Necessary XMI Attributes

Element Identification Attributes

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

These attributes are declared in an XML entity called **XMI.element.att**. Placing these attributes in an XML entity prevents errors in the declarations of these attributes in DTDs. Its declaration is as follows:

```
<!ENTITY % XMI.element.att
                    'xmi.id      ID #IMPLIED
                     xmi.label CDATA #IMPLIED
                     xmi.uuid  CDATA #IMPLIED ' >
```

xmi.id

XML semantics require the values of this attribute to be unique within an XML document; however, the value is not required to be globally unique. This attribute may be used as the value of the **xmi.idref** attribute defined in the next section. It may also be included as part of the value of the **href** attribute in XLinks. An example of the use of this attribute and the other attributes in this section can be found in Section 6.8.3, “Example from UML”.

xmi.label

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

xmi.uuid

The purpose of this attribute is to provide a globally unique identifier for an XML element. The values of this attribute should be globally unique strings prefixed by the type of identifier. For example, to include a DCE UUID as defined by The Open Group, the UUID would be preceded by "**DCE**:". The values of this attribute may be used in the **href** attribute in simple XLinks.

The form of the UUID (Universally Unique Identifier) is taken from a standard defined by the Open Group (was Open Software Foundation). This standard is widely used, including by Microsoft for COM (GUIDs) and by many companies for DCE, which is based on CORBA. The method for generating these 128-bit IDs is published in the standard and the effectiveness and uniqueness of the IDs is not in practice disputed.

When a UUID is placed in an XMI file, the form is "id namespace:uuid." The id namespace of UUIDs is typically DCE. An example is "DCE:2fac1234-31f8-11b4-a222-08002b34c003".

Linking Attributes

XMI requires the use of several XML attributes to enable XML elements to refer to other XML elements using the values of the attributes defined in the previous section. The purpose of these attributes is to allow XML elements to act as simple XLinks or to hold a reference to an XML element in the same document using the XML IDREF mechanism. See section 6.8 on linking.

The attributes described in this section must be included in a DTD as an XML entity. The entity must be declared as follows:

```
<!ENTITY % XMI.link.att
                    'xml:link      CDATA      #IMPLIED
                     inline   (true | false)    #IMPLIED
                     actuate  (show | user)  #IMPLIED
                     href          CDATA       #IMPLIED
                     role          CDATA       #IMPLIED
                     title         CDATA       #IMPLIED
                     show     (embed | replace | new) #IMPLIED
                     behavior    CDATA       #IMPLIED
                     xmi.idref     IDREF       #IMPLIED
                     xmi.uuidref CDATA       #IMPLIED ' >
```

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

Simple XLink Attributes

The first eight attributes declared in the above entity (**xml:link** through **behavior**) enable an XML element to act as a simple XLink according to the XLink and XPointer specifications. The declaration and use of those attributes are defined in the XLink and XPointer specifications. XMI enables the use of simple XLinks. XMI does not preclude the use of extended XLinks. Since the form of extended links is undergoing further development in the XLink specification, no recommendations for their use in XMI are given at this time.

To use simple XLinks, the set **xml:link**="simple". The **href** attribute can be used to reference XML elements whose **xmi.id**, **xmi.label** or **xmi.uuid** attributes are set to particular values. The **xmi.id** attribute value can be specified using a special URI form for XPointers defined in the XLink and XPointer specifications.

xmi.idref

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

xmi.uuidref

This attribute provides a mechanism for referring to another XML element within the same document by using a UUID specified in the **xmi.uuid** attribute of another XML element. The value of this attribute should be the value of a UUID, although XML does not enforce this restriction. **[DCE]**

6.5.2 Common XMI Elements

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

- XMI
- XMI.header
- XMI.content
- XMI.extensions
- XMI.extension
- XMI.documentation
- XMI.owner
- XMI.contact
- XMI.longDescription
- XMI.shortDescription
- XMI.exporter
- XMI.exporterVersion

- XMI.exporterID
- XMI.notice
- XMI.model
- XMI.metamodel
- XMI.metametamodel
- XMI.difference
- XMI.delete
- XMI.add
- XMI.replace
- XMI.reference
- XMI.field
- XMI.struct
- XMI.seqItem
- XMI.sequence
- XMI.arrayLen
- XMI.array
- XMI.enum
- XMI.discrim
- XMI.union
- XMI.any

6.5.3 XMI

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

```
<!ELEMENT XMI (XMI.header,
                    XMI.content?,
                    XMI.difference*,
                    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 submission.

6.5.4 XMI.header

The XMI.header element contains XML elements which identify the model, metamodel, and metametamodel for the metadata, as well as an optional XML element which contains various information about the metadata being transferred. The declaration is:

```
<!ELEMENT XMI.header (XMI.documentation?,
                      XMI.model*,
                      XMI.metamodel*,
                      XMI.metametamodel*) >
```

6.5.5 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 >
```

6.5.6 XMI.extensions

The XMI.extensions 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 >
<!ATTLIST XMI.extensions
        xmi.extender CDATA #REQUIRED
>
```

The **xmi.extender** attribute should indicate which tool made the extension. It is provided so that tools may ignore the extensions made by other tools before the content of the XMI.extensions element is processed.

6.5.7 XMI.extension

The XMI.extension element contains XML elements which also contain metadata that is an extension of the metamodel. This element can be directly included in XML elements in the content section of an XMI document to associate the extension metadata with a particular XML element. Its declaration is:

```
<!ELEMENT XMI.extension ANY >
<!ATTLIST XMI.extension
        %XMI.element.att;
        %XMI.link.att;
        xmi.extender CDATA #REQUIRED
        xmi.extenderID CDATA #IMPLIED
>
```

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

6.5.8 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 >
```

6.5.9 XMI.model

This XML element identifies the model to which the instance data being transferred conforms. There may be multiple models, if the model to which the instance data being transferred conforms to more than one model. This element is expected to become a simple XLink when it becomes a recommendation of the W3C. Its declaration is:

```
<!ELEMENT XMI.model ANY>
<!ATTLIST XMI.model
        %XMI.link.att;
```

```
        xmi.name    CDATA #REQUIRED
        xmi.version CDATA #REQUIRED
>
```

The **xmi.name** and **xmi.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.

6.5.10 XMI.metamodel

This XML element identifies the metamodel to which the model data that is transferred conforms. There may be multiple metamodels, if the model data that is transferred conforms to more than one metamodel. 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. Its declaration is:

```
<!ELEMENT XMI.metamodel ANY>
<!ATTLIST XMI.metamodel
        %XMI.link.att;
        xmi.name    CDATA #REQUIRED
        xmi.version CDATA #REQUIRED
>
```

The **xmi.name** and **xmi.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.

6.5.11 XMI.metametamodel

This XML element identifies the metametamodel to which the metadata that is transferred conforms. For the purposes of XMI this element will always refer to the MOF version which was used. 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. Its declaration is:

```
<!ELEMENT XMI.metametamodel ANY>
<!ATTLIST XMI.metametamodel
        %XMI.link.att;
        xmi.name    CDATA #REQUIRED
        xmi.version CDATA #REQUIRED
>
```

The **xmi.name** and **xmi.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.

6.5.12 XMI.difference

This XML element holds XML elements representing differences to base data. Users may use it within the content part of an XMI file or in a separate XMI.difference section. The attributes in this element allow references to be made to other elements using XLinks, XPointers, or IDREFs. Its declaration is:

```
<!ELEMENT XMI.difference (XMI.difference | XMI.add | XMI.delete |
                          XMI.replace)* >
<!ATTLIST XMI.difference
        %XMI.element.att;
        %XMI.link.att;
>
```

6.5.13 XMI.delete

This XML element represents a deletion to base metadata. It must be within an XMI.difference XML element. The attributes in this element allow references to be made to other elements using XLinks, XPointers, or XML IDREFs. Its declaration is:

```
<!ELEMENT XMI.delete EMPTY >
<!ATTLIST XMI.delete
        %XMI.element.att;
        %XMI.link.att;
>
```

6.5.14 XMI.add

This XML element represents an addition to base metadata. It must be within an XMI.difference XML element. The attributes in this element allow references to be made to other elements using XLinks, XPointers, or XML IDREFs. Its declaration is:

```
<!ELEMENT XMI.add ANY >
<!ATTLIST XMI.add
        %XMI.element.att;
        %XMI.link.att;
         xmi.position CDATA "-1"
>
```

The **xmi.position** attribute indicates where to place the addition relative to other XML elements.

6.5.15 XMI.replace

This XML element represents a replacement of base metadata with other metadata. It must be within an XMI.difference XML element. The attributes in this element allow references to be made to other elements using XLinks, XPointers, or XML IDREFs. Its declaration is:

```
<!ELEMENT XMI.replace ANY >
<!ATTLIST XMI.replace
        %XMI.element.att;
        %XMI.link.att;
        xmi.position CDATA "-1"
>
```

The **xmi.position** attribute indicates where to place the contents of the replacement element relative to other XML elements.

6.5.16 XMI.reference

This XML element allows references to other XML elements within an attribute of type string or an XMI.any element, which represents a data type that is not defined in the metamodel. It should be used within an XMI.any element or in attributes to specify a remote value. Its declaration is:

```
<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference %XMI.link.att; >
```

For information on how to use the link attributes, see the ‘‘Linking’’ section below.

6.5.17 XMI Datatype Elements

All XMI DTDs include some XMI data type elements in them to represent data values so they can be accurately represented and reified. These elements support CORBA data types, include CORBA data type Any, and CORBA TypeCodes.

The declarations of these elements are as follows:

```
<!ELEMENT XMI.TypeDefinitions ANY >

<!ELEMENT XMI.field ANY >

<!ELEMENT XMI.seqItem ANY >

<!ELEMENT XMI.octetStream (#PCDATA) >

<!ELEMENT XMI.unionDiscrim ANY >

<!ELEMENT XMI.enum EMPTY >
<!ATTLIST XMI.enum
        xmi.value CDATA #REQUIRED
>

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
        %XMI.link.att;
        xmi.type CDATA #IMPLIED
        xmi.name CDATA #IMPLIED
```

```
>

<!ELEMENT XMI.CorbaTypeCode (XMI.CorbaTcAlias | XMI.CorbaTcStruct |
                    XMI.CorbaTcSequence | XMI.CorbaTcArray |
                    XMI.CorbaTcEnum | XMI.CorbaTcUnion |
                    XMI.CorbaTcExcept | XMI.CorbaTcString |
                    XMI.CorbaTcWstring | XMI.CorbaTcShort |
                    XMI.CorbaTcLong | XMI.CorbaTcUshort |
                    XMI.CorbaTcUlong | XMI.CorbaTcFloat |
                    XMI.CorbaTcDouble | XMI.CorbaTcBoolean |
                    XMI.CorbaTcChar | XMI.CorbaTcWchar |
                    XMI.CorbaTcOctet | XMI.CorbaTcAny |
                    XMI.CorbaTcTypeCode | XMI.CorbaTcPrincipal |
                    XMI.CorbaTcNull | XMI.CorbaTcVoid |
                    XMI.CorbaTcLongLong |
                    XMI.CorbaTcLongDouble) >
<!ATTLIST XMI.CorbaTypeCode
        %XMI.element.att;
>

<!ELEMENT XMI.CorbaTcAlias (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcAlias
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcStruct (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcStruct
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcField (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcField
        xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcSequence (XMI.CorbaTypeCode |
                    XMI.CorbaRecursiveType) >
<!ATTLIST XMI.CorbaTcSequence
        xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaRecursiveType EMPTY >
<!ATTLIST XMI.CorbaRecursiveType
        xmi.offset CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcArray (XMI.CorbaTypeCode) >
```

```
<!ATTLIST XMI.CorbaTcArray
        xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcObjRef EMPTY >
<!ATTLIST XMI.CorbaTcObjRef
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcEnum (XMI.CorbaTcEnumLabel) >
<!ATTLIST XMI.CorbaTcEnum
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcEnumLabel EMPTY >
<!ATTLIST XMI.CorbaTcEnumLabel
        xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnionMbr (XMI.CorbaTypeCode, XMI.any) >
<!ATTLIST XMI.CorbaTcUnionMbr
        xmi.tcName CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcUnion (XMI.CorbaTypeCode, XMI.CorbaTcUnionMbr*)
>
<!ATTLIST XMI.CorbaTcUnion
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcExcept (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcExcept
        xmi.tcName CDATA #REQUIRED
        xmi.tcId   CDATA #IMPLIED
>

<!ELEMENT XMI.CorbaTcString EMPTY >
<!ATTLIST XMI.CorbaTcString
        xmi.tcLength CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcWstring EMPTY >
<!ATTLIST XMI.CorbaTcWstring
        xmi.tcLength CDATA #REQUIRED
>
```

```
<!ELEMENT XMI.CorbaTcFixed EMPTY >
<!ATTLIST XMI.CorbaTcFixed
        xmi.tcDigits CDATA #REQUIRED
        xmi.tcScale  CDATA #REQUIRED
>

<!ELEMENT XMI.CorbaTcShort EMPTY >

<!ELEMENT XMI.CorbaTcLong EMPTY >

<!ELEMENT XMI.CorbaTcUshort EMPTY >

<!ELEMENT XMI.CorbaTcUlong EMPTY >

<!ELEMENT XMI.CorbaTcFloat EMPTY >

<!ELEMENT XMI.CorbaTcDouble EMPTY >

<!ELEMENT XMI.CorbaTcBoolean EMPTY >

<!ELEMENT XMI.CorbaTcChar EMPTY >

<!ELEMENT XMI.CorbaTcWchar EMPTY >

<!ELEMENT XMI.CorbaTcOctet EMPTY >

<!ELEMENT XMI.CorbaTcAny EMPTY >

<!ELEMENT XMI.CorbaTcTypeCode EMPTY >

<!ELEMENT XMI.CorbaTcPrincipal EMPTY >

<!ELEMENT XMI.CorbaTcNull EMPTY >

<!ELEMENT XMI.CorbaTcVoid EMPTY >

<!ELEMENT XMI.CorbaTcLongLong EMPTY >

<!ELEMENT XMI.CorbaTcLongDouble EMPTY >
```

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

6.6 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 by XML processors.

6.6.1 Fully Qualified Names

The names of each XML element that represent part of a metamodel must be legal XML names and they must be fully qualified. The form of the qualified names is *name1.name2...name*, where *name1* is the name of the outermost package, *name2* is the name of a namespace within the metamodel, and *name* is a name that appears in the namespace of *name2* or in a namespace within the namespace of name2. The "." character is used as a name separator. The names will be legal XML names because MOF names are legal XML element names.

The rest of this chapter contains examples of parts of XMI documents. Simple names are used to make the examples easier to understand; however, in actual XMI files, each XML element name will be the fully qualified name.

6.6.2 Metamodel Multiplicities

In order to represent data in a metamodel, it is necessary to convert multiplicities that occur in the metamodel to XML multiplicities. A mapping between metamodel multiplicities and XML multiplicities can be made which enables XML processors to enforce metamodel multiplicities that match XML multiplicities. There are some metamodel multiplicities that do not match XML multiplicities, and those multiplicities are mapped to XML multiplicity "*". The following table indicates the multiplicity mapping which enforces metamodel multiplicities where possible:

Multiplicity Mapping

Metamodel Multiplicity	XML Multiplicity
0 or 1	?
1 exactly	''
0 or more	*
1 or more	+
other	*

Please note that this multiplicity mapping does not enable the transmission of incomplete metadata; see the section of this chapter that describes transmitting incomplete metadata for details.

6.6.3 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.extension*)? >
<!ATTLIST c
    %XMI.element.att;
    %XMI.link.att;
>
```

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 ‘XMI.extension*, compositionsForC’>

<!ELEMENT c (%cProperties, %cAssociations, %cCompositions)? >
<!ATTLIST c
    %XMI.element.att;
    %XMI.link.att;
>
```

Only the entities that are not empty are included in the content model of element “c” to maintain valid XML syntax.

6.6.4 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.

Multiple inheritance is treated in such a way that the properties, associations, and compositions of classes that occur more than once in the inheritance hierarchy are only included once in their subclasses. For details on how this may be accomplished, please see the DTD production rules.

6.6.5 Attribute Specification

The representation of attributes of metamodel class “c” generally 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*) >

The type specification for an element may come from the metamodel or be defined outside the metamodel. In the former case the type specification is the fully qualified name of the type; in the latter case it is **XMI.any**. If the data is a string type, then its type is mixed, and the specification must take the form:

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

An element is also declared to be of XML type string if the class contains a Tag XMIDataType with Value "string".

When “a” is an attribute with enumerated values or Boolean 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. Attributes of this type are declared as follows:

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

where enum1, enum2, … 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 “a1” is a string type and “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’ >
```

An element is also declared to be of XML type enumeration if the class contains a Tag XMIDataType with Value "enum". The set of allowed enumerated values are given in the Tag XMIEnumSet where the Values are delimited by spaces.

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.

The multiplicities of attributes affect the content of the Properties entity for a class. The multiplicities in the metamodel are mapped to XML multiplicities as defined in the multiplicity section above.

In the previous example, assume that the multiplicity of a1 was one or more, and the multiplicity of a2 was zero or more. Then the Properties entity for class c is declared as follows:

```
<!ENTITY % cProperties ‘a1+, a2*’ >
```

To transmit incomplete metadata using XMI, the multiplicity mappings between MOF and XML are slightly different. See Section 6.7, “Transmitting Incomplete Metadata” for further details.

6.6.6 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 (*content*)? >

The *content* is defined so that XML elements representing the classifier attached to the referenced associationEnd and any of its concrete subclasses may be included in XML element “r”. For example, if class c1 is the classifier attached to the association end r, and it has three subclasses, c2, c3, and c4, and c3 is abstract, the XML element r would be declared as follows:

<!ELEMENT r (c1 | c2 | c3 | c4)? >

The valid multiplicities for “*m*” are defined in the previous section on multiplicities.

6.6.7 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 ‘XMI.extension*, r*m*’ >

Note that “*m*” is defined as in the previous section on multiplicities.

6.7 Transmitting Incomplete Metadata

The goal is to provide a mechanism for transmitting incomplete models (i.e. fragments). An incomplete model is a model which may be missing some information, while maintaining the same structure required for valid models.

For example, a UML class may not have a definition for its "isActive" attribute, yet should still be transmissible using XMI. On the other hand, a UML operation which contained a package would not be a valid incomplete model and could not be transmitted.

This design changes the multiplicity specifications of the auxiliary functions in DTD generation, section 7.2.2. The rules are the same, but the multiplicity character changes.

6.7.1 Interchange of model fragments

In practice, most information is related. The ability to transfer a subset of known information is essential for practical information interchange. In addition, as

information models are developed, they will frequently need to be interchanged before they are complete.

The following guidelines apply for interchanging incomplete models via XMI:

- Information may be missing from a model. The transmission format should not require the addition or invention of new information.
- Model fragments may be disjoint sets. Each set may be transmitted in the same XMI file at the XMI.content level or in different XMI files.
- "Incomplete" indicates a quantity of information less than or equal to "complete." Additional information beyond that which the metamodel prescribes may be transmitted only via the extension mechanism.
- Semantic verification is performed on the metadata that is actually present just as if it was included in complete metadata.
- An additional DTD for incomplete models may be created based on the standard encoding with the following differences.

6.7.2 XMI encoding

The modification to allow fragment interchange is to allow less than but no more than is specified by the metamodel. It is accomplished by changing the mapping between metamodel multiplicities and XML multiplicities. This change in mapping allows a single DTD to be used for both complete and fragment models. The mapping is:

Mapping MetaModel Multiplicities to XML Multiplicities for Incomplete Metadata

Metamodel Multiplicity	XML Multiplicity
0 or 1	?
1 exactly	?
0 or more	*
1 or more	*
other	*

This multiplicity modification allows transmission of less information than the minimum imposed by the metamodel, but does not alter the metamodel structure or allow transmission of additional information beyond that specified by the metamodel.

6.7.3 Example

The following is an example of a valid fragment of a UML model, where the names of the XML elements are simple names to make the example easier to understand (the

fully qualified names must be used to conform to the UML DTD specified in Appendix A):

```
<Model xmi.id="i00000001">
    <name>model1</name>
    <ownedElement>
        <Class xmi.id="i00000002">
            <name>class1</name>
            <feature>
                <Attribute xmi.id="i00000003">
                    <name>attribute1</name>
                    <type><integer/></type>
                </Attribute>
            </feature>
        </Class>
    </ownedElement>
</Model>
```

Please note that the XML elements representing the **visibility**, **isAbstract**, **isLeaf**, and **isRoot** attributes of UML models and classes are not specified, nor is the **isActive** attribute of UML classes. Please note also that the XML element representing **attribute1** does not include an **owner** XML element that would be used to represent the class that contains the attribute.

The following XML elements are the ones that would be required if incomplete transmission of UML models was not allowed:

```
<Model xmi.id="i00000001">
    <name>model1</name>
    <visibility xmi.value="public"/>
    <isAbstract xmi.value="false"/>
    <isLeaf xmi.value="true"/>
    <isRoot xmi.value="true"/>
    <ownedElement>
        <Class xmi.id="i00000002">
            <name>class1</name>
            <visibility xmi.value="public"/>
            <isAbstract xmi.value="false"/>
            <isLeaf xmi.value="true"/>
            <isRoot xmi.value="true"/>
            <isActive xmi.value="true"/>
            <feature>
                <Attribute xmi.id="i00000003">
                    <name>attribute1</name>
                    <visibility xmi.value="public"/>
                    <ownerScope xmi.value="classifier"/>
                    <changeable xmi.value="none"/>
                    <multiplicity>1</multiplicity>
                    <initialValue>0</initialValue>
```

```
                    <owner>
                        <Class xmi.idref="i00000002" />
                    </owner>
                    <type><integer/></type>
                </Attribute>
            </feature>
        </Class>
    </ownedElement>
</Model>
```

6.8 Linking

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

6.8.1 Design principles:

- Links are based on XLinks to navigate to the document (which may be the current document) and XPointers to navigate to the element within the document.
- Links take the same form if the target is within the current or an external document.
- Link definitions are encapsulated in the entity XMI.link.att defined in Section 6.5.1.
- Elements act as a union, where they are either a definition or a proxy. Proxies use the XMI.link.att to define the link, and contain no nested elements.
- XMI.link.att supports external links through the XLink attributes, and internal links through the xmi.id and xmi.uuid attributes.
- Links are always to elements of the same type. Restricting proxies to reference the same element type reduces complexity, enhances reliability and type safety, and promotes caching.
- When acting as a proxy, XML attributes may be defined, but not contents. The XML attributes act as a cache which gives an indication if the link should be followed.
- Proxies may be chained.
- When following the link from a proxy, the definition of the proxy is replaced by the referenced element.
- It is efficient practice for maximizing caching and encapsulation to use local proxies of the same element within a document to link to a single proxy that holds an external reference.
- Association role elements typically contain proxies which link to the definitions of the classes that participate in the association.

6.8.2 Linking

XLinks

When specifying a XLink, the **xml:link** attribute would typically have the value "simple" for unidirectional links. 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".

When navigating into an XML document using an XPointer, the href="XLink|XPointer" form for locating an element by xmi.id is: XLink + "|" + id. For example, **href**="mydoc.xml|xxxx-yyyy..." The form for locating an element by **xml.label** is: XLink + "|descendent(1,type,attribute,value)" where type is the expected element type or "#element" for any type, attribute is the name of the attribute, and value is the name of the attribute. For example, **href**="|descendent(1,#element,**xmi.label**,class1)" . XLink specifies the document to search and is the empty string when using the current document.

IDrefs

The xmi.idref 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.

UUIDrefs

The xmi.uuidref attribute is used for linking using absolute object identity. The UUID specified should correspond to the value of a xmi.uuid within the same document. Although there is no built-in support for UUIDs in XML at this time, it is envisioned that this support will be added in the near future. The xmi.uuidref results in the same action as the XPointer "|descendant(1,#element,xmi.uuid,DCE:abcd-efgh)".

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.

6.8.3 Example from UML

There is an association between ModelElements and Constraints in UML. Operations are a subclass of ModelElements. This example shows an association between Operations and four Constraints with roles constraint and constrainedElement. Qualified names have been suppressed for clarity. Each of the methods of linking is shown. The Constraints are shown in both definition and proxy form.

Document 1:

```
<Operation xmi.id="idO1" xmi.label="op1" xmi.uuid="DCE:1234">
    <constraint>
        <Constraint xmi.id="idC1" xmi.label="co1" xmi.uuid="DCE:abcd">
            <body>First Constraint definition</body>
            <constrainedElement>
                <Operation xmi.uuidref="DCE:1234" />
            </constrainedElement>
        </Constraint>
        <Constraint xmi.idref="idC2" />
        <Constraint xmi.uuidref="DCE:ijkl" />
        <Constraint xml:link="simple" href="doc2.xml|idC4" />
    </constraint>
</Operation>
<Constraint xmi.id="idC2" xmi.label="co2" xmi.uuid="DCE:efgh">
    <body>Second Constraint definition</body>
    <constrainedElement>
        <Operation xmi.idref="idO1" />
    </constrainedElement>
</Constraint>
<Constraint xmi.id="idC3" xmi.label="co3" xmi.uuid="DCE:ijkl">
    <body>Third Constraint definition</body>
    <constrainedElement>
        <Operation href="|descendent(1,Operation,xmi.label,op1)"/>
    </constrainedElement>
</Constraint>
```

Document 2:

```
<Constraint xmi.id="idC4" xmi.label="co4" xmi.uuid="DCE:mnop">
    <body>Fourth Constraint definition</body>
    <constrainedElement>
        <Operation href="doc1.xml|descendent(1,#element,xmi.uuid,abcd)"/>
    </constrainedElement>
</Constraint>
```

The first constraint is a definition. The <constrainedElement> role contains an Operation proxy which has a local reference to the initial Operation definition using **xmi.uuidref**. The second constraint is a proxy referencing a constraint definition using the **xmi.idref** of "idC2." The third constraint is a proxy reference to the definition using **xmi.uuidref** to the constraint "DCE"ijkl". The fourth constraint is an XLink and

XPointer reference proxy to the definition of the constraint using the **xml:link=**"simple" and **href** to the file doc2.xml with id "idC4".

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

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

6.8.4 XMI.reference

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.

XMI.reference can be used for values by pointing to large resources such as bitmaps outside of XML.

6.9 Transmitting Metadata Differences

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

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

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

6.9.1 Definitions:

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

A. New - Old = Difference

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

B. New = Old + Difference

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

6.9.2 Differences

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

- Delete (reference to deleted element): The delete operation refers to a particular element of the old model and specifies a deep removal of the referenced element and all of its contents.
- Add (reference to containing element, new element, optional position): The add operation refers to a particular element of the old model and specifies a deep addition. The element and its contents are added. The contents of the new element are added at the optional position specified, the default being as the last element of the contents. The optional position form is based on XPointer's position form. 1 means the first position, -1 means the last position, and higher numbers count across the contents in the specified direction.
- Replace (reference to replaced element, replacement element, optional position): This operation deletes the old element but not its contents. The new element and its contents are added at the position of the old element. The original contents of the old element are then added to the contents of the new element at the optional position specified, the default being at the end.

6.9.3 XMI encoding

The following are the elements used to encode the differences:

XMI.difference

The XMI.difference element is contained by the XMI.content section of the XMI document. There may be 0 or more XMI.difference elements and each XMI.difference element may contain 0 or more particular differences. The difference element optionally links to the original document to which the differences are applied.

XMI.delete

The XMI.delete element is contained by XMI.difference. Its link attributes contains a link to the element to be deleted.

XMI.add

The XMI.add element is contained by XMI.difference. The contents of XMI.add is the element to be added. The link attributes contain a link to the element to be deleted and an optional position element. The numbering corresponds to XPointer numbering, where 1 is the first and -1 is the last element.

XMI.replace

The XMI.replace element is contained by XMI.difference. The contents of XMI.replace is the element to replace the old element with. The attributes contain a link to the element to be replaced and an optional position element for the replacing element's contents. The numbering corresponds to XPointer numbering, where 1 is the first and -1 is the last element.

6.9.4 Example

This example will delete a class and its attributes, add a second class, and rename a package. Fully qualified names are shortened for clarity.

The original document:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
        <Class xmi.id="ccc" xmi.label="c1">
            <ownedElement>
                <Attribute xmi.label="a1"/>
                <Attribute xmi.label="a2"/>
            </ownedElement>
        </Class>
    </Package>
</XMI.content>
```

The differences document:

```
<XMI.content>
    <XMI.difference href="original.xml">
        <XMI.delete href="original.xml|ccc"/>
        <XMI.add href="original.xml|ppp">
            <Class xmi.label="c2"/>
        </XMI.add>
        <XMI.replace href="original.xml|ppp"/>
            <Package xmi.id="ppp" xmi.label="p2"/>
        </XMI.replace>
    </XMI.difference>
</XMI.content>
```

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

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
    </Package>
</XMI.content>
```

Next, the XMI.add:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p1">
        <Class xmi.label="c2">
        </Class>
    </Package>
</XMI.content>
```

Finally, the XMI.replace:

```
<XMI.content>
    <Package xmi.id="ppp" xmi.label="p2">
        <Class xmi.label="c2">
        </Class>
    </Package>
</XMI.content>
```

6.10 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 **xmi.uuid** and extensions are used together to preserve tool-specific information. In particular, tools may have particular requirements on their IDs which makes ID interchange difficult. Extensions are used to hold tool-specific information, including tool-specific IDs.

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

6.10.1 Definitions:

General:

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

IDs:

- **xmi.uuid** - The universally unique ID of an MC, expressed as the **xmi.uuid** attribute. Example: <Class **xmi.uuid**="ABCDEFGH">

- xmi.extenderID - The tool-specific ID of an MC. The xmi.extenderID is stored in an XMI.extension of the MC when it differs from the **xmi.uuid**.

Tool ID policies:

Every tool is either Open or Closed.

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

6.10.2 Procedures:

Document Creation:

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

Document Import:

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

1. If the importing tool is an Open tool, the **xmi.uuid**s are accepted internally and no conversion is needed.
2. If the importing tool is a closed tool, the tool looks for a contained **XMI.extension** (identified by **xmi.extender**) with a **xmi.extenderID.** If one does not exist, the importing tool creates its own internal id.

Document Modification:

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

6.10.3 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 xmi.label="c1" xmi.uuid="abcdefgh">
</Class>
```

2. The class is imported into Tool2. Tool2 assigns xmi.extenderID "JKLMNOPQRST". A second class is added with name "c2" and xmi.extenderID "X012345678"

3. The model is merged back to XMI:

```
<Class xmi.label="c1" xmi.uuid="abcdefgh">
    <XMI.extension xmi.extender="Tool2" xmi.extenderID="JKLMNOPQRST"/>
</Class>
<Class xmi.label="c2" xmi.uuid="X012345678">
</Class>
```

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

5. The model is merged back to XMI:

```
<Class xmi.label="c1" xmi.uuid="abcdefgh">
    <XMI.extension xmi.extender="Tool2" xmi.extenderID="JKLMNOPQRST"/>
</Class>
<Class xmi.label="c2" xmi.uuid="X012345678">
    <XMI.extension xmi.extender="Tool1" xmi.extenderID="ijklmnop"/>
</Class>
<Class xmi.label="c3" xmi.uuid="qrstuvwxyz">
</Class>
```

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

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

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

6.11 UML DTD

Appendix A contains an automatically generated DTD generated 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.

6.12 General datatype mechanism

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

The domain metamodel is supplemented by adding the domain data type metamodel. These metamodels are connected by adding a relationship between the metamodels. For example, the IDL data type superclass Type_spec could be made a subclass of the UML class DataType.

The data type metamodel is then mapped into XML types. Each "primitive" element of the data type metamodel is mapped to an XML data type. Currently, XML supports two data types, string and enumeration. Future versions of XML are expected to support additional types. The mapping is accomplished by attaching a Tag-Value to the primitive data type.

The Tag XMIDataType indicates that this class is a datatype with XML mapping. If the XMIDataType Value is "string" the XML string datatype is used. If the Value is "enum" the XML enumeration type is used. The set of allowed values is provided by the Value of the XMIEnumSet Tag. The DTD declarations for these types are shown in Section 6.6.5.

In UML, the DataType classes "String" and "Integer" have XMIDataType "string". The class Boolean has XMIDataType "enum" and XMIEnumSet "true false". The

class VisibilityKind has XMIDataType "enum" and XMIEnumSet "public private protected."

XML DTD Production 7

7.1 Purpose

This section describes the rules for creating a DTD from a MOF-based metamodel. Each of the three types of DTDs defined by the rules in this section describes the XML text created by following the rules of Chapter 9, *XML Document Production on page 167*. These rules are derived from informal programs written to implement each of the three methods described in this chapter. A formal reference implementation of the DTD production rules, when and if it appears, would, in all likelihood, include a revision of these rules.

Conformance

Any conformance to the XMI specification is based on generated XML and not 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.

Notation

Since DTD generation programs are not a conformance point of this specification, the rules are expressed as pseudocode rather than any specific programming language. The stylistic guidelines for the DTD generation can be found in Chapter 6, *XMI DTD Design Principles on page 49*.

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.

The names generated using these rules are *qualified* names. A qualified name consists of a sequence of names, separated by the period (".") character. The sequence begins at the outermost container of the item being named and proceeds inward by containment until the item is reached. For further information on qualified names, see Section 6.6, “Metamodel Class Specification.

Note – 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.

Incomplete Models

These rules specify Attribute and Reference multiplicities based on those specified for the corresponding entity in the metamodel. An XMI document generated under these rules for a metamodel instance (model) that is not complete will fail for those cases where there are too few instances of the Attribute or Reference actually present.

Incomplete models can be exchanged by generating a separate DTD with relaxed multiplicity rules for this and other functions specifying multiplicities. The lower bound specifier, "+", must be replaced by "*", and the empty specifier must be replaced by "?". This is shown in the various generation rules below as the "relaxed DTD" case.

For additional information on incomplete data, see Section 6.7, “Transmitting Incomplete Metadata.

Note – An incomplete model, while it can be exchanged between most tools using a relaxed DTD, cannot be loaded into a MOF-compliant repository. A MOF-compliant repository will insist that all required information be present.

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 2:PackageDTD+

To generate a DTD:

```
Generate the FixedContent XMI definitions.
For each Package in the Metamodel not contained by another Package Do
    Generate a PackageDTD(#2).
End
```

2. PackageDTD

A PackageDTD is a sequence of DTD elements of various types, reflecting the contents of the Package. It includes DTD elements describing the Packages and Classes contained in the Package as well as DTD elements for Classifier-level Attributes of the Classes contained in the Package and for the References to compositions made by the Classes of the Package. The rather unusual case of an Association with no References is also handled at the Package level.

2. PackageDTD ::= (2:PackageDTD | 3:ClassDTD
| 4:AttributeElementDef | 7:CompositionDTD
| 10:AssociationDTD)*
9:PackageElementDef

To Generate a PackageDTD:

For Each **Class** of the **Package** Do
    For each **Attribute** of the **Class** Do
        If **isDerived** is **false** Then
            If the **scope** of the **Attribute** is **classifierLevel** Then
                Generate an AttributeElementDef (#4) for the **Attribute**
            End
        End
    End
End
For Each **Association** of the **Package** Do
    If **isDerived** is **false** Then
        If the **Association** contains an **AssociationEnd** whose **aggregation** is **composite**Then
            Generate the CompositionDTD (#7) for the **Association**
        Else If the **Association** has no **References** Then
            Generate the AssociationDTD(#10) for the **Association**
        End
    End
End
For Each **Class** of the **Package** Do
    Generate the ClassDTD (#3) for the **Class**
End
For Each (sub) **Package** of the **Package** Do
    Generate the PackageDTD (#2) for the (sub) **Package**
End
Generate the PackageElementDef (#9) for the **Package**

3. ClassDTD

A ClassDTD is a set of DTD fragments that describe the contents of a Class. These fragments include element definitions for the instance-scope Attributes of the Class and for its non-composition References. The Classifier-scope Attributes of the Class are defined at the level of the Package that contains the Class, as are the composition References which are included in the Class.

3. ClassDTD::= (4:AttributeElementDef | 5:ReferenceElementDef)*
6:ClassElementDef

To Generate a ClassDTD:

For Each **Attribute** of the **Class** Do
  If **isDerived** is **false** Then
    If **scope** is **instanceLevel** then
      Generate the AttributeElementDef (#4) for the **Attribute**
    End
  End
End
For Each **Reference** of the **Class** Do
  If the **isDerived** attribute of the associated **Association** is **false** Then
    If the the **aggregation** of the **AssociationEnd** which is the **exposedEnd** of the **Reference** is not **composite** Then
      Generate the ReferenceElementDef (#5) for the **Reference**
    End
  End
End
Generate the ClassElementDef (#6) for the **Class**

4. AttributeElementDef

An AttributeElementDef is the XML element definition for an Attribute. It gives the name and type for the Attribute. If the attribute type is a data type, then the data value is the element content. If the attribute type is an object, then the actual object will be embedded as the element content.

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

To Generate an AttributeElementDef:

Set AttribName := the qualified name of the **Attribute**.
If the **type** reference refers to a **DataType** Then
    If **DataType.typeCode** is **tk_Boolean** or **tk_enum** Then
        Set AttribContents := 'EMPTY'
        Set AttribAttList := 'value (' + the enumerated values, separated by "|" +
                ')' + '#REQUIRED"
    Else If **DataType.typeCode** is **tk_string** or **tk_wstring** or **tk_char** or **tk_wchar** Then
        Set AttribContents := '(#PCDATA | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_struct** Then
        Set AttribContents := '(XMI.field | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_union** Then
        Set AttribContents := (XMI.unionDiscrim, XMI.field)
    Else If **DataType.typeCode** is **tk_sequence** or **tk_array** Then
        Set AttribContents := '(XMI.octetStream | XMI.seqItem | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_any** Then
        Set AttribContents := '(XMI.any)'
    Else If **DataType.typeCode** is **tk_objref** Then
        Set AttribContents := '(XMI.reference)'
    Else If **DataType.typeCode** is **tk_TypeCode** Then
        Set AttribContents := '(XMI.CorbaTypeCode | XMI.reference)'
    Else
        Set AttribContents := '(#PCDATA | XMI.reference)*
    End
Else the **type** refers to a **Class** Then
    Set AttribContents := '(' + GetClasses(**Class**, '') + ')'
End
Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents, and AttribAttList

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 and the Class which is the type of its referencedEnd. The content model also includes the subclasses of this Class, since any subclass can appear where the Class appears.

The multiplicity of the reference is specified in this rule as well as in GetAllReferences call made by the rule for generation of the ClassElementDef. This duplication allows grouping of references for compactness. Multiple references can be grouped together under one element tag or each reference can be its own element. For example, the Generalizes reference in the GeneralizableElement class of UML can be expressed as:

```
<generalization>
    <Generalization xmi.idref="X1"/>
    <Generalization xmi.idref="X2"/>
</generalization>
```

or as

```
<generalization>
    <Generalization xmi.idref="X1"/>
</generalization>
<generalization>
```

```
    <Generalization xmi.idref="X2"/>
</generalization>
```

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

To generate a ReferenceElementDef:

Set RefName := The qualified name of the **Reference**
Set cls := **Reference.referencedEnd.type** (which is constrained to be a **Class**)
Set m := GetReferenceMultiplicity(the **Reference**)
Set RefContents := '(' + GetClasses(cls, '') + ')' + m
Generate the !ELEMENT definition using RefName and RefContents

6. ClassElementDef

The ClassElementDef for a Class is the XML element definition for the Class. Its content model combines the AttributeElementDefs, ReferenceElementDefs and CompositionElementDefs for the Class. If the Class contains other Classes its content model also refers to the ClassElementDefs for the contained Classes.

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

To Generate a ClassElementDef:

Set ClassName := the qualified name of the **Class**
Set atts := GetAllInstanceAttributes(the **Class**, '')
Set refs := GetAllReferences(this Class, '')
If Length(refs) > 0 Then
    Set refs := '(' + 'XMI.extension' + '*' + ',' + refs + ')'
Else
    Set refs := '(' + 'XMI.extension' + '*' + ')'
End
Set comps1 := GetAllComposedRoles(this Class, '')
Set comps2 := GetContainedClasses(this Class, '')
Set ClassContents to match the pattern:
    atts, refs, comps1, comps2
Remove dangling commas due to empty terms from ClassContents
Set ClassContents := '(' + ClassContents + ')' + '?'
Set ClassAttlistItems := '%XMI.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents and ClassAttlistItems.

7. CompositionDTD

A CompositionDTD is a DTD fragment for an Association which has an AssociationEnd whose aggregation is composite. The CompositionDTD, although defined at the Package level, appears in the content model of the Class that contains

the Reference to the AssociationEnd as an exposedEnd. It also appears in the content models of the subclasses of this Class.

7. CompositionDTD ::= 8:CompositionElementDef

To generate a CompositionDTD:

Generate the CompositionElementDef (#8)

8. CompositionElementDef

The CompositionElementDef is the XML element generated for an Association which has a Reference whose aggregation is composite. It names the Reference and the Class which is the type of its referencedEnd. It also contains the names of the subclasses of this Class, since an instance of one of these can be used wherever the Class is used

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

To Generate a CompositionElementDef:

Set Container := the **Class** containing the **Reference** whose **exposedEnd** is the **AssociationEnd** whose **aggregation** is **composite**.
Set RoleName := the qualified name of the **Reference** in Container.
Set Contained := the **Class** which is **Reference.referencedEnd.type**
Set CompContents := GetClasses(Contained, '')
Set m:= GetReferenceMultiplicity(the **Reference**)
Set CompContents := '(' + CompContents + ')' + m
Generate the !ELEMENT definition using RoleName and CompContents

9. PackageElementDef

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

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

To Generate a PackageElementDef

Set PkgName := the qualified name of the **Package**
Set atts := GetClassLevelAttributes(the **Package**)
Set atts2 := ''
For each **Package** contained in the **Package** Do
    Set temp := GetNestedClassLevelAttributes(the contained **Package**)
    If Length(temp) > 0 Then
        If Length(att2s) > 0 Then
            Set atts2 := '(' + atts2 + ')' + ,
        End
        Set temp := '(' + temp + ')'
    End
    Set atts2 := atts2 + temp
End
Set classes := GetPackageClasses(the **Package**)
Set assns = GetUnreferencedAssociations(the **Package**)
Set pkgs := GetContainedPackages(the **Package**)
Set PkgContents to match the pattern:
    (atts) , (atts2) , (classes | assns | 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.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents and PkgAttlistItems

10. AssociationDTD

An AssociationDTD is generated only for Associations which have no References. Associations with at least one Reference are handled as normal References or Compositions. The AssociationDTD defines elements for the two AssociationEnds of the Association..

10. AssociationDTD ::= 11:AssociationEndDef 11:AssociationEndDef 12: AssociationDef

To Generate an AssociationDTD:

Generate an AssociationEndDef (#11) for the first **AssociationEnd** of the **Association**
Generate an AssociationEndDef (#11) for the second **AssociationEnd** of the **Association**
Generate the AssociationDef (#12) for the **Association**

11. AssociationEndDef

An AssociationEndDef is generated for an AssociationEnd of an Association with no references. It is simply a place holder for a content reference..

11. AssociationEndDef ::= '<!ELEMENT' S EndName S 'EMPTY' '>'
'<!ATTLIST' S EndName S EndAtts'>'

To Generate an AssociationEndDef:

Set EndName := the qualified name of the **AssociationEnd**.
Set EndAtts := '%XMI.link.att;'
Generate the AssociationEndDef using EndName and EndAtts

12. AssociationDef

An AssociationDef is generated for an Association with no References and contains a specification that allows an unlimited number of end1-end2 pairs..

12. AssociationDef ::= '<!ELEMENT' S AssnName S 'AssnContents '>'
'<!ATTLIST' S AssnName S AssnAtts'>'

To Generate an AssociationDef:

Set AssnName := the qualified name of the **Association**.
Set EndAtts := '%XMI.element.att; %XMI.link.att;'
Generate the AssociationDef using AssnName and AssnAtts

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.

GetAllInstanceAttributes

The GetAllInstanceAttributes function produces a string containing the names of all of the non-derived instance-scope Attributes of the given Class, separated by commas to indicate ordering in XML.

The list includes the Attributes defined in the Class itself as well as those in the Class(es) from which it is derived. The Attribute names produced are ordered by the inheritance hierarchy of the Class, with those of any Class appearing after those of its parent Class(es). Within a Class the ordering of Attributes is determined by their ordering in the MOF definition of the Class. In the event of multiple inheritance, one inheritance path is chosen arbitrarily and its set of Attributes appears completely. These are followed by all of the Attributes of another arbitrarily-chosen inheritance path, and so on. The Attributes of a Class in the inheritance hierarchy appear only once, regardless of how many times the Class appears in the hierarchy. The "previousCls" parameter is used to enforce this rule.

The definition of GetAllInstanceAttributes is:

```
Function GetAllInstanceAttributes(in cls : Class,
                                  inout previousCls : String) Returns String
    If cls appears in previousCls, return the empty string
    Set parentAtts := ''
    For each parent Class of cls Do
        Set temp := GetAllInstanceAttributes(parent Class, previousCls)
        If Length(parentAtts) >0 and Length(temp) >0 Then
            Set parentAtts := parentAtts + ','
        End
        Set parentAtts := parentAtts + temp
    End
    Set atts := GetAttributes(cls, 'instance')
    If Length(parentAtts) > 0 and Length(atts) > 0) Then
        Set parentAtts := parentAtts + ','
    End
    Add cls to previousCls
    Return parentAtts + atts
End
```

GetAttributes

The GetAttributes function returns a string containing the names and multiplicities of all of the non-derived Attributes of a Class, separated by commas (",") to indicate their ordering. The ordering should be the same as that in the MOF definition of the Class. The "type" parameter indicates whether the instanceLevel or classifierLevel Attributes are to be returned. Only the Attributes of the Class itself are returned. Inheritance is handled by the caller of this function.

```
Function GetAttributes(in cls : Class, in type: String) Returns String
    Set rslt := ‘’
    For each Attribute of cls, in the order specified by the MOF definition of the Class Do
        If isDerived is false Then
            If (type = ’instance’ And scope is instanceLevel)  Or
              (type = ’classifier’ And scope is classifierLevel) Then
                Set name := Qualified name of the Attribute
                If the multiplicity of the Attribute is “1..*” Then
                    Set m := ‘+’ (or ’*’ for a relaxed DTD)
                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 := ‘’ (or ’?’ for a relaxed DTD)
                End
                If Length(rslt) > 0 Then
                    Set rslt := rslt + ‘,’
                End
                Set rslt := rslt + name + m
            End
        End
    End
    Return rslt
End
```

GetAllReferences

The GetAllReferences function returns a string containing all of the References for the given Class and the Class(es) from which it is derived. The entries on the list are partially ordered. All of the References of a Class appear after those of its parent Class(es), but the ordering of the References within a Class is not specified.

In the case of multiple inheritance, one inheritance path is chosen arbitrarily as the first path and appears completely before any other path. The References of a Class appear only once in the generated list, regardless of how many times the Class appears in the inheritance hierarchy. The "previousCls" parameter is used to enforce this rule.

The definition of GetAllReferences is:

```
Function GetAllReferences(in cls : Class, inout previousCls: String) Returns String
    If cls appears in previousCls, return the empty string
    Set parentRefs := ''
    For each parent Class of cls Do
        Set temp := GetAllReferences(parent Class)
        If Length(parentRefs) > 0 and Length(temp) > 0 Then
            Set parentRefs := parentRefs + ','
        End
        Set parentRefs := parentRefs + temp
    End
    Set refs := GetReferences(cls)
    If Length(refs) > 0 Then
        If Length(parentRefs) > 0 Then
            Set parentRefs := parentRefs + ','
        End
    End
    Add cls to previousCls
    Return parentRefs + refs
End
```

GetReferences

The GetReferences function returns a string containing the names and multiplicities of all of the References of a Class, separated by commas (","). The References should be listed in the order defined in the MOF. Only the References of the Class itself are returned. Inheritance is handled by the caller of this function.

```
Function GetReferences(in cls : Class) Returns String
    Set refs := ''
    For Each Reference contained in cls Do
        If Reference.exposedEnd.aggregation is not composite Then
            Set name := Qualified name of the Reference
            Set m := GetReferenceMultiplicity(Reference)
            Set temp := name + m
            If Length(refs) > 0 Then
                Set refs := refs + ‘,’
            End
            Set refs := refs + temp
        End
    End
    Return refs
End
```

GetReferenceMultiplicity

This function returns a string containing the XML representation of the multiplicity of a Reference. This function relies on the constraint that both ends of an Association cannot be composite. This allows it to be used for both composition and non-composition References.

Note – References by composed objects to the object into which they are composed are optional in an XMI DTD, notwithstanding any specified multiplicity in the metamodel. The XML element of a composed objects in XMI appears as part of the XML element which composes them, making this reference redundant in most cases.

```
Function GetReferenceMultiplicity(in ref:Reference) Returns String
    If Ref.referencedEnd.multiplicity is “0..1” Or
    Ref.referencedEnd.aggregation is composite Then
        Set m := ‘?’
    Else If Ref.referencedEnd.multiplicity is “1..*” Then
        Set m := ‘+’ (or ’*’ for a relaxed DTD)
    Else If Ref.referencedEnd.multiplicity is not “1..1” Then
        Set m := ‘*’
    Else
        Set m := ‘’ (or ’?’ for a relaxed DTD)
    End
    Return m
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.

The definition of GetContainedClasses is:

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

GetAllComposedRoles

The GetAllComposedRoles function returns a string containing the names of References of a Class which place the Class in a containing role in an Association. A Class is in a containing role in an Association if it (or a subclass) contains a Reference whose exposedEnd is an AssociationEnd with an aggregation of composite.

The string produced by this function is partially ordered. All composed roles of a Class appear after the composed roles of its parent Class(es). The composed roles within a particular Class are not ordered. In the event of multiple inheritance, the role names in one arbitrarily-chosen inheritance path appear in their entirety, followed by those of another arbitrarily-chosen path, and so on. No role name appears more than once, regardless of the number of times the referring Class appears. The “previousCls” parameter is used to enforce this rule.

The definition of GetAllComposedRoles is:

```
Function GetAllComposedRoles(in cls : Class, inout previousCls : String) Returns String
    If cls appears in previousCls, return the empty string
    Set parentRoles := ''
    For each parent Class of cls Do
        Set temp := GetAllComposedRoles(parent Class)
        If Length(parentRoles) > 0 and Length(temp) > 0 Then
            Set parentRoles := parentRoles + ','
        End
        Set parentRoles := parentRoles + temp
    End
    Set roles := GetComposedRoles(cls)
    If Length(roles) > 0 Then
        If Length(parentRoles) > 0) Then
            Set parentRoles := parentRoles + ','
        End
    End
    Add cls to previousCls
    Return parentRoles + roles
End
```

GetComposedRoles

The GetComposedRoles function returns a string containing the names of References of a Class which place the Class in a containing role in an Association, separated by commas (",") to indicate ordering. The composed roles should appear in the order they are defined in the MOF. A Class is in a containing role in an Association if it (or a subclass) contains a Reference whose exposedEnd is an AssociationEnd with an aggregation of composite.

```
Function GetComposedRoles(in cls : Class) Returns String
    Set rslt := ''
    For Each Reference of cls Do
        If the aggregation of the AssociationEnd which is exposedEnd of the
        Reference is composite Then
            Set name := Qualified name of the  Reference
            Set m := GetReferenceMultiplicity(the Reference)
            If Length(rslt) > 0 Then
                Set rslt := rslt + ','
            End
            Set rslt :=  rslt + name + m
        End
    End
    Return rslt
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 a number of places where a Class is used and a subclass of the Class may also appear. The prevCls parameter is used to prevent duplication of class names in the event of multiple inheritance.

```
Function GetClasses(in cls : Class, inout prevCls) Returns String
    If cls appears in prevCls, return the empty string ('')
    Set rslt :=  the qualified name of cls
    For Each subclass of cls Do
        Set Temp := GetClasses(the subclass, prevCls)
        If (Length(Temp) > 0) Then
            Set rslt := rslt + '|'
        End
        Set rslt := rslt + Temp
    End
    Add cls to prevCls
    Return rslt
End
```

GetClassLevelAttributes

The GetClassLevelAttributes function produces a string containing the names of all of the non-derived Classifier-scope Attributes of all the Classes of the given Package and any Packages which contain it or generalize it.

The ordering rule is that the Classifier-level Attributes of a Package follow those of its parent or containing Package(s). The ordering of Attributes within a Package are determined by their ordering in the Classes where they are defined.

The definition of GetClassLevelAttributes is:

```
Function GetClassLevelAttributes(in pkg : Package)  Returns String
    If pkg has a parent or containing Package Then
        Set parentAtts := GetClassLevelAttributes(parent Package)
    End
    Set atts := ''
    For Each Class of pkg Do
        Set temp := GetAttributes(the Class,'classifier')
        If Length(temp) > 0 And Length (atts) > 0) Then
            Set atts := atts + '|'
        End
        Set atts := atts + temp
    End
    If Length(atts) > 0) then
        If Length(parentAtts) > 0 Then
            Set parentAtts := parentAtts + ','
        End
        Set atts := '(' + atts + ')'
    End
    Return parentAtts + atts
End
```

GetNestedClassLevelAttributes

The GetNestedClassLevelAttributes 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.

The definition of GetNestedClassLevelAttributes is:

```
Function GetNestedClassLevelAttributes(in pkg : Package) Returns String
    Set rslt := ''
    For each Class of pkg Do
        Set temp := GetAttributes(the Class, 'classifier')
        If Length(temp) > 0 Then
            If Length (rslt) > 0) Then
                Set rslt := rslt + '|'
            End
            Set temp := '(' + temp + ')'
        End
        Set rslt := rslt + temp
    End
    For Each Package of Pkg
        Set childAtts := GetNestedClassLevelAttributes(contained Package)
        If Length(childAtts) > 0 Then
            If Length(rslt) > 0 Then
                Set rslt := '(' + rslt + ')' + ','
            End
            Set childAtts := '('+ childAtts + ')'
        End
        Set rslt := rslt + childAtts
    End
    Return rslt
End
```

GetPackageClasses

The GetPackageClasses function gets all of the Classes in the given Package and any Packages from which it is derived or in which it is contained.

```
Function GetPackageClasses(in pkg : Package) Returns String
    If pkg has a parent or containing Package Then
        Set  parentClasses := GetPackageClasses(parent Package)
    End
    Set classes := ‘’
    For Each  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(in pkg:Package) Returns String
    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 (sub) Package.
        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
```

GetUnreferencedAssociations

This auxiliary function gets all of the Associations of the Package (and its parent packages) that have no References.

```
Function GetUnreferencedAssociations(in pkg: Package) Returns String
    Set parentAssns := ''
    If pkg has a parent Package Then
        Set parentAssns := GetUnreferencedAssociations(parent Package)
    End
    Set assns := ''
    For each Association of pkg Do
        if isDerived is false Then
            If The Association has no References Then
                Set temp := qualified name of the Association
                If Length(assns) > 0) then
                    Set assns := assns + '|'
                End
                Set assns := assns + temp
            End
        End
    End
    If Length(parentAssns >0 ) and Length(assns > 0) Then
        Set parentAssns := parentAssns + '|'
    End
    Return parentAssns + assns
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. The repetition of detail also makes it difficult to perform modifications for the purposes of extension or experimentation. 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 the Classes from which it is inherited, direct or indirect.

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. The savings is about 15 to 20 percent over the Simple DTD rule set. In addition, since the Attributes, References and compositions of an object are defined in only one place, modification is greatly simplified.

This rule set requires somewhat more computational complexity than the Simple DTD rule set. In particular, the DTD generation program must:

- 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 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

The XMI DTD under Rule Set 2 consists of the fixed DTD content which is required for any XMI DTD and the various Package DTD elements. Rule Set 2 adds a set of entity definitions, which must appear before the Package DTD elements, since entities must be defined before their use.

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 are 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 15:EntityDTD 2:PackageDTD+

To generate a DTD:

Generate initial fixed XMI definitions common to all MOF-based metamodel DTDs
Generate the EntityDTD (#15)
Generate the PackageDTD (#2) for each **Package** which is not contained by another **Package**.

2. PackageDTD

A PackageDTD is a sequence of DTD elements of various types, reflecting the contents of the Package. It includes DTD elements describing the Packages and Classes contained in the Package as well as DTD elements for Classifier-level Attributes of the Classes contained in the Package and for the References to compositions made by the Classes of the Package. The rather unusual case of an Association with no References is also handled at the Package level.

2. PackageDTD ::= (2:PackageDTD | 3:ClassDTD
| 4:AttributeElementDTD | 12:CompositionDTD
16:AssociationDTD)*
14:PackageElementDef

To Generate a PackageDTD:

For Each **Class** of the **Package** Do
  For each **Attribute** of the **Class** Do
    If **isDerived** is **false** Then
      If the **scope** of the Attribute is **classifierLevel** Then
        Generate an AttributeElementDTD (#4) for the **Attribute**
      End
    End
  End
End
For Each **Association** of the **Package** Do
  If **isDerived** is **false** Then
    If the **Association** contains an **AssociationEnd** whose **aggregation** is **composite** Then
      Generate the CompositionDTD (#12) for the **Association**
    Else If the **Association** has no **References** Then
      Generate the AssociationDTD(#16) for the **Association**
    End
  End
End
For Each **Class** of the **Package** Do
  Generate the ClassDTD (#3) for the **Class**
End
For Each (sub) **Package** of the **Package** Do
  Generate the PackageDTD (#2) for the (sub) **Package**
End
Generate the PackageElementDef (#14) 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.

3. ClassDTD::= (4:AttributeElementDTD | 7:ReferenceElementDef) * 11:ClassElementDef?

To Generate a ClassDTD:

For Each **Attribute** of the **Class** Do
  If **isDerived** is **false** Then
    If **scope** is **instanceLevel** then
      Generate the AttributeElementDTD (#4) for the **Attribute**
    End
  End
End
For Each **Reference** of the **Class** Do
  If the **isDerived** attribute of the associated **Association** is **false** Then
    If the the **aggregation** of the **AssociationEnd** which is the **exposedEnd** of the **Reference** is not **composite** Then
      Generate the ReferenceElementDef (#7) for the **Reference**
    End
  End
End
Generate the ClassElementDef (#11) for the **Class**

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 ::= 5:AttributeEntityDef? 6:AttributeElementDef

To Generate an AttributeElementDTD:
If the **type** of the Attribute refers to a **DataType** Then
  If the **DataType.typeCode** is Boolean or enum Then
    If an AttributeEntityDef for this type name has not previously been produced, Then
      Generate an AttributeEntityDef (#5) for this **DataType**
    End
  End
End
Generate an AttributeElementDef (#6) 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 S Q 'xmi.value' '(' enumvalues ')' '#REQUIRED' Q '>'

To Generate an AttributeEntityDef:

```
Set TypeName := the qualified name of the DataType
Set enumvalues := ''
For Each possible enumerated value in DataType.typeCode 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 AttribAttList '>')?

To Generate an AttributeElementDef:

Set AttribName := the qualified name of the **Attribute**.
If the **type** reference refers to a **DataType** Then
    If **DataType.typeCode** is **tk_Boolean** or **tk_enum** Then
        Set AttribContents := 'EMPTY'
        Set TypeName := the qualified name of the enumerated type or Boolean
        Set AttribAttList := '%' + TypeName + ';'
    Else If **DataType.typeCode** is **tk_string** or **tk_wstring** or **tk_char** or **tk_wchar** Then
        Set AttribContents := '(#PCDATA | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_struct** Then
        Set AttribContents := '(XMI.field | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_union** Then
        Set AttribContents := (XMI.unionDiscrim, XMI.field)
    Else If **DataType.typeCode** is **tk_sequence** or **tk_array** Then
        Set AttribContents := '(XMI.octetStream | XMI.seqItem | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_any** Then
        Set AttribContents := '(XMI.any)'
    Else If **DataType.typeCode** is **tk_objref** Then
        Set AttribContents := '(XMI.reference)'
    Else If **DataType.typeCode** is **tk_TypeCode** Then
        Set AttribContents := '(XMI.CorbaTypeCode | XMI.reference)'
    Else
        Set AttribContents := '(#PCDATA | XMI.reference)*
    End
Else (the **type** refers to a **Class**)
    Set AttribContents := '(' + GetClasses(**Class**, '') + ')'
End
Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents and AttribAttlist.

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 the Class which is the type of its referencedEnd. The content model also includes the subclasses of this Class, since any subclass can appear where the Class appears.

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

To generate a ReferenceElementDef:

Set RefName := The qualified name of the **Reference**
Set cls := **Reference.referencedEnd.type** (which is constrained to be a **Class**)
Set m := GetReferenceMultiplicity(the **Reference**)
Set RefContents := '(' + GetClasses(cls, '') + ')' + m
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 S Q PropsList Q '>'

To Generate a PropertiesEntityDef:

The PropertiesEntityDef is generated by the OutputEntityDefs2 call (see EntityDTD #15)

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 S Q RefsList Q '>'

To Generate a RefsEntityDef:

The RefsEntityDef is generated by the OutputEntityDefs2 call (see EntityDTD #15)

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 S Q CompsList Q '>'

To Generate a CompsEntityDef:

The CompsEntityDef is generated by the OutputEntityDefs2 call (see EntityDTD #15)

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 ClassAttListItems '>'

To Generate a ClassElementDef:

Set ClassName := the qualified name of the **Class**
Set props := GetPropertiesEntities2(the **Class**, '')
Set refs := GetRefsEntities2(the **Class**, '')
If Length(refs) > 0 Then
    Set refs := '(' +'XMI.extension' + '*' + ', ' + refs + ')'
Else
    Set refs := '(' + 'XMI.extension' +'*'+ ')'
End
Set comps := GetCompsEntities2(the **Class**, '')
Set comps2 := GetContainedClasses(the **Class**, '')
Set ClassContents to match the pattern:
    props , refs , comps, comps2
Remove dangling commas caused by empty terms in ClassContents
Set ClassContents := '(' + ClassContents + ')' + '?'
Set ClassAttlistItems := '%XMI.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents and ClassAttlistItems.

12. CompositionDTD

A CompositionDTD is a DTD fragment for an Association which has an AssociationEnd whose aggregation is composite. The CompositionDTD, although defined at the Package level, appears in the content model of the Class that contains the Reference to the AssociationEnd as an exposedEnd. It also appears in the content models of the subclasses of this Class.

12. CompositionDTD ::= 13:CompositionElementDef

To generate a CompositionDTD:

Generate the CompositionElementDef (#13)

13. CompositionElementDef

The CompositionElementDef is the XML element generated for an Association which has a Reference whose aggregation is composite. It names the Reference and the Class

which is the type of its referencedEnd. It also contains the names of the subclasses of this Class, since an instance of one of these can be used wherever the Class is used.

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

To Generate a CompositionElementDef:

Set Container := the **Class** containing the **Reference** whose **exposedEnd** is the **AssociationEnd** whose **aggregation** is **composite**.
Set RoleName := the qualified name of the **Reference** in Container.
Set Contained := the **Class** which is **Reference.referencedEnd.type**
Set m := GetReferenceMultiplicity(the **Reference**)
Set CompContents := GetClasses(Contained, '')
Set CompContents := '(' + CompContents + ')' + m
Generate the !ELEMENT definition using RoleName and CompContents

14. PackageElementDef

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

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

To Generate a PackageElementDef

```
Set PkgName := the fully qualified name of the Package
Set atts := GetClassLevelAttributes(the Package)
Set atts2 := ''
For each Package contained in the Package Do
    Set temp := GetNestedClassLevelAttributes(the contained Package)
    If Length(temp) > 0 Then
        If Length(att2s) > 0 Then
            Set atts2 := '(' + atts2 + ')' + ,
        End
        Set temp := '(' + temp + ')'
    End
    Set atts2 := atts2 + temp
End
Set classes := GetPackageClasses(the Package)
Set assns := GetUnreferencedAssociations(the Package)
Set pkgs := GetContainedPackages(the Package)
Set PkgContents to match the pattern:
    ( atts ) , ( atts2) , (classes | assns | 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.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents and
PkgAttlistItems
```

15. EntityDTD

Rather then being repeated in the Element definition for a Class and all of its subclasses, the Attributes, References and compositions of the Class are placed into Entity definitions and referenced from the Element definitions of the Class and its subclasses. Changing an Entity definition results in the change appearing in all of these Package. There can be up to three entity definitions for the Class, one each for the Attributes, References and compositions of the Class. If the content of the entity is empty, it need not be present.

15. EntityDTD ::= (8:PropsEntityDef? 9:RefsEntityDef? 10:CompsEntityDef?)+

To Generate an EntityDTD:

Call OutputEntityDefs2 (the topmost **Package** in the metamodel)

16. AssociationDTD

An AssociationDTD is generated only for Associations which have no References. Associations with at least one Reference are handled as normal References or

Compositions. The AssociationDTD defines elements for the two AssociationEnds of the Association..

16. AssociationDTD ::= 17:AssociationEndDef 17:AssociationEndDef
18: AssociationDef

To Generate an AssociationDTD:

Generate an AssociationEndDef (#17) for the first **AssociationEnd** of the **Association**
Generate an AssociationEndDef (#17) for the second **AssociationEnd** of the **Association**
Generate the AssociationDef (#18) for the **Association**

17. AssociationEndDef

An AssociationEndDef is generated for an AssociationEnd of an Association with no references. It is simply a place holder for a content reference..

17. AssociationEndDef ::= '<!ELEMENT' S EndName S 'EMPTY' '>'
'<!ATTLIST' S EndName S EndAtts'>'

To Generate an AssociationEndDef:

Set EndName := the qualified name of the **AssociationEnd**.
Set EndAtts := '%XMI.link.att;'
Generate the AssociationEndDef using EndName and EndAtts

18. AssociationDef

An AssociationDef is generated for an Association with no References and contains a specification that allows an unlimited number of end1-end2 pairs..

18:AssociationDef ::= '<!ELEMENT' S AssnName S 'AssnContents '>'
'<!ATTLIST' S AssnName S AssnAtts'>'

To Generate an AssociationDef:

Set AssnName := the qualified name of the **Association**.
Set EndAtts := '%XMI.element.att; %XMI.link.att;'
Generate the AssociationDef using AssnName and AssnAtts

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. Functions referenced which do not end in "2" are defined in the Simple DTD rule set, and their definitions are not repeated here.

OutputEntityDefs2

This function controls the definition of all entity definitions in the EntityDTD for the metamodel. It must first be called for the outermost Package in the model; it calls itself recursively for other Packages in the metamodel. It finds those Classes which are not derived from any other Class and calls the entity definition functions (OutputPropertiesEntityDef2, OutputRefsEntityDef2 and OutputCompsEntityDef2) for these Classes. These functions call themselves recursively for every subclass of these Classes, thereby generating all required entity definitions in the proper order.

```
Subroutine OutputEntityDefs2(in pkg: Package)
    For each Class in pkg Do
        If the Class.supertype is null Then
            Call OutputPropertiesEntityDef2 (the Class, ”, ”)
            Call OutputRefsEntityDef2(the Class, ”, ”)
            Call OutputCompsEntityDef2(the Class, ”, ”)
        End
    End
    For each Package contained in pkg Do
        Call OutputEntityDefs2(the Package)
    End
End
```

OutputPropertiesEntityDef2

The OutputPropertiesEntityDef2 function is a recursive function that creates an Entity definition for the instance-level Attributes of a Class and then calls itself to generate those for all of the subclasses of the Class. This Entity definition consists of a listing of all of the instance-level Attributes for the Class. It is possible for the entity content to be empty; if so, the entity is not generated. This fact is remembered so that the entity will not be referenced.

The prevCls parameter is used to insure that the function does not attempt to generate the PropertiesEntityDef more than once, which would otherwise happen in inheritance hierarchies including multiple inheritance.

The function is defined as follows:

```
Subroutine OutputPropertiesEntityDef2(in cls: Class, inout prevCls: String)
    If cls appears in prevCls, Then
        Return the empty string (’’)
    End
    Set PropsEntityName := the qualified name of the Class + ’Properties’
    Set PropsList := GetAttributes(cls, ’instance’)
    If Length(PropsList) > 0) Then
        Generate the PropertiesEntityDef (#8), using PropsEntityName and PropsList
        Remember that an entity was generated for cls
    End
    Add cls to prevCls
    For each subclass of cls Do
        Call OutputPropertiesEntityDef2(the subclass, prevCls)
    End
End
```

OutputRefsEntityDef2

The OutputRefsEntityDef2 function is similar to OutputPropertiesEntityDef2, except that it produces a set of RefsEntitiesDefs instead of PropertiesEntityDefs.

```
Subroutine OutputRefsEntityDef2(in cls: Class, inout prevCls: String,)
    If cls appears in prevCls, Then
        Return the empty string ('')
    End
    Set RefsEntityName := the qualified name of the Class + 'Associations'
    Set RefsList := GetReferences(cls)
    If Length(RefsList) > 0 Then
        Set RefsList := '('+ RefsList + ')'
        Generate the RefsEntityDef (#9), using RefsEntityName and RefsList
        Remember that an entity def was generated for cls
    End
    Add cls to prevCls
    For each subclass of cls Do
        Call OutputRefsEntityDef2(the subclass, prevCls)
    End
End
```

OutputCompsEntityDef2

The OutputCompsEntityDef2 function is similar to OutputPropertiesEntityDef2, except that it produces a set of CompsEntitiesDefs instead of PropertiesEntityDefs.

```
Subroutine OutputCompsEntityDefEntityDTD(in cls: Class, inout prevCls: String)
    If cls appears in prevCls, Then
        Return the empty string ('')
    End
    Set CompsEntityName := the qualified name of the Class + 'Compositions'
    Set CompsList := GetComposedRoles(cls)
    If Length(CompsList) > 0 Then
        Set CompsList := '(' + CompsList + ')'
        Generate the CompsEntityDef (#9), using CompsEntityName and CompsList
        Remember that an entity was generated for cls
    End
    Add cls to prevCls
    For each subclass of cls Do
        Call OutputCompsEntityDef2(the subclass
    End
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 classes := ‘’
    For Each  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
    Return classes
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 Then
        Return the empty string (’’)
    End
    Set parentProps := the empty string (’’)
    For each parent Class of cls Do
        Set temp := GetPropertiesEntities2(the parent Class, prevCls)
        If Length (temp) > 0 Then
            If Length(parentProps) > 0 Then
                Set parentProps := parentProps + ’,’
            End
            Set parentProps := parentProps + temp
        End
    End
    Set ClassName := the qualified name of cls
    Set props := the empty string (’’)
    If a property ENTITY was generated for cls (see #8) Then
        If Length (parentProps) > 0 Then
            Set parentProps := parentProps + ’,’
        End
        Set props := ‘%’ + ClassName + ’Properties’ + ’;’
    End
    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 Then
        Rreturn the empty string (’’)
    End
    Set parentRefs := the empty string (’’)
    For each parent Class of cls Do
        Set temp := GetRefsEntities2(the parent Class, previousCls)
        If Length (temp) > 0) Then
            If Length (parentRefs) > 0 Then
                Set parentRefs := parentRefs + ’,’
            End
            Set parentRefs := parentRefs + temp
        End
    End
    Set ClassName := the qualified name of cls
    Set refs := the empty string (’’)
    If a References ENTITY was generated for cls (See #9) Then
        If Length(parentRefs) > 0 Then
            Set parentRefs := parentRefs + ’,’
        End
        Set ref :=  ‘%’ + ClassName + ’Associations’ + ’;’
    End
    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 Then
        Return the empty string (’’)
    End
    Set parentComps := the empty string (’’)
    For each parent Class of cls Do
        Set temp := GetCompsEntities2(the parent Class, previousCls)
        If Length(temp) > 0 Then
            If Length(parentComps) > 0 Then
                Set parentComps := parentComps + ’,’
            End
            Set parentComps := parentComps + temp
        End
    End
    Set ClassName := the  qualified name of cls
    Set comps := the empty string (’’)
    If a compositions !ENTITY was generated for cls Then
        If Length (parentComps) > 0 Then
            Set parentComps := parentComps + ’,’
        End
        Set comps := ‘%’ + ClassName + ’Compositions’ + ’;’
    End
    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. The size of the generated DTD is approximately the same as that in Rule Set 2.

A more complete description of the design principles used in this Rule Set can be found in Section 6.6, “Metamodel Class Specification.

This rule set requires much more computational complexity than the Simple DTD rule set and somewhat 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 avoid duplication of entities in cases of multiple inheritance, 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

The XMI DTD under Rule Set 3 consists of the fixed DTD content which is required for any XMI DTD, the initial set of entity definitions and the various Package DTD elements.

Note – 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 are placed in the content model of this root element.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 15:EntityDTD 2:PackageDTD+

To generate a DTD:

Generate initial fixed XMI definitions common to all MOF-based metamodel DTDs
Generate the EntityDTD (#15).
Generate the PackageDTD (#2) elements for each **Package** which is not contained by another **Package**.

2. PackageDTD

A PackageDTD is a sequence of DTD elements of various types, reflecting the contents of the Package. It includes DTD elements describing the Packages and Classes contained in the Package as well as DTD elements for Classifier-level Attributes of the Classes contained in the Package and for the References to compositions made by the Classes of the Package. The rather unusual case of an Association with no References is also handled at the Package level.

2. PackageDTD ::= (2:PackageDTD | 3:ClassDTD
| 4:AttributeElementDTD | 12:CompositionDTD
| 16:AssociationDTD)*
14:PackageElementDef

To Generate a PackageDTD:

For Each **Class** of the **Package** Do
For each **Attribute** of the **Class** Do
If **isDerived** is **false** Then
If the **scope** of the Attribute is **classifierLevel** Then
Generate an AttributeElementDTD (#4) for the **Attribute**
End
End
End
End
For Each **Association** of the **Package** Do
If **isDerived** is **false** Then
If the **Association** contains an **AssociationEnd** whose **aggregation** is **composite** Then
Generate the CompositionDTD (#12) for the **Association**
Else If the **Association** has no **References** Then
Generate the AssociationDTD(#16) for the **Association**
End
End
End
For Each **Class** of the **Package** Do
Generate the ClassDTD (#3) for the **Class**
End
For Each (sub) **Package** of the **Package** Do
Generate the PackageDTD (#2) for the (sub) **Package**
End
Generate the PackageElementDef (#14) for the **Package**

3. ClassDTD

A ClassDTD is a set of DTD fragments containing type information for non-derived instance-scope Attributes of the Class and the References that it makes. These are in addition to entity definitions that summarize the Attributes, References and compositions of the Class.

3. ClassDTD ::= (4:AttributeElementDTD | 7:ReferenceElementDef)* 11:ClassElementDef?

To Generate a ClassDTD:

For Each **Attribute** of the **Class** Do
  If **isDerived** is **false** Then
    If **scope** is **instanceLevel** Then
      Generate the AttributeElementDTD (#4) for the **Attribute**
    End
  End
End
For Each **Reference** of the **Class** Do
  If the **isDerived** attribute of the associated **Association** is **false** Then
    If the the **aggregation** of the **AssociationEnd** which is the **exposedEnd** of the **Reference** is not **composite** Then
      Generate the ReferenceElementDef (#7) for the **Reference**
    End
  End
End
Generate the ClassElementDef (#11) for the **Class**

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 ::= 5:AttributeEntityDef? 6:AttributeElementDef

To Generate an AttributeElementDTD:

If the **type** of the Attribute refers to a **DataType** Then
  If the **DataType.typeCode** is Boolean or enum Then
    If an AttributeEntityDef for this type name has not previously been produced, Then
      Generate an AttributeEntityDef (#5) for this **DataType**
    End
  End
End
Generate an AttributeElementDef (#6) 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 S Q 'xmi.value' '(' enumvalues ')' '#REQUIRED' Q '>'

To Generate an AttributeEntityDef:

```
Set TypeName := the name of the DataType
Set enumvalues := ''
For Each possible enumerated value of DataType.typeCode 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 AttribAttList '>')?

To Generate an AttributeElementDef:

Set AttribName := the qualified name of the **Attribute**.
If the **type** reference refers to a **DataType** Then
    If **DataType.typeCode** is **tk_Boolean** or **tk_enum** Then
        Set AttribContents := 'EMPTY'
        Set TypeName := the name of the enumerated type or Boolean
        Set AttribAttList := '%' + TypeName + ';'
    Else If **DataType.typeCode** is **tk_string** or **tk_wstring** or **tk_char** or **tk_wchar** Then
        Set AttribContents := '(#PCDATA | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_struct** Then
        Set AttribContents := '(XMI.field | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_union** Then
        Set AttribContents := (XMI.unionDiscrim, XMI.field)
    Else If **DataType.typeCode** is **tk_sequence** or **tk_array** Then
        Set AttribContents := '(XMI.octetStream | XMI.seqItem | XMI.reference)*'
    Else If **DataType.typeCode** is **tk_any** Then
        Set AttribContents := '(XMI.any)'
    Else If **DataType.typeCode** is **tk_objref** Then
        Set AttribContents := '(XMI.reference)'
    Else If **DataType.typeCode** is **tk_TypeCode** Then
        Set AttribContents := '(XMI.CorbaTypeCode | XMI.reference)'
    Else
        Set AttribContents := '(#PCDATA | XMI.reference)*
    End
Else (the **type** refers to a **Class**)
    Set AttribContents := '(' + GetClasses(**Class**, '') + ')'
End
Generate the !ELEMENT and !ATTLIST definitions using AttribName, AttribContents and AttribAttlist.

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 cls := **Reference.type** (which constrained to be a **Class**)
Set m := GetReferenceMultiplicity(the **Reference**)
Set RefContents := '(' + GetClasses(cls, '') + ')' + m
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 expands to the Attributes of the Class(es) from which it is derived.

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

To Generate a PropertiesEntityDef:

The PropertiesEntityDef is generated by OutputEntityDefs3 call (see EntityDTD #15)

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 S Q RefsList Q '>'

To Generate a RefsEntityDef

The RefsEntityDef is generated by OutputEntityDef s3 call (see EntityDTD #15)

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 S Q CompsList Q '>'

To Generate a CompsEntityDef:

The CompsEntityDef is generated by the OutputEntityDefs3 call (See EntityDTD #15)

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 or subtypes of the AssociationEnds which is the referencedEnds of composition References of the Class.

In this Rule Set, the ClassElementDef consists simply of up to three entity invocations rather than a complete listing of Attributes, References and composition roles. These

entities summarize this information instead. The entity invocations do not appear if they would be empty.

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

To Generate a ClassElementDef:

Set ClassName := the qualified name of the **Class**
Set props := ''
If a properties entity was generated for this **Class** Then
    Set props := '%' + ClassName + 'Properties' + ";"
End
Set refs :=
If a References entity was generated for this **Class** Then
    Set refs := '|' + '%' + ClassName + 'Associations' + ';'
End
Set refs := '(' + ' XMI.extension' + '*' + refs + ')'
Set comps := ''
If a comps entity was generated for this **Class** Then
    Set comps := '%' + ClassName + 'Compositions' + ';'
End
Set comps2 := GetContainedClasses(the **Class**, '')
Set ClassContents to match the pattern:
    props , refs, comps1, comps2
Remove dangling commas caused by empty terms in ClassContents
If Length(ClassContents) = 0) then
    ClassContents := 'EMPTY'
Else
    ClassContents := '(' + ClassContents + ')' + '?'
End
Set ClassAttlistItems :='%XMI.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using ClassName, ClassContents and ClassAttlistItems.

12. CompositionDTD

A CompositionDTD is a DTD fragment for an Association which has an AssociationEnd whose aggregation is composite. The CompositionDTD, although defined at the Package level, appears in the content model of the Class that contains the Reference to the AssociationEnd as an exposedEnd. It also appears in the content models of the subclasses of this Class.

12. CompositionDTD ::= 13:CompositionElementDef

To generate a CompositionDTD:

Generate the CompositionElementDef (#13)

13. CompositionElementDef

The CompositionElementDef is the XML element generated for an Association which has a Reference whose aggregation is composite. It names the Reference and the Class which is the type of its referencedEnd. It also contains the names of the subclasses of this Class, since an instance of one of these can be used wherever the Class is used.

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

To Generate a CompositionElementDef:

Set Container := the **Class** containing the **Reference** whose **exposedEnd** is the **AssociationEnd** whose **aggregation** is **composite**.
Set RoleName := the qualified name of the **Reference** in Container.
Set Contained := the **Class** which is **Reference.referencedEnd.type**
Set CompContents := GetClasses(Contained, '')
Set CompContents := '(' + CompContents + ')'
Generate the !ELEMENT definition using RoleName and CompContents

14. PackageElementDef

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

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

To Generate a PackageElementDef

```
Set PkgName := the fully qualified name of the Package
Set atts := GetClassLevelAttributes(the Package)
Set atts2 := ''
For each Package contained in the Package Do
    Set temp := GetNestedClassLevelAttributes(the contained Package)
    If Length(temp) > 0 Then
        If Length(att2s) > 0 Then
            Set atts2 := '(' + atts2 + ')' + ,
        End
        Set temp := '(' + temp + ')'
    End
    Set atts2 := atts2 + temp
End
Set classes := GetPackageClasses(the Package)
Set assns := GetUnreferencedAssociations (the Package)
Set pkgs := GetContainedPackages(the Package)
Set PkgContents to match the pattern:
    ( atts ) , ( atts2) , (classes | assns | 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.element.att; %XMI.link.att;'
Generate the !ELEMENT and !ATTLIST definitions using PkgName, PkgContents and
PkgAttlistItems
```

15. EntityDTD

The EntityDTD portion of the DTD consists of the entity definitions for all Classes of all Packages in the metamodel. This is managed by a single function, OutputEntityDefs3, since the Class inheritance hiearchy (ies) does (do) not necessarily follow Package boundaries, and the process must start at the parent Class(es) of the hiearchy(ies).

15. EntityDTD ::= (8:PropertiesEntityDef | 9:RefsEntityDef | 10:CompsEntityDef)+

To Generate the EntityDTD:

```
Call OutputEntityDefs3(the topmost  Package in the metamodel)
```

16. AssociationDTD

An AssociationDTD is generated only for Associations which have no References. Associations with at least one Reference are handled as normal References or

Compositions. The AssociationDTD defines elements for the two AssociationEnds of the Association..

16. AssociationDTD ::= 17:AssociationEndDef 17:AssociationEndDef
18: AssociationDef

To Generate an AssociationDTD:

Generate an AssociationEndDef (#17) for the first **AssociationEnd** of the **Association**
Generate an AssociationEndDef (#17) for the second **AssociationEnd** of the **Association**
Generate the AssociationDef (#18) for the **Association**

17. AssociationEndDef

An AssociationEndDef is generated for an AssociationEnd of an Association with no references. It is simply a place holder for a content reference..

17. AssociationEndDef ::= '<!ELEMENT' S EndName S 'EMPTY' '>'
'<!ATTLIST' S EndName S EndAtts'>'

To Generate an AssociationEndDef:

Set EndName := the qualified name of the **AssociationEnd**.
Set EndAtts := '%XMI.link.att;'
Generate the AssociationEndDef using EndName and EndAtts

18. AssociationDef

An AssociationDef is generated for an Association with no References and contains a specification that allows an unlimited number of end1-end2 pairs..

18:AssociationDef ::= '<!ELEMENT' S AssnName S 'AssnContents '>'
'<!ATTLIST' S AssnName S AssnAtts'>'

To Generate an AssociationDef:

Set AssnName := the qualified name of the **Association**.
Set EndAtts := '%XMI.element.att; %XMI.link.att;'
Generate the AssociationDef using AssnName and AssnAtts

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.

OutputEntityDefs3

This function controls the definition of all entity definitions in the EntityDTD for the metamodel. It must first be called for the outermost Package in the model; it calls itself recursively for Packages that are enclosed in Packages. It finds those Classes which are not derived from any other Class and calls the entity definition functions (OutputPropertiesEntityDef3, OutputRefsEntityDef3 and OutputCompsEntityDef3) for these Classes. These functions call themselves recursively for every subclass of these Classes, thereby generating all required entity definitions in the proper order.

```
Subroutine OutputEntityDefs3(in pkg: Package)
    For each Class in pkg Do
        If the Class.supertype is null Then
            Call OutputPropertiesEntityDef3 (the Class, '', '')
            Call OutputRefsEntityDef3(the Class, '', '')
            Call OutputCompsEntityDef3(the Class, '', '')
        End
    End
    For each Package contained in pkg Do
        Call OutputEntityDefs3(the Package)
    End
End
```

OutputPropertiesEntityDef3

The OutputPropertiesEntityDef3 function is a recursive function that creates an Entity definition for the instance-level Attributes of a Class and then calls itself to generate those for all of the subclasses of the Class. This Entity definition consists of a listing of the instance-level Attributes for the Class itself, plus a reference to the Properties entity of the Class from which it is derived. If the Class is derived from more than one Class, there is still only one entity reference. The Attributes from the additional parent Class and those of its parents are listed separately in their entirety, except for those which would appear in the expansion of the entity. This avoids multiple definition of Attributes should the inheritance tree for the additional parent Class intersect that of the first parent Class. It is possible for the entity content to be empty; if so, the entity is not generated. This fact is remembered so that the entity will not be referenced.

The prevCls parameter is used to insure that the function does not attempt to generate the PropertiesEntityDef more than once, which would otherwise happen in inheritance hierarchies including multiple inheritance.

The baseCls parameter is used to detect multiple inheritance and provide the control mechanism for the inclusion of the Attributes from the additional inheritance hierarchy(ies). It is a list of Classes filled in with the Classes encountered as the function goes *down* the inheritance hierarchy. When multiple inheritance is detected, the algorithm proceeds *up* the second (and other) inheritance hierarchy(ies) until a Class in baseCls is encountered. It stops at this point, since the Attributes from this Class and its parents already appear as part of the entity invocation generated for the first parent. Note that baseCls is refreshed prior to calling each subclass, since the inheritance harriers is different for each.

The function is defined as follows:

```
Subroutine OutputPropertiesEntityDef3(in cls: Class, inout prevCls: String,
                                      inout baseCls: String)
    If cls appears in prevCls, Then
        Return the empty string ('')
    End
    Set PropsEntityName := the qualified name of the Class + 'Properties'
    Set temp := baseCls
    Set PropsList := GetAllInstanceAttributes3(cls, temp)
    If Length(PropsList) > 0) Then
        Set PropsList := '(' + PropsList + ')'
        Generate the PropertiesEntityDef (#8), using PropsEntityName and PropsList
        Remember that an entity was generated for cls
    End
    Add cls to baseCls
    Set temp := baseCls
    Add cls to prevCls
    For each subclass of cls Do
        Set baseCls := temp
        Call OutputPropertiesEntityDef3(the subclass, prevCls, baseCls)
    End
End
```

OutputRefsEntityDef3

The OutputRefsEntityDef3 function is similar to OutputPropertiesEntityDef3, except that it produces a set of RefsEntitiesDefs instead of PropertiesEntityDefs.

```
Subroutine OutputRefsEntityDef3(in cls: Class, inout prevCls: String,
                                inout baseCls: String)
    If cls appears in prevCls, Then
        Return the empty string ('')
    End
    Set RefsEntityName := the qualified name of the Class + 'Associations'
    Set temp := baseCls
    Set RefsList := GetAllReferences3(cls, temp)
    If Length(RefsList) > 0 Then
        Set RefsList := '(' + RefsList + ')'
        Generate the RefsEntityDef (#9), using RefsEntityName and RefsList
        Remember that an entity def was generated for cls
    End
    Add cls to baseCls
    Set temp := baseCls
    Add cls to prevCls
    For each subclass of cls Do
        Set baseCls := temp
        Call OutputRefsEntityDef3(the subclass, prevCls, baseCls)
    End
End
```

OutputCompsEntityDef3

The OutputCompsEntityDef3 function is similar to OutputPropertiesEntityDef3, except that it produces a set of CompsEntitiesDefs instead of PropertiesEntityDefs.

```
Subroutine OutputCompsEntityDef3(in cls: Class, inout prevCls: String,
                                 inout baseCls: String)
    If cls appears in prevCls, Then
        Return the empty string ('')
    End
    Set CompsEntityName := the qualified name of the Class + 'Compositions'
    Set temp := baseCls
    Set CompsList := GetAllComposedRoles3(cls, temp)
    If Length(CompsList) > 0 Then
        Set CompsList := '(' + CompsList + ')'
        Generate the CompsEntityDef (#10), using CompsEntityName and CompsList
        Remember that an entity was generated for cls
    End
    Add cls to baseCls
    Set temp := baseCls
    Add cls to prevCls
    For each subclass of cls Do
        Set baseCls := temp
        Call OutputCompsEntityDef3(the subclass, prevCls, baseCls)
    End
End
```

GetAllInstanceAttributes3

The GetAllInstanceAttributes3 function returns a string containing the name of the Properties entity of the parent Class of the given class plus all of the non-derived instance-level attributes of the Class itself.

In the case of multiple inheritance, this function invokes a multiple-inheritance management function to gets the Attributes from the parent Classes in the second (and any additional) set of parent Classes. These are between the parent Properties entity and the Attributes of the Class itself.

```
Function GetAllInstanceAttributes3(in cls : Class,  in baseCls: String) Returns String
    Set parentEntity := ''
    Set parentContents := ''
    For each Class referenced by cls.supertype Do
        If cls.supertype is in baseCls Then (it is the first inheritance tree)
            If an entity was generated for cls.supertype Then
                Set parentEntity := '%' + the qualified name of cls.supertype  +
                'Properties;'
            End
        Else (it is in another inheritance tree)
            Set temp := GetParentAttributes3(cls.supertype, baseCls)
            If Length(temp) >  0 and Length(parentContents) > 0) Then
                Set parentContents := parentContents + ','
            End
            Set parentContents := parentContents + temp
        End
    End
    If Length(parentEntity) >0 and Length(parentContents) > 0 Then
        Set parentEntity := parentEntity + ','
    End
    Set parentContents := parentEntity + parentContents
    Set temp := GetAttributes(cls, 'instance')
    If Length (temp) > 0 and Length(parentContents) > 0 Then
        Set parentContents := parentContents + ','
    End
    Return parentContents + temp
End
```

GetAllReferences3

The GetAllReferences3 is similar to the GetAllInstanceAttributes3 function, except that it generates References instead of Attributes.

```
Function GetAllReferences3(in cls : Class, in baseCls: String) Returns String
    Set parentEntity := ''
    Set parentContents := ''
    For each Class referenced by cls.supertype Do
        If cls.supertype is in baseCls Then (it is the first inheritance tree)
            If an entity was generated for cls.supertype Then
                Set parentEntity := '%' + the qualified name of cls.supertype +
                 'Associations;'
            End
        Else (it is in another inheritance tree)
            Set temp := GetParentReferences3(cls.supertype, baseCls)
            If Length(temp) >  0 and Length(parentContents) > 0) Then
                Set parentContents := parentContents + ','
            End
            Set parentContents := parentContents + temp
        End
    End
    If Length(parentEntity) >0 and Length(parentContents) > 0 Then
        Set parentEntity := parentEntity + ','
    End
    Set parentContents := parentEntity + parentContents
    Set temp := GetReferences(cls)
    If Length (temp) > 0 Then
        If Length(parentContents) > 0 Then
            Set parentContents := parentContents + ','
        End
        Set temp := '(' + temp + ')'
    End
    Return parentContents + temp
End
```

GetAllComposedRoles3

The GetAllComposedRoles3 function is similar to the GetAllInstanceAttributes3 function, except that it deals with "composed roles" instead of Attributes.

```
Function GetAllComposedRoles3(in cls : Class, in baseCls: String) Returns String
    Set parentEntity := ''
    Set parentContents := ''
    For each member of cls.supertype Do
        If cls.supertype is in baseCls Then (it is the first inheritance tree)
        If an entity was generated for cls.supertype Then
                Set parentEntity := '%' + qualified name of cls.supertype +
                 'Compositions;'
            End
        Else (it is in another inheritance tree)
            Set temp := GetParentCompositionRoles3(cls.supertype, baseCls)
            If Length(temp) >  0 and Length(parentContents) > 0) Then
                Set parentContents := parentContents + ','
            End
            Set parentContents := parentContents + temp
        End
    End
    If Length(parentEntity) >0 and Length(parentContents) > 0 Then
        Set parentEntity := parentEntity + ','
    End
    Set parentContents := parentEntity + parentContents
    Set temp := GetComposedRoles(cls)
    If Length (temp) > 0 and Length(parentContents) > 0 Then
        Set parentContents := parentContents + ','
    End
    Return parentContents + temp
End
```

GetParentAttributes3

This is an auxiliary function used by GetAllInstanceAttributes3 to produce the list of Attributes in parent Classes of a Class, up to the point where a parent Class is encountered which has already been processed.

```
Function GetParentAttributes3(in cls: Class, in baseCls: String) : Return String
    If cls is in baseCls Then
        Return the empty string (’’)
    End
    Set parentContents := ’’
    For each Class referenced by cls.supertype Do
        Set temp := GetParentAttributes3(cls.supertype, baseCls)
        If Length(temp) > 0 and Length(parentContents) > 0 Then
            Set parentContents := parentContents + ’,’
        End
        Set parentContents := parentContents + temp
    End
    Set temp := GetAttributeContents(cls, ’instance’)
    If Length(temp) > 0 and Length(parentContents > 0) Then
        Set parentContents := parentContents + ’,’
    End
    Return parentContents + temp
End
```

GetParentReferences3

This function is similar to GetParentAttributes, except that is called by GetAllReferences3.

```
Function GetParentReferences3(in cls: Class, in baseCls: String) : Return String
    If cls is baseCls Then
        Return the empty string (’’)
    End
    Set parentContents := ’’
    For each Class referenced by cls.supertype Do
        Set temp := GetParentReferences3(cls.supertype, baseCls)
        If Length(temp) > 0 and Length(parentContents) > 0 Then
            Set parentContents := parentContents + ’,’
        End
        Set parentContents := parentContents + temp
    End
    Set temp := GetReferences(cls)
    If Length(temp) > 0 and Length(parentContents > 0) Then
        Set parentContents := parentContents + ’,’
    End
    Return parentContents + temp
End
```

GetParentCompositionRoles3

This function is similar to GetParentAttributes3, except that is called by GetAllComposedRoles3.

```
Function GetParentCompositionRoles3(in cls: Class, in baseCls: String) : Return String
    If cls is in baseCls Then
        Return the empty string (’’)
    End
    Set parentContents := ’’
    For each Class referenced by cls.supertype Do
        Set temp := GetParentCompositionRoles3(cls.supertype, baseCls)
        If Length(temp) > 0 and Length(parentContents) > 0 Then
            Set parentContents := parentContents + ’,’
        End
        Set parentContents := parentContents + temp
    End
    Set temp := GetCompositionContents(cls)
    If Length(temp) > 0 and Length(parentContents > 0) Then
        Set parentContents := parentContents + ’,’
    End
    Return parentContents + temp
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 at the beginning of the generated DTD. 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 "Metamodel Class Specification" on page 64..

The FixedContent elements are:

```
<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI is the top-level XML element for XMI transfer text       -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI (XMI.header, XMI.content?, XMI.difference*,
                  XMI.extensions*) >
<!ATTLIST XMI
              xmi.version CDATA #FIXED "1.0"
              timestamp CDATA #IMPLIED
              verified (true | false) #IMPLIED >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.header contains documentation and identifies the model,  -->
<!-- metamodel, and metametamodel                                 -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.header (XMI.documentation?, XMI.model*, XMI.metamodel*,
                          XMI.metametamodel*) >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- documentation for transfer data                              -->
<!-- ____________________________________________________________ -->
<!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 >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.element.att defines the attributes that each XML element  -->
<!-- that corresponds to a metamodel class must have to conform to -->
<!-- the XMI specification.                                       -->
<!-- ____________________________________________________________ -->
<!ENTITY % XMI.element.att
                 'xmi.id ID #IMPLIED
                  xmi.label CDATA #IMPLIED
                  xmi.uuid
                  CDATA #IMPLIED ' >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.link.att defines the attributes that each XML element that -->
<!-- corresponds to a metamodel class must have to enable it to     -->
<!-- function as a simple XLink as well as refer to model           -->
<!-- constructs within the same XMI file.                           -->
<!-- ____________________________________________________________ -->
<!ENTITY % XMI.link.att
             'xml:link CDATA #IMPLIED
              inline (true | false) #IMPLIED
              actuate (show | user) #IMPLIED
              href CDATA #IMPLIED
              role CDATA #IMPLIED
              title CDATA #IMPLIED
              show (embed | replace | new) #IMPLIED
              behavior CDATA #IMPLIED
              xmi.idref IDREF #IMPLIED
              xmi.uuidref CDATA #IMPLIED' >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.model identifies the model(s) being transferred          -->
<!-- ____________________________________________________________ -->
```

```
<!ELEMENT XMI.model ANY >
<!ATTLIST XMI.model
              %XMI.link.att;
              xmi.name      CDATA #REQUIRED
              xmi.version   CDATA #IMPLIED >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.metamodel identifies the metamodel(s) for the transferred -->
<!-- data                                                         -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.metamodel ANY >
<!ATTLIST XMI.metamodel
              %XMI.link.att;
              xmi.name      CDATA #REQUIRED
              xmi.version   CDATA #IMPLIED >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.metametamodel identifies the metametamodel(s) for the    -->
<!-- transferred data                                             -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.metametamodel ANY >
<!ATTLIST XMI.metametamodel
              %XMI.link.att;
              xmi.name      CDATA #REQUIRED
              xmi.version   CDATA #IMPLIED >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.content is the actual data being transferred             -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.content ANY >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- XMI.extensions contains data to transfer that does not conform -->
<!-- to the metamodel(s) in the header                            -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.extensions ANY >
<!ATTLIST XMI.extensions
              xmi.extender CDATA #REQUIRED >

<!-- ____________________________________________________________ -->
```

```
<!--                                                                    -->
<!-- extension contains information related to a specific model     -->
<!-- construct that is not defined in the metamodel(s) in the      -->
<!-- header                                                         -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.extension ANY >
<!ATTLIST XMI.extension
              %XMI.element.att;
              %XMI.link.att;
              xmi.extender    CDATA #REQUIRED
              xmi.extenderID CDATA #REQUIRED >

<!-- ____________________________________________________________ -->
<!--                                                                -->
<!-- XMI.difference holds XML elements representing differences to -->
<!-- a base model                                                   -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.difference (XMI.difference | XMI.delete | XMI.add |
                              XMI.replace)* >
<!ATTLIST XMI.difference
              %XMI.element.att;
              %XMI.link.att; >

<!-- ____________________________________________________________ -->
<!--                                                                -->
<!-- XMI.delete represents a deletion from a base model            -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.delete EMPTY >
<!ATTLIST XMI.delete
              %XMI.element.att;
              %XMI.link.att; >

<!-- ____________________________________________________________ -->
<!--                                                                -->
<!-- XMI.add represents an addition to a base model                 -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.add ANY >
<!ATTLIST XMI.add
              %XMI.element.att;
              %XMI.link.att;
              xmi.position CDATA "-1" >

<!-- ____________________________________________________________ -->
<!--                                                                -->
```

```
<!-- XMI.replace represents the replacement of a model construct   -->
<!-- with another model construct in a base model                  -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.replace ANY >
<!ATTLIST XMI.replace
              %XMI.element.att;
              %XMI.link.att;
              xmi.position CDATA "-1" >

<!-- ____________________________________________________________ -->
<!--                                                               -->
<!-- XMI.reference may be used to refer to data types not defined  -->
<!-- in the metamodel                                              -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.reference ANY >
<!ATTLIST XMI.reference
              %XMI.link.att; >

<!-- ____________________________________________________________ -->
<!--                                                              -->
<!-- This section contains the declaration of XML elements        -->
<!-- representing data types                                      -->
<!-- ____________________________________________________________ -->
<!ELEMENT XMI.TypeDefinitions ANY >

<!ELEMENT XMI.field ANY >
<!ELEMENT XMI.seqItem ANY >

<!ELEMENT XMI.octetStream (#PCDATA) >

<!ELEMENT XMI.unionDiscrim ANY >

<!ELEMENT XMI.enum EMPTY >
<!ATTLIST XMI.enum
              xmi.value CDATA #REQUIRED
>

<!ELEMENT XMI.any ANY >
<!ATTLIST XMI.any
              %XMI.link.att;
              xmi.type CDATA #IMPLIED
              xmi.name CDATA #IMPLIED
>
```

```
<!ELEMENT XMI.CorbaTypeCode (XMI.CorbaTcAlias | XMI.CorbaTcStruct |
                                XMI.CorbaTcSequence | XMI.CorbaTcArray |
                                 XMI.CorbaTcEnum | XMI.CorbaTcUnion |
                                XMI.CorbaTcExcept | XMI.CorbaTcString |
                                XMI.CorbaTcWstring | XMI.CorbaTcShort |
                                 XMI.CorbaTcLong | XMI.CorbaTcUshort |
                                 XMI.CorbaTcUlong | XMI.CorbaTcFloat |
                                 XMI.CorbaTcDouble |
                                 XMI.CorbaTcBoolean |
                                 XMI.CorbaTcChar | XMI.CorbaTcWchar |
                                 XMI.CorbaTcOctet | XMI.CorbaTcAny |
                                 XMI.CorbaTcTypeCode |
                                 XMI.CorbaTcPrincipal |
                                 XMI.CorbaTcNull | XMI.CorbaTcVoid |
                                 XMI.CorbaTcLongLong |
                                 XMI.CorbaTcLongDouble) >
<!ATTLIST XMI.CorbaTypeCode
              %XMI.element.att; >

<!ELEMENT XMI.CorbaTcAlias (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcAlias
              xmi.tcName CDATA #REQUIRED
              xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcStruct (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcStruct
              xmi.tcName CDATA #REQUIRED
              xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcField (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcField
              xmi.tcName CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcSequence (XMI.CorbaTypeCode |
                                   XMI.CorbaRecursiveType) >
<!ATTLIST XMI.CorbaTcSequence
              xmi.tcLength CDATA #REQUIRED >

<!ELEMENT XMI.CorbaRecursiveType EMPTY >
<!ATTLIST XMI.CorbaRecursiveType
              xmi.offset CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcArray (XMI.CorbaTypeCode) >
<!ATTLIST XMI.CorbaTcArray
```

```
                xmi.tcLength CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcObjRef EMPTY >
<!ATTLIST XMI.CorbaTcObjRef
                xmi.tcName CDATA #REQUIRED
                xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcEnum (XMI.CorbaTcEnumLabel) >
<!ATTLIST XMI.CorbaTcEnum
                xmi.tcName CDATA #REQUIRED
                xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcEnumLabel EMPTY >
<!ATTLIST XMI.CorbaTcEnumLabel
                xmi.tcName CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcUnionMbr (XMI.CorbaTypeCode, XMI.any) >
<!ATTLIST XMI.CorbaTcUnionMbr
                xmi.tcName CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcUnion (XMI.CorbaTypeCode,
                                XMI.CorbaTcUnionMbr*) >
<!ATTLIST XMI.CorbaTcUnion
                xmi.tcName CDATA #REQUIRED
                xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcExcept (XMI.CorbaTcField)* >
<!ATTLIST XMI.CorbaTcExcept
                xmi.tcName CDATA #REQUIRED
                xmi.tcId   CDATA #IMPLIED >

<!ELEMENT XMI.CorbaTcString EMPTY >
<!ATTLIST XMI.CorbaTcString
                xmi.tcLength CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcWstring EMPTY >
<!ATTLIST XMI.CorbaTcWstring
                xmi.tcLength CDATA #REQUIRED >

<!ELEMENT XMI.CorbaTcFixed EMPTY >
<!ATTLIST XMI.CorbaTcFixed
                xmi.tcDigits CDATA #REQUIRED
                xmi.tcScale  CDATA #REQUIRED >
```

```
<!ELEMENT XMI.CorbaTcShort EMPTY >
<!ELEMENT XMI.CorbaTcLong EMPTY >
<!ELEMENT XMI.CorbaTcUshort EMPTY >
<!ELEMENT XMI.CorbaTcUlong EMPTY >
<!ELEMENT XMI.CorbaTcFloat EMPTY >
<!ELEMENT XMI.CorbaTcDouble EMPTY >
<!ELEMENT XMI.CorbaTcBoolean EMPTY >
<!ELEMENT XMI.CorbaTcChar EMPTY >
<!ELEMENT XMI.CorbaTcWchar EMPTY >
<!ELEMENT XMI.CorbaTcOctet EMPTY >
<!ELEMENT XMI.CorbaTcAny EMPTY >
<!ELEMENT XMI.CorbaTcTypeCode EMPTY >
<!ELEMENT XMI.CorbaTcPrincipal EMPTY >
<!ELEMENT XMI.CorbaTcNull EMPTY >
<!ELEMENT XMI.CorbaTcVoid EMPTY >
<!ELEMENT XMI.CorbaTcLongLong EMPTY >
<!ELEMENT XMI.CorbaTcLongDouble EMPTY >
```

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 Meta Object Facility (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 Chapter 9, *XML Document Production on page 167*.

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

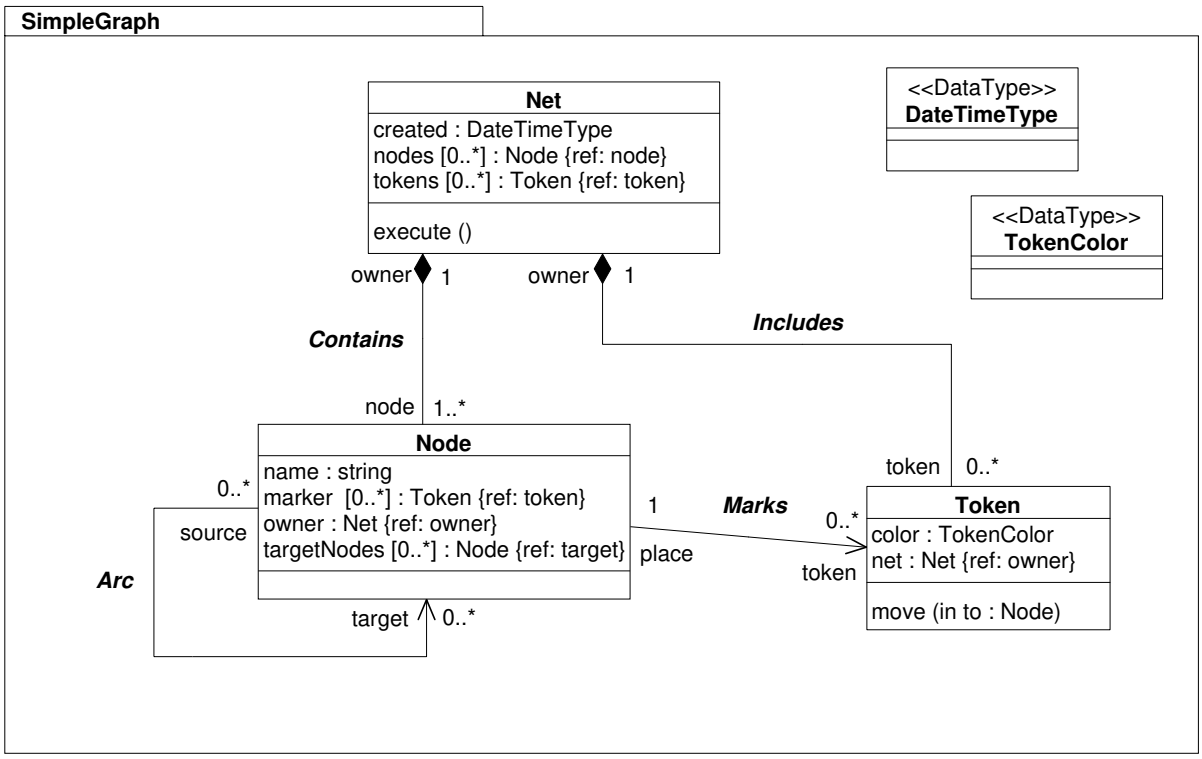

Figure 8-1 A very simple metamodel for graph modeling

in exchanging model information, provides two distinct methods of specifying the modeling elements which are used to generate an XML document.

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, for example, 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 of which it is the root.

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 154 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 155 (with much detail omitted).

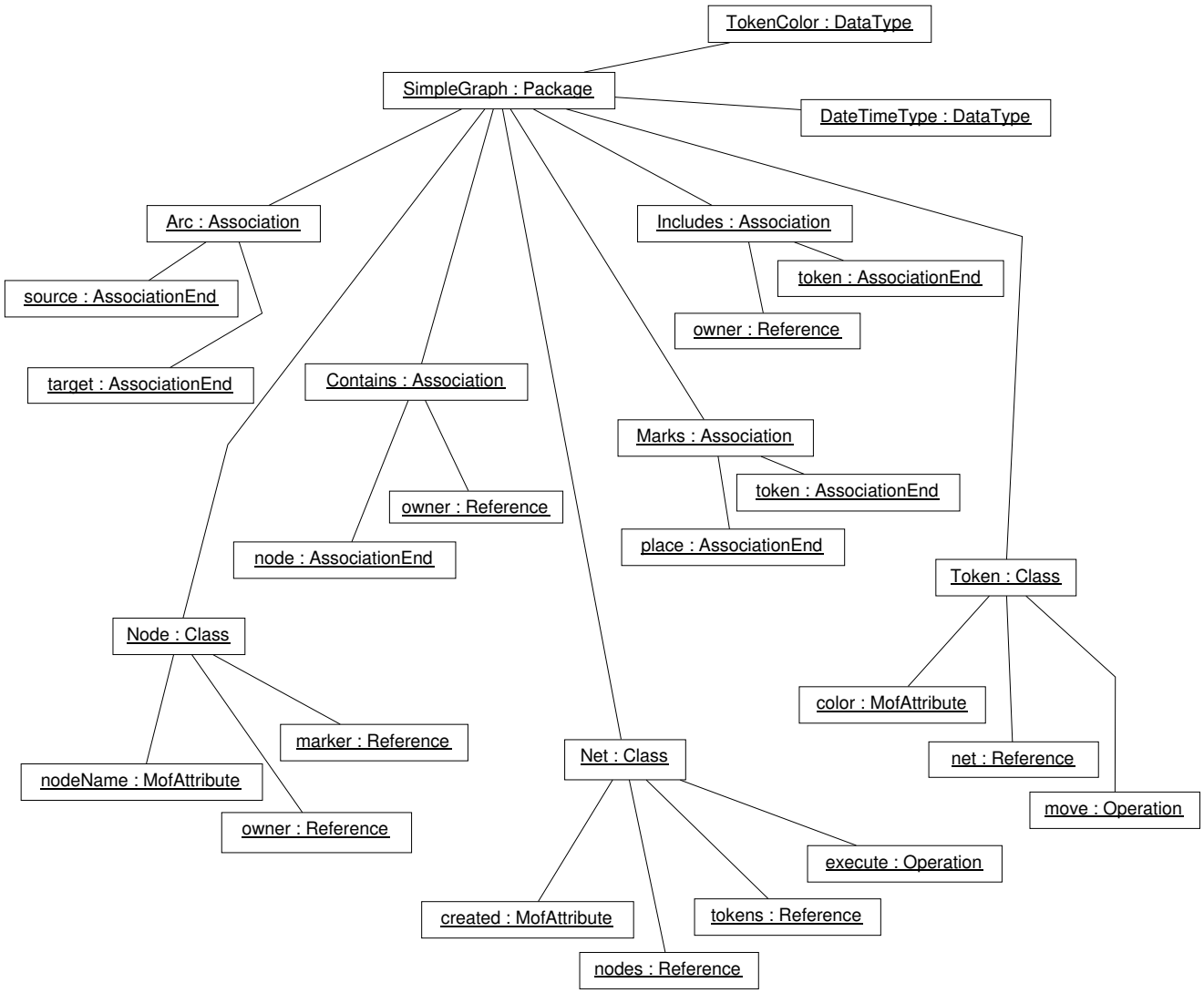

Figure 8-2 Object diagram showing simple metamodel as an instance of the MOF Model

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 notation in Figure 8-3 on page 156. As instances of the metamodel elements, the same model would form the object diagram in Figure 8-4 on page 156.

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

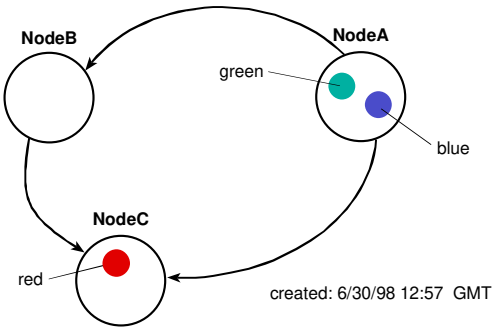

Figure 8-3 Example Net as a model of the SimpleGraph metamodel

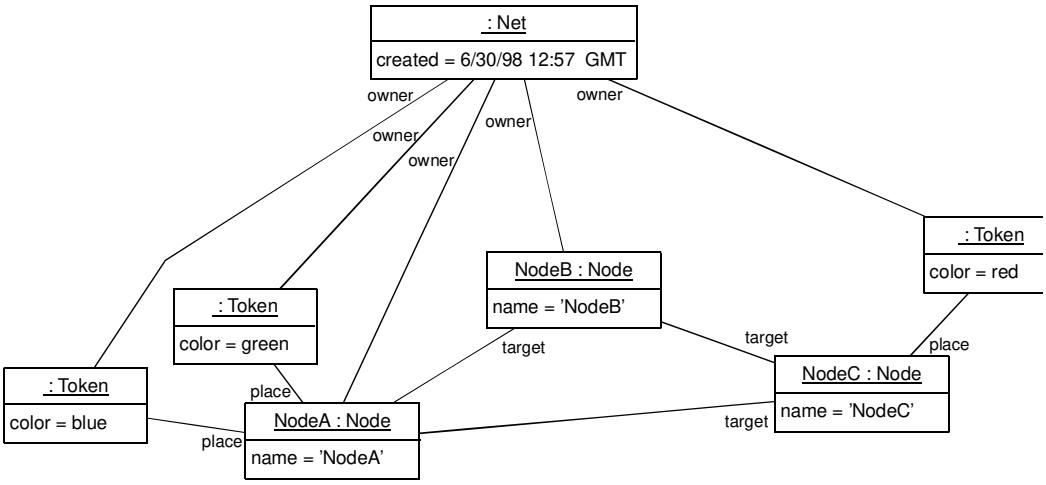

Figure 8-4 Objects forming the example SimpleGraph model

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 Chapter 9, *XML Document Production on page 167*. Section 6.5, *Necessary XMI DTD Declarations* on page 53 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:

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

The element attribute xmi.id provides a unique identifier with the document for this element.

Note that all names in XMI are fully qualified, based on the MOF description of their metamodel. The name of the item is formed by the sequence of containments and compositions, starting at the outermost package of the metamodel and separated by dots.

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:

```
<SimpleGraph.Net.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:

```
<XMI.field>1873852</XMI.field>
<XMI.field>GMT</XMI.field>
```

Then the attribute is completed with the corresponding end tag:

```
</SimpleGraph.Net.created>
```

Were there other attributes of the Net object, they would follow in a similar manner. These are followed by the Net object's references.

The MOF supports the use of *References* in defining metamodels. A reference provides the object's navigability to linked objects. Following the attributes of an object, each of its references are written. XMI considers references to be of two different types and treats them differently.

An object linked to another via a link defined in the metamodel as having an aggregation other than composite is considered to be a *normal* reference. On the other hand, if an object is linked to another object via a link defined in the metamodel as a composite association, with the composite end corresponding to link end of the composite object, then the reference used is a *composite* reference.

In XMI, all of the normal references of an object are written, followed by all of the composite references. In XMI, this composition is indicated by XML element containment.

In this example, there is a total of three Node objects and three Token objects contained by the Net object using composite references. The "nodes" reference will be expressed as:

```
<SimpleGraph.Net.nodes>
```

to indicate the Node objects it contains through the "nodes" reference. 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:

```
<SimpleGraph.Node xmi.id='a2'>
```

the value of the attribute id of the XML element 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:

```
<SimpleGraph.Node.name>NodeA</SimpleGraph.Node.name>
```

Next the normal, i.e. 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:

```
<SimpleGraph.Node.marker>
  <SimpleGraph.Token xmi.idref='a5' />
  <SimpleGraph.Token xmi.idref='a6' />
</SimpleGraph.Node.marker>
```

Since this is a normal, rather than a composite, reference, the Token objects are not written at this point. Rather, a reference is used to point within the document to the elements that actually define the objects. A complete set of linking attributes is defined in XMI; the xmi.idref could, for example, be replaced by an href to element definitions in another location. See Section 6.5.1, *Necessary XMI Attributes* on page 47, for a discussion on linking attributes.

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

```
<SimpleGraph.Node.targetNodes>
  <SimpleGraph.Node xmi.idref='a3' />
</SimpleGraph.Node.targetNodes>
<SimpleGraph.Node.targetNodes>
  <SimpleGraph.Node xmi.idref='a4' />
</SimpleGraph.Node.targetNodes>
```

This example illustrates the fact that, for references with multiplicities with upper bounds which may be greater than one, it is not necessary to place all of the references under a single tag. Although this clearly wasteful of space in the XML document, it is allowed.

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:

```
</SimpleGraph.Node>
```

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

```
<SimpleGraph.Node xmi.id='a3'>
  <SimpleGraph.Node.name>NodeB</SimpleGraph.Node.name>
  <SimpleGraph.Node.targetNodes>
```

```
      <SimpleGraph.Node xmi.id='a3' />
    </SimpleGraph.Node.targetNodes>
  </SimpleGraph.Node>
  <SimpleGraph.Node xmi.id='a4'>
    <SimpleGraph.Node.name>NodeC</SimpleGraph.Node.name>
    <SimpleGraph.Node.marker>
      <SimpleGraph.Node xmi.idref='a7' />
    </SimpleGraph.Node.marker>
  </SimpleGraph.Node>
```

Notice that for NodeB, the "marker" reference element is omitted. When the lower bound of the multiplicity of an Attribute or a Reference is zero, and no value is present, the element tag may be omitted. In a similar fashion, 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:

```
</SimpleGraph.Net.nodes>
```

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

```
<SimpleGraph.Net.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.

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

Since the value of the attribute is encoded in the tag of the empty element, a separate end tag is not used. The Token class is defined with the single attribute. If the class were derived from a supertype, the values of attributes and references defined in the supertype would also be written out as XML, preceding the attributes of the class itself. Unlike 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 "net" object in the composite link. These references need not be written, since the XML element containment indicates the composition. They are useful when the contained object is reached via a link attribute.

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

```
<SimpleGraph.Token xmi.id='a6'>
  <SimpleGraph.Token.color xmi.value='blue' />
</SimpleGraph.Token>
<SimpleGraph.Token xmi.id='a7'>
  <SimpleGraph.Token.color xmi.value='red' />
</SimpleGraph.Token>
</SimpleGraph.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:

```
</SimpleGraph.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.

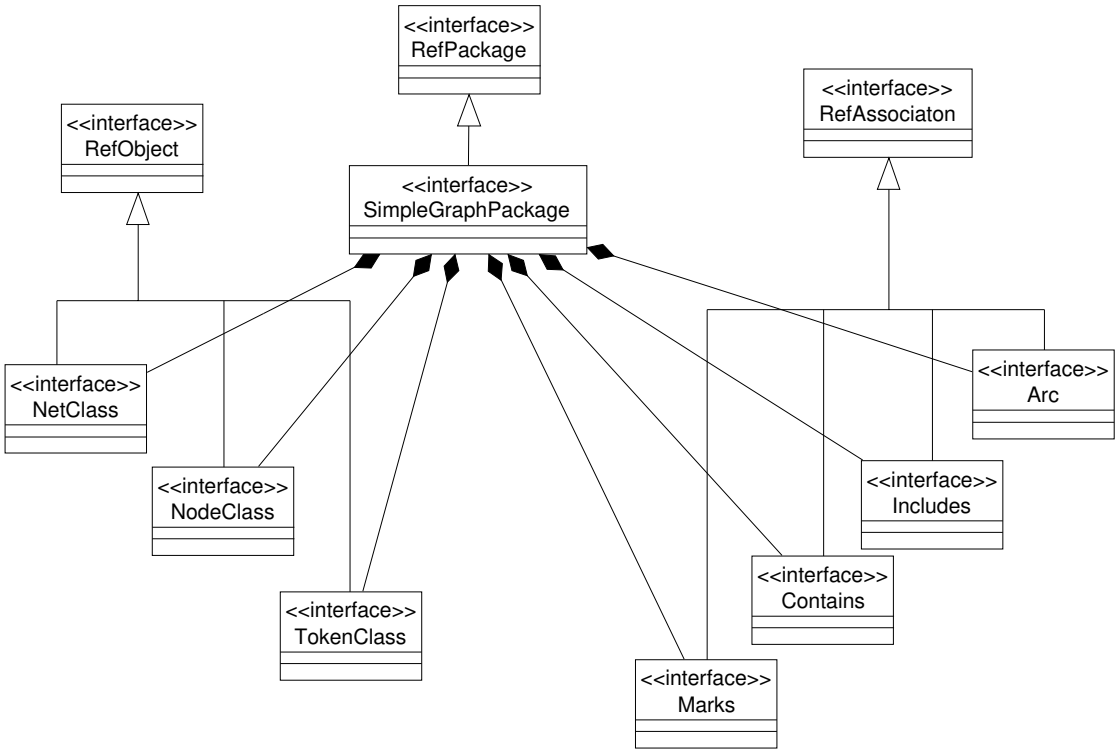

Figure 8-5 Generated interfaces from the SimpleGraph metamodel

In this example, the IDL generation creates interfaces SimpleGraphPackage, NetClass, NodeClass, TokenClass, and Arc. Figure 8-5 on page 161 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 162. These nets are shown in Figure 8-7 on page 163, 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

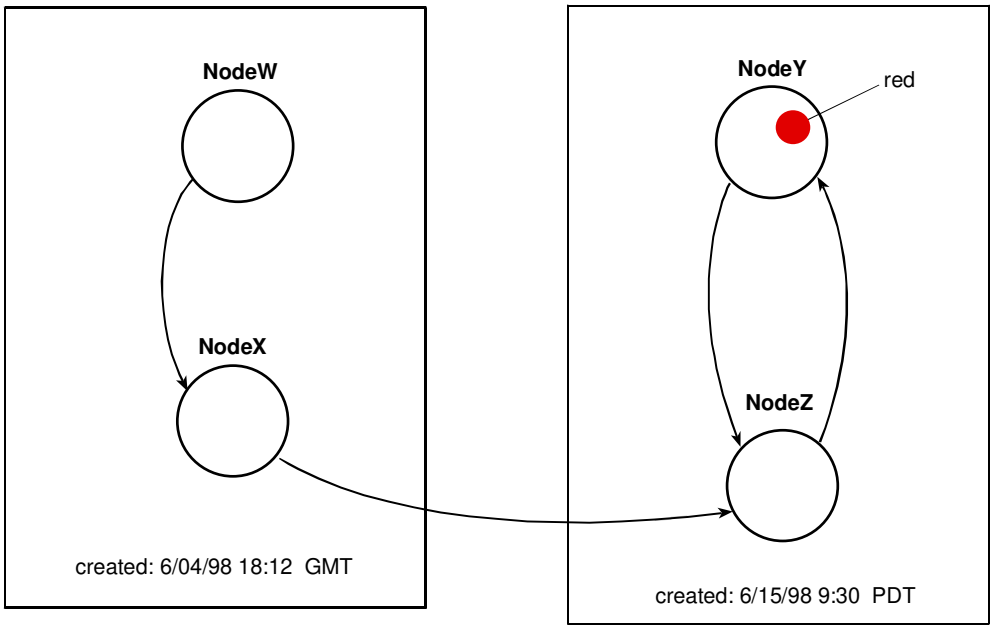

Figure 8-6 Example of two nets with a connecting arc

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

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

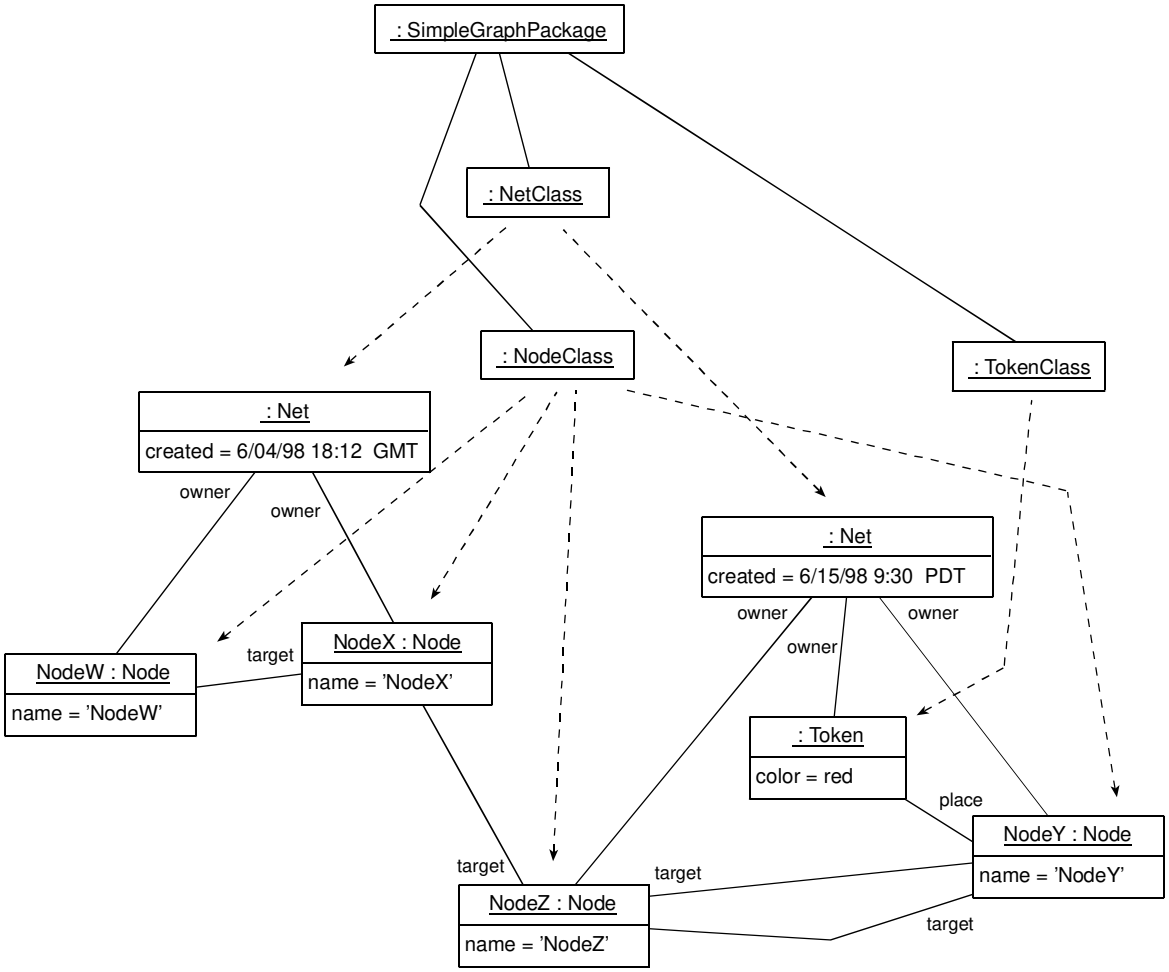

Figure 8-7 Objects representing multiple Nets and instances of RefPackage and RefObject subtypes

```
            <SimpleGraph.Node xmi.id='a6' />
          </SimpleGraph.Node.targetNodes>
        </SimpleGraph.Node>
      </SimpleGraph.nodes>
    </SimpleGraph.Net>
```

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

```
<SimpleGraph.Net xmi.id='a4'>
  <SimpleGraph.Net.created>
    <XMI.field>1872537</XMI.field>
    <XMI.field>GMT</XMI.field>
  </SimpleGraph.Node.created>
  <SimpleGraph.Net.nodes>
    <SimpleGraph.Node xmi.id='a5'>
```

```
      <SimpleGraph.Node.name>NodeY</SimpleGraph.Node.name>
      <SimpleGraph.Node.targetNodes>
        <SimpleGraph.Node xmi.idref='a6' />
      </SimpleGraph.Node.targetNodes>
    </SimpleGraph.Node>
    <SimpleGraph.Node xmi.id='a6'>
      <SimpleGraph.Node.name>NodeW</SimpleGraph.Node.name>
      <SimpleGraph.Node.targetNodes>
        <SimpleGraph.Node xmi.idref='a5' />
      </SimpleGraph.Node.targetNodes>
      <SimpleGraph.Node.marker>
        <SimpleGraph.Token xmi.idref='a7' />
      </SimpleGraph.Node.marker>
    </SimplGraph.Node>
  </SimpleGraph.Node.nodes>
  <SimpleGraph.Node.tokens>
    <SimpleGraph.Token xmi.id='a7'>
      <SimpleGraph.Token.color xmi.value='red' />
    </SimpleGraph.Token>
  </SimpleGraph.Node.tokens>
</SimpleGraph.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
- classifier-level attributes need to be replicated.

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 href 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 in the contents of an Arc element, which itself 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 where 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 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 (the class proxies) contained by the RefPackage. For Production by object containment, the values of a classifier-level attribute are not included. Conversely, in Production by Package Extent, all classifier-level attributes are included in the XML document. 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.

8.4.4 Standard Elements

Model data placed in an XML document using the rules of XMI are contained in standard XML elements defined by XMI. XMI document is encapsulated in a set of standard XMI elements. These elements are described in Section 6.2.2, *Requirements for XMI DTDs* on page 50.

XML Document Production 9

9.1 Purpose

This section specifies the production of an XML document from a model. It is essential for successful model interchange that this specification be complete and unambiguous. It is also essential that all significant aspects of the metadata are included in the XML document and can be recovered from it.

9.2 Introduction

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

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

9.3 Rules Representation

The XML produced by XMI is represented here in Extended Backus Naur 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.

9.3.1 EBNF Productions

Within the EBNF productions, the various expression elements are distinguished in font and face in the following manner:

EBNF Expression Elements

Element	Font and Face Example
expression symbols	**(* +)**
terminal	**<XMI.content>**
value to be filled in	***class-name***
symbol defined in another expression	**AttributeValue**

when this document is viewed in an electronic form, symbols defined in other expressions may provide hyperlinks to their corresponding defining expression.

9.3.2 OCL Rules

The OCL expressions make use of both operations defined in MOF Model elements, and operations defined for the MOF Reflective Interfaces. Because these operations are well defined in the MOF specification, their use does not diminish 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:

```
Producer-Attribute ← 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.

Attributes Defined for the Producer Class

Attribute	Description	Set	Used
root : RefObject	the supplied root of the model (*for production by object containment only*)	at the start of generation	In finding links not included as reference values
byContainment : boolean	specifies whether production is by containment or extent	at the start of generation	in determining whether an object is in the scope of the objects being exported
objectInventory : Sequence (RefObject)	used with objectIds to provide a dictionary; objects which have an id assigned	as each object is given an id during XML production	when an id is needed for an object which has one assigned
objectIds : Sequence(integer)	the ids associated with the objects in objectInventory	as each object is given an id during XML production	when an id is needed for an object which has one assigned
refPkg : RefPackage	the supplied RefPackage instance (*for production by package extent only*)	at the start of generation	In finding links not included as reference values
metamodel : Package	the metamodel of the input model, as an instance of Package	at the start of generation	in finding the DataType defining the type of a value provided as an Any
typeDefinitions : Sequence(string)	any type definitions required for datatypes which are not in the metamodel	whenever a data value is encountered whose type is not in the metamodel	as the last part of generating the XMI.content
tcInventory : Sequence (TypeCode)	used with tcIds to provide a dictionary; TypeCodes which are currently represented in typeDefinitions	Whenever an element is added to typeDefinitions	When an XMI.any element needs to refer to a type definition already in typeDefinitions
tcIds : Sequence(integer)	the ids associated with the TypeCodes in tcInventory	Whenever an element is added to typeDefinitions	When an XMI.any element needs to refer to a type definition already in typeDefinitions
constructedTcList : Sequence (TypeCode)	the struct TypeCodes encountered in generating a TypeCode; used to support TypeCodes for recursive sequences	initialized at the start of a top-level TypeCode; set when a struct TypeCode is encountered	a TypeCode with a recursive sequence is encountered

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 (e.g., the string `'<State><name>on</name></State>'` is equivalent to `Sequence{ '<State>','<name>','on','</name>','</State>'}`).

9.4 Production Rules

As described in Section 8.3, *Two Model Sources* on page 153, an XML document can be produced by two methods: by object containment, based upon a root object of a containment hierarchy; and by package extent, based upon a package proxy. So the XMI.content element of a document production is represented by the alternative of two rules, as shown by the following EBNF expression:

XMIContent ::= **(ContentsFromRoot | ContentsFromExtent)**

These two productions are composed of other productions. Most of the productions are shared by both of these schemes.

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 ContentsFromRoot

The ContentsFromRoot production generates the XMI.content element and its contents from the root object of a model. The contents of the element is provided by the root object, along with all the objects contained in the model, possibly some links, and possibly some type definitions. The rules do not require that the supplied object be a root (uncontained) object.

ContentsFromRoot ::= **<XMI.content>**
ObjectAsElement OtherLinks? RequiredTypeDefinitions?
</XMI.content>

In the OCL operation, the OtherLinks and RequiredTypeDefinitions operations are always evaluated. However, as shown in the definition of those operations, an empty sequence may be returned.

```
ContentsFromRoot(root : RefObject) : Sequence(string)
ContentsFromRoot(root) =
  Sequence{ '<XMI.content>',
            ObjectAsElement(root),
            OtherLinks(root),
            RequiredTypeDefinitions(),
            '</XMI.content>'
          }
```

9.4.1.2 OtherLinks

Object references provide a representation of links, when a Reference is defined in the metamodel for that object. However, for Associations where neither end has a corresponding Reference, the links will not be represented via references. This production adds the unrepresented links, when the objects at both link ends are in the model.

OtherLinks ::= < *association-name* >
(ReferencingElement ReferencingElement)*
</ *association-name* >

This operation gets all the Associations in the metamodel which have no corresponding Reference. Among those operations, the Associations are selected which have one or more links in the model (as identified by the root). Over that sequence of Associations, a sequence of strings is produced. For each Association, the element start tag is produced with the association name. Then, for each link of the Association which is in the model, a sequence of strings is produced. For each link, two Reference elements are produced, representing the two link ends. Finally, for each Association, the corresponding element end tag is produced.

```
OtherLinks(root : RefObject) : sequence(String)
OtherLinks(root) =
  UnreferencedAssoc(this.metaModel)->select(a |
      LinksInRoot(a)->notEmpty)->collect(a |
          Sequence{ '<',
                    DotNotation(a.qualifiedName),
                    '>',
                    LinksInRoot(a)->collect(lnk |
                       Sequence{ '<',
                                 DotNotation(End(1, a).qualifiedName)
                                 '>
                                 ReferencingElement(lnk->at(1)),
                                 '</',
                                 DotNotation(End(1, a).qualifiedName)
                                 '>
                                 '<',
                                 DotNotation(End(2, a).qualifiedName)
                                 '>
                                 ReferencingElement(lnk->at(2)),
                                 '</',
                                 DotNotation(End(2, a).qualifiedName)
                                 '>
```

```
                    } ),
                '</',
                DotNotation(a.qualifiedName),
                '>'
        } )
```

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 160).

9.4.2.1 ContentsFromExtent

The contents of the XML document are enclosed in an XMI.content element. The contents of that element will contain class attributes, if any are present in the extent. Then, for each uncontained object in the extent, an element is produced, whose contents will represent the object and all its contained objects. Additionally, other links of the extent may be represented, as well as type definitions.

ContentsFromExtent ::= **<XMI.content>**
ClassAttributes* ObjectAsElement*
OtherExtentLinks? RequiredTypeDefinitions?
</XMI.content>

This OCL operation accepts a RefPackage, a package proxy corresponding to an uncontained Package instance from a MOF-defined metamodel The operation first produces the XMI.content element start tag. Then, for each class proxy (instance of a RefObject subtype corresponding to a Class instance in the metamodel) any classifier-level attributes are represented. For each uncontained object in the extent, the object and all its contained objects are represented. Additionally, any links in the extent not represented as reference values are then represented. Note that the two operations, ClassAttributes and OtherExtentLinks, may return empty sequences.

```
ContentsFromExtent(pkgProxy : RefPackage) : Sequence(string)
ContentsFromExtent(pkgProxy) =
  Sequence{ '<XMI.content>',
            AllClassProxies(pkgProxy)->collect(c | ClassAttributes(c)),
            AllUncontainedObjects(pkgProxy)->collect(obj | ObjectAsElement(obj)),
            OtherExtentLinks(pkgProxy),
            RequiredTypeDefinitions(),
            '</XMI.content>'
          }
```

9.4.2.2 ClassAttributes

The class attributes of a class proxy results in zero or more elements representing the attributes.

ClassAttributes ::= **AttributeAsElement***

For a class proxy, all the classifier-level attributes of the corresponding Class instance in the metamodel are determined. For each of those, the AttributeAsElement operation returns a sequence of strings representing the attribute.

```
ClassAttributes(classProxy : RefObject) : Sequence(string)
  ClassAttributes(classProxy) =
    classProxy.metaObject().oclAsType(Class).findElementsByTypeExtended
        (MofAttribute, false)->select(attr |
            attr.scope = classifier_level)->
                 collect(cAttr | AttributeAsElement(iAttr, obj.value(cAttr)))
```

9.4.2.3 OtherExtentLinks

This production adds the links which are not represented by reference values. In this case, the links are represented regardless of whether or not the link end objects are in the extent.

OtherExtentLinks ::= **< *association-name* >**
(ReferencingElement ReferencingElement)*
</ *association-name* >

This operation gets all the Associations in the metamodel which have no corresponding Reference. Over that sequence of associations, a sequence of strings is produced. For each association, the element start tag is produced with the association name. Then, for each link of the association, a sequence of strings is produced. For each link, two reference elements are produced, representing the two link ends. Finally, for each Association, the corresponding element end tag is produced

```
OtherExtentLinks(pkgProxy : RefPackage) : sequence(String)
OtherExtentLinks(pkgProxy) =
  UnreferencedAssoc(pkgProxy.metaObject().oclAsType(Package))->collect(a |
      Sequence{ '<',
                DotNotation(a.qualifiedName),
                '>',
                a.allLinks()->collect(l |
                   Sequence{ ReferencingElement(End(1, a), l->at(1)),
                             ReferencingElement(End(2, a), l->at(2))
                   } )
                '</',
                DotNotation(a.qualifiedName),
                '>'
      } )
```

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 (and any objects it contains) as the contents of the element, followed by the element end tag.

ObjectAsElement ::= **< *class-name* xmi.id=" IdOfObject ">**
ObjectContents </ *class-name* >

The operation produces the name of the element from the Class instance in the metamodel defining the class of this object. The fully-qualified name, using a dot notation, is used to avoid any ambiguity in naming the Class. The element start tag also includes an identifier of the object, which is required to be a locally-unique identifier. The ObjectContents operation provides the contents of the element.

```
ObjectAsElement(obj : RefObject) : Sequence(string)
ObjectAsElement(obj) =
  Sequence{ '<',
            DotNotation(obj.metaObject().oclAsType(Class).qualifiedName),
            ' xmi.id="',
            IdOfObject(obj),
            '">',
            ObjectContents(obj.metaObject().oclAsType(Class), obj),
            '</',
            DotNotation(obj.metaObject().oclAsType(Class).qualifiedName),
            '>'
          }
```

9.4.3.2 ObjectContents

The ObjectContents operation produces XML to represent the contents of an object – its state (attributes and references). The object is represented by its attributes, its non-composing references, and its components. The attribute values of the object are included in the contents of this element. The references corresponding to Associations defined as composite are treated separately from other references. Those which are not composite are represented in the contents of this element using XML's XLink mechanism. Those reference values which correspond to contained elements of the composition are represented wholly in the contents of this element.

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

The ObjectContents operation produces XML to represent the contents of an object – its state (attributes and references). Three steps are required to produce XML from the input object: produce the XML for the object's non-derived, instance-level attribute values, then produce the XML for the object's non-derived, non-composite, non-component reference values, and finally produce the XML for the objects' component objects. From the object's class, all the Attributes are obtained, including inherited attributes. Among those, the non-derived, instance-level attributes are selected. Over that sequence, string representations of the values are produced, using the AttributeAsElement operation. The value operation, from the MOF's RefObject interface, provides the attribute value or values.

Then, among the classes' References, those which match the following criteria are selected: not based upon a derived Association, not with a referencedEnd which is a composite, and not with an exposedEnd (the other association end) which is a composite. From those References, the ReferenceAsElement is used to produce the XML representing the value or values of each reference. References of a component object which refer to its composite object (reference's referencedEnd with composite aggregation) may be optionally included. The generated DTD supports the optional inclusion of this reference, but it is not shown here.

The non-derived References corresponding to contained elements are then obtained. These References have their exposedEnd's AssociationEnd with composite aggregation. Over these references, the CompositeAsReference operation is used on the reference value or values of each reference to produce the XML.

```
ObjectContents(metaClass : MofClass, obj : RefObject) : Sequence(string)
ObjectContents(metaClass, obj) =
  Sequence{
     metaClass.findElementsByTypeExtended(MofAttribute, false)->select(attr |
        attr.scope = instance_level and not attr.isDerived)->collect(iAttr |
           AttributeAsElement(iAttr, obj.value(iAttr))),

     metaClass.findElementsByTypeExtended(Reference, false)->select(ref |
        ref.exposedEnd.aggregation <> composite and
        ref.referencedEnd.aggregation <> composite and not
        ref.referencedEnd.container.oclAsType(Association).isDerived)->
           collect(r | ReferenceAsElement(r, obj.value(r))),

     metaClass.findElementsByTypeExtended(Reference, false)->select(ref |
        ref.exposedEnd.aggregation = composite and not
        ref.referencedEnd.container.oclAsType(Association).isDerived)->
           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 without identifiers in the element start tag. Identifiers are not required when an object is not participating in any link.

EmbeddedObject ::= **< *class-name* > ObjectAsElement </ *class-name* >**

Because the interactions with the MOF are defined using the MOF's reflective interfaces, the values of object attributes, references and link ends are represented using the CORBA Any type, matching the return type of those interfaces' operations. The extract_Object operation is an operation defined for the CORBA Any type. It is used to convert the Any value to an object. The ObjectContents operation is used to define the contents of this element.

```
EmbeddedObject(metaClass : MofClass, value : Any) : Sequence(string)
EmbeddedObject(metaClass, value) =
  Sequence{
     '<',
     DotNotation(metaClass.qualifiedName),
     '>',
     ObjectContents(metaClass,
                    value.extract_Object().oclAsType(RefObject)),
     '</',
     DotNotation(metaClass.qualifiedName),
     '>'
  }
```

9.4.4 Attribute Production

9.4.4.1 AttributeAsElement

Each object attribute value is represented in XML in enclosing start and end tags which identify the attribute. If the attribute is multi-valued (holding a more than one instance of an object or datatype) then all those values may be represented within the contents of the single element representing the attribute. Because of the differences between single- and multi-valued references, they are handled in separate operations.

AttributeAsElement ::= **(SvAttributeContents | MvAttributeContents)**

If the Attribute's multiplicity has an upper bound greater than one, including unbounded, or a lower bound of zero, the attribute value or values are returned in the MOF as a possibly-empty collection.

```
AttributeAsElement(attr : MofAttribute, value : Any) : Sequence(string)
AttributeAsElement(attr, value) =
  if attr.multiplicity.lower = 0 or
     attr.multiplicity.upper > 1 or
     attr.multiplicity.upper = unbound then
    MvAttributeContents(attr, value)
  else
    SvAttributeContents(attr, value)
  endif
```

9.4.4.2 ObjectReference

This operation represents the reference as a link, using either XML's local linking ability, or XLink.

ObjectReference :: `<XMI.reference (( xmi.idref=" IdOfObject " ) | ( href=" UrlOfObject " )) />`

The IdOfObject will return a local id if the referenced object is part of the model. If not, it returns an empty string.

```
ObjectReference(value : Any) : Sequence(string)
ObjectReference(value) =
  Sequence{ '<XMI.reference',
            (if IdOfObject(value.extract_Object()) <> '' then
               Sequence{ ' xmi.idref="', IdOfObject(value.extract_Object()), '"' }
             else
                Sequence{ ' href="', UrlOfObject(value.extract_Object()), '"' }
             endif),
            '/>'
  }
```

9.4.4.3 SvAttributeContents

The contents of a single valued attribute is either an object or a data value If it is a datavalue of type boolean, or of an enum type, special production rules are used. Enum and boolean values are represented in the XML element's attribute, while all other values are represented in the Attribute element's contents.

SvAttributeContents ::= `( < attribute-name > EmbeddedObject </ attribute-name > ) | EnumAttribute | ( < attribute-name > ObjRefOrDataValue </ attribute-name >)`

Values which are objects are handled by providing an enclosing pair of elements, named by the attribute's qualified name, and the EmbeddedObject operation. For data values, if the value is boolean or an enum type, the EnumAttribute operation provides the empty element holding the value. Otherwise, the ObjRefOrDataValue operation provides the element contents for the attribute element.

```
SvAttributeContents(attr : MofAttribute, value : Any) : Sequence(string)
SvAttributeContents(attr, value) =
  if attr.type().oclIsOfType(Class) then
    Sequence{ '<',
              DotNotation(attr.qualifiedName),
              '>',
              EmbeddedObject(attr, value),
              '</',
              DotNotation(attr.qualifiedName),
```

```
            '>',
    }
  else
    if DeAlias(attr.type().typeCode()).kind() = tk_boolean or
       DeAlias(attr.type().typeCode()).kind() = tk_enum then
      EnumAttribute(attr, value, DeAlias(attr.type().typeCode()).kind())
    else
      Sequence{ '<',
                DotNotation(attr.qualifiedName),
                '>',
                ObjRefOrDataValue(value, DeAlias(attr.type().typeCode()).kind()),
                '</',
                DotNotation(attr.qualifiedName),
                '>'
      }
    endif
  endif
```

9.4.4.4 *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 ::= **<** ***attribute-name*** **> (EmbeddedObject+ | EnumAttribute+ | ObjRefOrDataValue+) </** ***attribute-name*** **>**

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

```
MvAttributeContents(attr : MofAttribute, value : Any) : Sequence(string)
MvAttributeContents(attr, value) =
  Sequence{
    '<',
    DotNotation(attr.qualifiedName),
    '>' ,
    (if attr.type().oclIsOfType(Class) then
       ExtractSequence(value)->collect( obj | EmbeddedObject(attr, obj))
     else
       if DeAlias(attr.type().typeCode()).kind() = tk_boolean or
          DeAlias(attr.type().typeCode()).kind() = tk_enum then
         ExtractSequence(value)->collect(item |
           EnumAttribute(attr, item,
                         DeAlias(attr.type().typeCode()).kind()))
       else
         ExtractSequence(value)->collect(item |
           Sequence{ '<XMI.seqItem>',
                ObjRefOrDataValue(item, DeAlias(attr.type().typeCode()).kind()),
                     '</XMI.seqItem>'
           }
       endif
     endif),
    '</',
```

```
    DotNotation(attr.qualifiedName),
    '>'
  }
```

9.4.4.5 EnumAttribute

An attribute of an enum of boolean type is treated special. The value is represented in an element attribute, rather than the element contents.

EnumAttribute ::= **< *attribute-name* `xmi.value="`**
(BooleanAsString | EnumAsString) `"/>`

```
EnumAttribute(attr, MofAttribute, value : Any, kind : TCKind) :
    Sequence(string)
EnumAttribute(attr, value, kind) =
  Sequence{ '<',
            DotNotation(attr.qualifiedName),
            'xmi.value="',
            (if kind = tk_boolean then
               BooleanAsString(value)
             else
               EnumAsString(value)
             endif)
            '"/>'
  }
```

9.4.5 Reference Productions

In the MOF, the object-to-object navigability via links is supported through the definition of References in Classes. This operation defines the representation of reference values when they are not component values of a composition.

9.4.5.1 ReferenceAsElement

The ReferenceAsElement operation provides the representation one or more reference values.

ReferenceAsElement ::= **(< *reference-name* > ReferencingElement***
</ *reference-name* >) |
( )

When the Reference's multiplicity has an upper bound greater than one, including unbounded, or a lower bound of zero, the reference is checked to see if any values exist. If not, no elements are returned.

```
ReferenceAsElement(ref : Reference, value : Any)
ReferenceAsElement(ref, value) =
  if ref.multiplicity.lower = 0 or
     ref.multiplicity.upper > 1 or
     ref.multiplicity.upper = unbound then
    if ExtractSequence(value)->notEmpty then
      Sequence{ '<',
                DotNotation(ref.qualifiedName),
                '>',
                ExtractSequence(value)->collect(item |
                   ReferencingElement(ref,
                                      item.extract_Object().oclAsType(RefObject))),
                '<',
                DotNotation(ref.qualifiedName),
                '/>'
      }
    else
      Sequence {}
  else
    Sequence{ '<',
              DotNotation(ref.qualifiedName),
              '>',
             ReferencingElement(ref, value.extract_Object().oclAsType(RefObject))
              '<',
              DotNotation(ref.qualifiedName),
              '/>'
    }
  endif
```

9.4.5.2 *ReferencingElement*

This operation represents an association end value (a link end) or reference value as a link, using either XML's local linking ability, or XLink. The element name could represent either the reference name or the association end name, depending on its use.

ReferencingElement :: **<** ***element-type-name***
((**xmi.idref="** **IdOfObject** **") |**
(**href="** **UrlOfObject** **")) />**

The Element's name is provided from the fully-qualified name of the supplied ModelElement. The IdOfObject will return a local id if the referenced object is part of the model. If not, it returns an empty string. UrlOfObject returns an URL linking to a representation of the object outside the document.

```
ReferencingElement(obj : RefObject) : Sequence(string)
ReferencingElement(obj) =
  Sequence{ '<',
            DotNotation(obj.metaObject().oclAsType(Class).qualifiedName),
            (if IdOfObject(obj) <> '' then
                Sequence{ ' xmi.idref="', IdOfObject(obj), '"' }
             else
                Sequence{ ' href="', UrlOfObject(obj), '"' }
             endif),
```

```
            '/>'
  }
```

9.4.6 Composition Production

9.4.6.1 CompositeAsElement

A Reference to a contained element of a composition has the referenced objects included in the contents of the reference element.

CompositeAsElement ::= < *reference-name* > **ObjectAsElement*** </ *reference-name* >

The element name is supplied by the fully-qualified name of the reference. The multiplicity of the Reference determines whether the value is an object or a collection of objects.

```
CompositeAsElement(ref : Reference, value : Any)
CompositeAsElement(ref, value) =
  Sequence{ '<',
            DotNotation(ref.qualifiedName),
            '>',
            (if attr.multiplicity.lower = 0 or
                attr.multiplicity.upper > 1
                  attr.multiplicity.upper = unbound then
               ObjectAsElement(value.extract_Object().oclAsType(RefObject))
             else
               ExtractSequence(value)->collect (item |
                  ObjectAsElement(item.extract_Object().oclAsType(RefObject)))
             endif),
            '<',
            DotNotation(ref.qualifiedName),
            '/>'
          }
```

9.4.7 DataValue Productions

The MOF currently uses the CORBA type system as the base type system for data types. In a metamodel, additional data types can be defined, such as Date or BigInteger. However, values of those types will be represented in terms of CORBA type values. As a consequence, XMI needs to be able to encode values of CORBA data types. This is what the DataValue productions are designed to do.

Data values are represented in XMI with a conservative number of elements. The expected type of most data values in an XML document are defined by the metamodel and determined by the context. In particular, the expected type for the value of an attribute is defined in the metamodel by the DataType associated with the Attribute.

Thus XML elements that represent data values or their component parts do not need to identify the type.

The exception to this is instances of the CORBA primitive data type Any. The CORBA Any type is a "universal union" type; i.e. a type which can encapsulate a value of any CORBA data type or object reference in a type safe way. An instance of the Any type consists of two parts; the encapsulated value, and a descriptor for the encapsulated value's type. The latter is expressed as an instance of another CORBA primitive data type called TypeCode.

When an instance of the Any type is encoded in XMI, the expected type of the encapsulated value cannot be determined before hand. Rather, the type is expressed using element attributes of the XMI.any element. In most cases, the type information can be represented using the kind and name element attributes. In the most general case, however, an href element attribute provides a link to an XMI.CorbaTypeCode element, to provide the type description.

9.4.7.1 ObjRefOrDataValue

Based upon the supplied type representation, the appropriate operation is used to represent the value. This production can encounter an object reference when the value is an item within a datatype value (a struct, union, sequence, or array) or a CORBA Any-typed value (which, strictly speaking, is also a data value) In these cases, objects are treated as object references (i.e. using XLink to point to the object) rather than representing the object directly in the element contents. Likewise, an enum-typed value will only be encountered in this production as an element of a datatype, or as an Any-typed value.

ObjRefOrDataValue ::= **(StructValue | SequenceValue | UnionValue | StringValue | CharacterValue | EnumAsElement | IntegralValue | RealValue | TypeCodeValue | AnyValue | ObjectReference)**

```
ObjRefOrDataValue(value : Any, kind : TcKind) : Sequence(string)
ObjRefOrDataValue(value, kind) =
  if kind = tk_struct then
    StructValue(value)
  else
    if kind = tk_sequence or kind = tk_array then
      SequenceValue(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 or kind = tk_boolean then
              EnumAsElement(value)
```

```
            else
              if Set{ tk_short, tk_ushort, tk_long,
                      tk_ulong, tk_longlong,
                      tk_ulonglong, tk_octet }->includes(kind) then
                IntegralValue(value)
              else
               if kind = tk_float or kind = tk_double or kind = tk_longdouble then
                  RealValue(value)
                else
                  if kind = tk_TypeCode then
                    TypeCodeValue(value)
                  else
                    if kind = tk_any then
                      AnyValue(value)
                    else
                      if kind = tk_objref then
                        ObjectReference(value)
                      else
                        -- should never be the case
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  endif
```

9.4.7.2 StructValue

A value of a struct type is represented as a sequence of its fields, each enclosed in a field element.

StructValue ::= **(<XMI.field> ObjRefOrDataValue </XMI.field>)+**

The operation defines a integer sequence over the range of struct fields. Over this sequence, each field is represented. For each index value, the field element is produced, enclosing the value of the field. The member_type operation, part of the TypeCode object, returns the type of the field at the index. The DeAlias operation, applied to this typecode, removes any aliases.

```
StructValue(value : Any) : Sequence(string)
StructValue(value) =
  Sequence{ 0..value.type().member_count() - 1 }->collect(index |
     Sequence{ '<XMI.field>',
               ObjRefOrDataValue(ExtractField(value, index),
                                 DeAlias(value.type().member_type(index)).kind()),
               '</XMI.field>'
             })
```

9.4.7.3 SequenceValue

A value of type sequence is represented by having each of its items contained in a sequence item element. However, a sequence of octets is treated special, allowing a blob-like value to be represented more efficiently. A sequence of octet (eight-bit) values is represented as a string encoded in hexadecimal. A value of type Array is treated identically a sequence in this production rule.

SequenceValue ::= **(<XMI.octetStream> HexAsString***
</XMI.octetStream>) |
(<XMI.seqItem> ObjRefOrDataValue </XMI.seqItem>)*

If the content type of the sequence value is octet, each item is produced as two hexadecimal characters. Otherwise, each sequence item is produced in the form of a XMI.seqItem element. The TypeCode's content_type operation returns the type of the sequence or array items.

```
SequenceValue(value : Any) : Sequence(string)
SequenceValue(value) =
  if DeAlias(value.type()).content_type()).kind() = tk_octet then
    Sequence{ '<XMI.octetStream>',
              ExtractSequence(value)->collect(seqItem |
                 HexAsString(seqItem.extract_octet())),
              '<XMI.octetStream>'
    }
  else
    ExtractSequence(value)->collect( seqItem |
        Sequence{ '<XMI.seqItem>',
                  ObjRefOrDataValue(seqItem,
                                    DeAlias(value.type()).content_type()).kind()),
                  '</XMI.seqItem>'
        } )
  endif
```

9.4.7.4 UnionValue

The union value has both its discriminator value and field value represented in XML elements.

UnionValue ::= **<XMI.unionDiscrim> ObjRefOrDataValue**
</XMI.unionDiscrim>
<XMI.field> ObjRefOrDataValue </XMI.field>

The discriminator value is provided by ExtractUnionDiscrim operation, the field value is provided by the ExtractUnionField operation, and the field type is determined by the GetUnionFieldType operation. These three OCL operations are described in this chapter.

```
UnionValue(value : Any) : Sequence(string)
UnionValue(value) =
```

```
  Sequence {
     '<XMI.unionDiscrim>',
     ObjRefOrDataValue(ExtractUnionDiscrim(value),
              DeAlias(DeAlias(value.type().discriminator_type())).kind(),
     '</XMI.unionDiscrim>',
     '<XMI.field>'
     ObjRefOrDataValue(ExtractUnionField(value),
     DeAlias(GetUnionFieldType(value)).kind()),
     '</XMI.field>'
  }
```

9.4.7.5 StringValue

A string value is represented directly, without an enclosing element. However, it must be encoded to remove any characters which can be confused for XML markup, and thereby become unparsable.

StringValue ::= **EncodedString**

The handling of string and wide string differs here only in the representation of extraction from the Any-typed value.

```
StringValue(value : Any) : Sequence(string)
StringValue(value) =
  (if kind = tk_string then
     EncodedString(value.extract_string())
   else
     EncodedString(value.extract_wstring())
   endif)
```

9.4.7.6 CharacterValue

A character may need to be converted, if it is an XML markup character.

CharacterValue ::= **EncodedCharacter**

Character and wide character values are have different operations for extraction from Any values.

```
CharacterValue(value : Any) : string
CharacterValue(value) =
  (if kind = tk_char then
     EncodedCharacter(value.extract_char())
   else
     EncodedCharacter(value.extract_wchar())
   endif)
```

9.4.7.7 EnumAsElement

In XMI, the values of enum and boolean attributes are represented directly in the element representing the attribute, as element attribute values. This approach allows the DTD to represent the range of labels, and supports XML parser verification of the values. However, when the enum value is an item in a constructed data type (struct, union, sequence, or array) or an Any-typed value, that form of representation is not possible. For those cases, the enum value is represented in the element defined here.

EnumAsElement ::= `<XMI.enum xmi.value="` *enum-value-label* `">`

The BooleanAsString or EnumAsString operation provides the string representation of the value.

```
EnumAsElement(value : Any) : Sequence(string)
EnumAsElement(value) =
  Sequence{ '<XMI.enum xmi.value="',
            (if kind = tk_boolean then
               BooleanAsString(value)
             else
               EnumAsString(value)
             endif),
            '"/>'
  }
```

9.4.7.8 IntegralValue

All integer types are handled by this operation.

IntegralValue ::= IntegralAsString

The specific type of the value determines the operation used to extract the integer value from the argument.

```
IntegralValue(value : Any) : string
IntegralValue(value) =
  IntegralAsString(
     if value.type().kind() = tk_short then
       value.extract_short()
     else
       if value.type().kind() =  tk_ushort then
         value.extract_ushort()
       else
         if value.type().kind() =  tk_long then
           value.extract_long()
         else
           if value.type().kind() =  tk_ulong then
             value.extract_ulong()
           else
             if value.type().kind() =   tk_longlong then
               value.extract_longlong()
```

```
            else
              if value.type().kind() =  tk_ulonglong then
                value.extract_ulonglong()
              else
                if value.type().kind() =  tk_octet then
                  value.extract_octet()
                else
                  -- undefined
                endif
              endif
            endif
          endif
        endif
      endif
    endif)
```

9.4.7.9 RealValue

Each of the real types are handled by this operation.

RealValue ::= **RealAsString**

The type of the value determines which extraction operation to use.

```
RealValue(value : Any) : string
RealValue(value) =
  RealAsString(if value.type().kind() = tk_float then
                 value.extract_float()
               else
                 if value.type().kind() =  tk_double then
                   value.extract_double()
                 else
                   -- undefined
                 endif
               endif)
```

9.4.7.10 AnyValue

This operation supports the transfer of values represented by the Any type. Ideally, XMI.any could represent any object or data value of any type. However, currently this element is limited to representing objects and data values whose types are defined in the metamodel, plus other objects and datatypes when the element links to a type description of the value's type. This when linking to a type description, this element is currently limited to linking to elements representing CORBA TypeCodes.

AnyValue ::= **<XMI.any (TypeId | TypeRef) > ObjRefOrDataValue**
</XMI.any>

XMI.any's element attributes are determined, based on the type of the argument. The TypeId operation will either provide the required element attributes, or will return an

empty string. When that operation does return an empty string, the TypeRef operation is used to provide an element attribute representing a link to a type description.

```
AnyValue(value : Any) : Sequence(string)
AnyValue(value) =
  Sequence{ '<XMI.any',
            (if TypeId(value.type()) <>  ' ' then
               TypeId(value.type())
             else
               TypeRef(value.type())
             endif),
            '>',
            ObjRefOrDataValue(value, DeAlias(value.type()).kind()),
            </XMI.any>'
  }
```

9.4.7.11 RequiredTypeDefinitions

The RequiredTypeDefinitions production generates the type representation for any value in the document whose type is not defined by the metamodel. The types are defined by TypeCode values, represented with the TypeCodeState production (which produces XMI.CorbaTypeCode elements).

RequiredTypeDefinitions ::= `<XMI.TypeDefinitions>` **TypeCodeState+** `</XMI.TypeDefinitions>`

The Producer object includes an attribute, typeDefinitions, which accumulates any required type definitions during the generation of values in a model. If this attribute is not empty, then the TypeDefinitions element is included in the XML document, with the sequence of strings held by the Producer attribute as the contents of this element. That sequence of strings will constitute one or more TypeCodeState elements.

```
RequiredTypeDefinitions(objs : Set(RefObject)) : Set(TypeCode)
RequiredTypeDefinitions() =
  if self.typeDefinitions->notEmpty then
    Sequence{ '<XMI.TypeDefinitions>',
              self.typeDefinitions,
              '</XMI.TypeDefinitions>
    }
  else
    Sequence{ }
  endif
```

9.4.7.12 TypeId

This production generates XML element attributes to represent the type of an Any value, when the type is either a primitive type or a type defined in the metamodel. For any other type, this production returns an empty string.

XMI.any attributes that encode the type description of the value inside an Any type. There are three encodings of the type information:

- If the type is a CORBA primitive type or an unbounded string or wstring, it is encoded as the type's kind. (Note that this case does not include aliases of primitive types created via a typedef.)
- If the type corresponds to a DataType in the metamodel, it is encoded as the type's kind followed by the qualified name of the DataType. When the type is an alias, the kind represents the original, or dealiased type.
- Otherwise, an empty string is returned. In this case, another production must be employed to identify the type.

TypeId ::= `( xmi.kind=" type-kind " ) |`
`( xmi.kind=" type-kind " xmi.name=" type-name " ) |`
`( )`

```
TypeId(tc : TypeCode) : Sequence(string)
TypeId(tc) =
  if  PrimitiveOrUnboundString(tc) then
    Sequence { ' xmi.kind = "', CorbaTypeName(tc.kind()), '"' }
  else
    if CorrespondingDataType(tc) <> '' then
      Sequence { ' xmi.kind = "',
                 CorbaTypeName(tc.kind()),
                 '"',
                 ' xmi.name = "',
                 CorrespondingDataType(tc),
                 '"'
       }
    else
      Sequence { '' }
    endif
  endif
```

9.4.8 CORBA-Specific Types

Although all the data types are CORBA data types, most are general in nature, and have analogous types in other type systems. The following operations support types and features which are more CORBA-specific, including the CORBA TypeCode type, and the representation of the type of a value in an XMI.any element using a TypeCode.

9.4.8.1 TypeRef

This production generates an element attribute for the XMI.any element which references a separate type definition element. This attribute is used when the value represented by the XMI.any attribute is of a type not defined in the metamodel.

TypeRef ::= `href="` | **IdOfTypeDefinition** `"`

The element attribute uses an XPointer representation of the link to the type definition corresponding to the typecode.

```
TypeRef(tc : TypeCode) : Sequence(string)
TypeRef(tc) =
 Sequence { ' href="|', IdOfTypeDefinition(tc), '"' }
```

9.4.8.2 TypeCodeValue

This production creates a representation of a TypeCode value.

TypeCodeValue ::= TypeCodeState

The id argument in the operation provides an identifier for the typecode element produced in the TypeCodeState operation. an empty string may be provided when an element identifier is not needed.

```
TypeCodeValue(value : Any, id : string) : Sequence(string)
TypeCodeValue(value) =
  TypeCodeState(value.extract_TypeCode(), id)
```

Producer object Modifications:

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

The Producer's constructedTcList attribute supports the representation of recursive sequence types. This attribute is initialized here; each struct embedded in this typecode is added to the sequence.

9.4.8.3 TypeCodeState

This production generates an element representing the state of a TypeCode value. Because a TypeCode may include other TypeCode values, this production may be employed recursively.

TypeCodeState ::= **<XMI.CorbaTypeCode>**
(TcAlias | TcStruct | TcSequence | TcArray | TcObjRef | TcEnum | TcUnion | TcExcept | TcString | TcWstring | TcFixed | TcSimple)
</XMI.CorbaTypeCode>

An id element attribute is only generated if the supplied id is not an empty string. The appropriate operation is used, based on the type that the provided TypeCode is representing.

```
TypeCodeState(tc : TypeCode, id : string) : Sequence(string)
TypeCodeState(tc) =
  Sequence{ '<XMI.CorbaTypeCode',
            (if id <> '' then
```

```
      Sequence { ' xmi.id="', id, '"' }
    else
      Sequence { }
    endif),
  '>',
  (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_array then
           TcArray(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
                     if tc.kind() = tk_wstring then
                       TcWstring(tc)
                     else
                        if tc.kind() = tk_fixed then
                          TcFixed(tc)
                        else
                          TcSimple(tc)
                        endif
                     endif
                   endif
                 endif
               endif
             endif
           endif
         endif
       endif
     endif
   endif),
  '</XMI.CorbaTypeCode>'
}
```

9.4.8.4 TcAlias

This production generates a representation of an alias type.

TcAlias ::= `<XMI.CorbaTcAlias xmi.tcName="` *alias-name* `" xmi.tcId="` *alias-id* `">` TypeCodeState `</XMI.CorbaTcAlias>`

The operation uses the CORBA-defined operations on TypeCode to extract the state of the alias typecode. The TypeCode operation is used to represent the type that is being aliased.

```
TcAlias(tc : TypeCode) : Sequence(string)
TcAlias(tc) =
  Sequence{ '<XMI.CorbaTcAlias xmi.tcName="',
            tc.name(),
            '" xmi.tcId =",
            tc.id(),
            '">',
            TypeCodeState(tc.content_type(), ''),
            '</XMI.CorbaTcAlias>'
  }
```

9.4.8.5 TcStruct

This production generates a representation of a struct type.

TcStruct ::= `<XMI.CorbaTcStruct xmi.tcName="` *struct-name* `" xmi.tcId="` *struct-id* `">`
`(` `<XMI.CorbaTcField xmi.tcName="` *field-name* `">` TypeCodeState `</XMI.CorbaTcField>` `)*`
`</XMI.CorbaTcStruct>`

The operation iterates through the a zero-based sequence of integers, with a size equal to the number of fields in the struct. At each iteration, the field definition is extracted.

```
TcStruct(tc : TypeCode) : Sequence(string)
TcStruct(tc) =
  Sequence{ '<XMI.CorbaTcStruct xmi.tcName="',
            tc.name(),
            '" xmi.tcId =",
            tc.id(),
            '">',
            Sequence{ 0..(tc.member_count() - 1) }->collect(i |
               Sequence{ '<XMI.CorbaTcField xmi.tcName="',
                         tc.member_name(i),
                         '">',
                         TypeCodeState(tc.member_type(i), ''),
                         '</XMI.CorbaTcField>'
               } ),
```

```
            '</XMI.CorbaTcStruct>'
  }
```

Producer Object Modifications

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

Each struct defined within the outermost typecode is held while the XML representation of the typecode is generated. If a recursive sequence is employed within the typecode, the sequence of structs will be required.

9.4.8.6 TcSequence

This production generates a representation of a sequence typecode. CORBA allows the type of the sequence elements to be defined as a struct which encloses this sequence, known as a recursive sequence. The production must represent that type is a special manner.

TcSequence ::= `<XMI.CorbaTcSequence xmi.tcLength="` *type-length* `">` (TypeCodeState | TcRecursiveLink) `</XMI.CorbaTcSequence>`

A sequence length of zero means unbounded – no upper limit on the number of elements. If the constructedTcList attribute contains the typecode representation of the element type, then this is a recursive sequence.

```
TcSequence(tc : TypeCode) : Sequence(string)
TcSequence(tc) =
  Sequence{ '<XMI.CorbaTcSequence xmi.tcLength="',
            tc.length(),
            '">',
            (if self.constructedTcList->includes(tc.content_type()) then
               TcRecursiveLink(tc, tc.content_type())
             else
               TypeCodeState(tc.content_type(), '')
             endif),
            '</XMI.CorbaTcSequence>'
          }
```

9.4.8.7 TcArray

The Array typecode is represented similarly to the sequence typecode, but recursive sequences are not permitted.

TcArray ::= `<XMI.CorbaTcArray xmi.tcLength="` *type-length* `">` TypeCodeState `</XMI.CorbaTcArray>`

The array element type is represented using the TypeCodeState operation.

```
TcArray(tc : TypeCode) : Sequence(string)
TcArray(tc) =
  Sequence{ '<XMI.CorbaTcArray xmi.tcLength="',
            tc.length(),
            '">',
            TypeCodeState(tc.content_type(), ''),
            '</XMI.CorbaTcArray>'
```

9.4.8.8 TcObjRef

A TypeCode element defining an object reference is generated with this production.

TcObjRef ::= `<XMI.CorbaTcObjRef xmi.tcName="` *type-name* `" tcId="` *type-id* `"/>`

This operation generates an empty element.

```
TcObjRef(tc : TypeCode) : Sequence(string)
TcObjRef(tc) =
  Sequence{ '<XMI.CorbaTcObjRef xmi.tcName="',
            tc.name(),
            '" xmi.tcId ="',
            tc.id(),
            '"/>'
  }
```

9.4.8.9 TcEnum

This production generates the representation of an enum TypeCode.

TcEnum ::= `<XMI.CorbaTcEnum xmi.tcName="` *type-name* `" xmi.tcId="` *type-id* `">`
`(` `<XMI.CorbaTcEnumLabel xmi.tcName="` *enum-label* `"/>` `)*`
`</XMI.CorbaTcEnum>`

The operation iterates over a zero-based sequence of integers equal in length to the number of enum labels in the typecode.

```
TcEnum(tc : TypeCode) : Sequence(string)
TcEnum(tc : TypeCode) : Sequence(string)
TcEnum(tc) =
  Sequence{ '<XMI.CorbaTcEnum xmi.tcName="',
            tc.name(),
            '" xmi.tcId =",
            tc.id(),
            '">',
            Sequence{ 0..(tc.member_count-1) }->collect(i |
                Sequence{ '<XMI.CorbaTcEnumLabel xmi.tcName="',
                          tc.member_name(i),
```

```
                                '"/>'
                    }
                }
                '</XMI.CorbaTcEnum>'
    }
```

9.4.8.10 TcUnion

This production generates the representation of a CORBA Union typecode.

TcUnion ::= `<XMI.CorbaTcUnion xmi.tcName="` *type-name* `" xmi.tcId="` *type-id* `">` TypeCodeState `(` `<XMI.CorbaTcUnionMbr xmi.tcName="` *field-name* `">` TypeCodeState AnyValue `</XMI.CorbaTcUnionMbr>` `)*` `</XMI.CorbaTcUnion>`

```
TcUnion(tc : TypeCode) : Sequence(string)
TcUnion(tc) =
  Sequence{ '<XMI.CorbaTcUnion xmi.tcName="',
            tc.name(),
            '" xmi.tcId=",
            tc.id(),
            '">',
            TypeCodeState(tc.discriminator_type(), ''),
            Sequence{ 0..(tc.member_count-1) }->collect(i |
               Sequence{ '<XMI.CorbaTcUnionMbr xmi.tcName="',
                         tc.member_name(i),
                         '">',
                         TypeCodeState(tc.member_type(i), ''),
                         AnyValue(tc.member_label(i)),
                         '</XMI.CorbaTcUnionMbr>'
               } ),
            '</XMI.CorbaTcUnion>'
  }
```

9.4.8.11 TcExcept

This production generates the representation of a CORBA exception typecode.

TcExcept ::= `<XMI.CorbaTcExcept xmi.tcName="` *exception-name* `" xmi.tcId ="` *exception-id* `">` `(` `<XMI.CorbaTcField xmi.tcName="` *field-name* `">` TypeCodeState `</XMI.CorbaTcField>` `)*` `</XMI.CorbaTcExcept>`

```
TcExcept(tc : TypeCode) : Sequence(string)
TcExcept(tc : TypeCode) : Sequence(string)
TcExcept(tc) =
  Sequence{ '<XMI.CorbaTcExcept xmi.tcName="',
```

```
            tc.name(),
            '" xmi.tcId=",
            tc.id(),
            '">',
            Sequence{ 0..(tc.member_count-1) }->collect(i |
               Sequence{ '<XMI.CorbaTcField tcName="',
                         tc.member_name(i),
                         '">',
                         TypeCodeState(tc.member_type(i), ''),
                         '</XMI.CorbaTcField>'
              } ),
            '</XMI.CorbaTcExcept>'
  }
```

9.4.8.12 TcString

CORBA supports differing string types, since each type may specify a distinct length for its values.

TcString ::= `<XMI.CorbaTcString xmi.tcLength="` *type-length* `"/>`

A length of zero indicates that the string length is unbounded.

```
TcString(tc : TypeCode) : Sequence(string)
TcString(tc) =
  Sequence{ '<XMI.CorbaTcString xmi.tcLength="',
            tc.length(),
            '"/>'
  }
```

9.4.8.13 TcWstring

This production is similar to the TcString production, except that it is for wide strings.

TcWstring ::= `<XMI.CorbaTcWstring xmi.tcLength="` *type-length* `"/>`

A length of zero indicates that the string length is unbounded.

```
TcWstring(tc : TypeCode) : Sequence(string)
TcWstring(tc) =
  Sequence{ '<XMI.CorbaTcWstring xmi.tcLength="',
            tc.length(),
            '"/>'
  }
```

9.4.8.14 TcFixed

TcFixed ::= `<XMI.CorbaTcFixed xmi.tcDigits="` *digits* `" xmi.tcScale="` *scale* `"/>`

```
TcFixed(tc : TypeCode) : Sequence(string)
TcFixed(tc)=
  Sequence{ '<XMI.CorbaTcFixed xmi.tcDigits="',
                      tc.fixed_digits(),
                      '" xmi.tcScale="',
                      tc.fixed_scale(),
                      '"/>'
  }
```

9.4.8.15 TcSimple

For the rest of the TypeCodes, the empty element itself completely describes the typecode.

TcSimple ::= `(<XMI.CorbaTcShort/>` | `<XMI.CorbaTcLong/>` | `<XMI.CorbaTcUshort/>` | `<XMI.CorbaTcUlong/>` | `<XMI.CorbaTcFloat/>` | `<XMI.CorbaTcDouble/>` | `<XMI.CorbaTcBoolean/>` | `<XMI.CorbaTcChar/>` | `<XMI.CorbaTcWchar/>` | `<XMI.CorbaTcOctet/>` | `<XMI.CorbaTcAny/>` | `<XMI.CorbaTcTypeCode/>` | `<XMI.CorbaTcPrincipal/>` | `<XMI.CorbaTcNull/>` | `<XMI.CorbaTcVoid/>` | `<XMI.CorbaTcLongLong/>` | `<XMI.CorbaTcLongLong/>` | `<XMI.CorbaTcLongDouble/>`)

```
TcSimple(tc : TypeCode) : Sequence(string)
TcSimple(tc) =
  if tc.kind() = tk_short then
    '<XMI.CorbaTcShort/>'
  else
    if tc.kind() = tk_long then
      '<XMI.CorbaTcLong/>'
    else
      if tc.kind() = tk_ushort then
        '<XMI.CorbaTcUshort/>'
      else
        if tc.kind() = tk_ulong then
          '<XMI.CorbaTcUlong/>'
        else
          if tc.kind() = tk_float then
            '<XMI.CorbaTcFloat/>'
          else
            if tc.kind() = tk_double then
              '<XMI.CorbaTcDouble/>'
```

```
                else
                  if tc.kind() = tk_boolean then
                    '<XMI.CorbaTcBoolean/>'
                  else
                    if tc.kind() = tk_char then
                      '<XMI.CorbaTcChar/>'
                    else
                      if tc.kind() = tk_wchar then
                        '<XMI.CorbaTcWchar/>'
                      else
                        if tc.kind() = tk_octet then
                          '<XMI.CorbaTcOctet/>'
                        else
                          if tc.kind() = tk_any then
                            '<XMI.CorbaTcAny/>'
                          else
                            if tc.kind() = tk_TypeCode then
                              '<XMI.CorbaTcTypeCode/>'
                            else
                              if tc.kind() = tk_Principal, then
                                '<XMI.CorbaTcPrincipal/>'
                              else
                                if tc.kind() = tk_null then
                                  '<XMI.CorbaTcNull/>'
                                else
                                  if tc.kind() = tk_void then
                                    '<XMI.CorbaTcVoid/>'
                                  else
                                    if tc.kind() = tk_longlong then
                                      '<XMI.CorbaTcLongLong/>'
                                    else
                                      if tc.kind() = tk_ulonglong then
                                        '<XMI.CorbaTcUlongLong/>'
                                      else
                                        if tc.kind() = tk_longdouble then
                                          '<XMI.CorbaTcLongDouble/>'
                                        else
                                          -- undefined (not expected)
                                        endif
                                      endif
                                    endif
                                  endif
                                endif
                              endif
                            endif
                          endif
                        endif
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  endif
```

9.4.8.16 TcRecursiveLink

CORBA allows the recursive definition of TypeCodes. Without a special mechanism, it would be impossible to represent this recursion in XML. CORBA restricts the use of recursion to the definition of the items of a sequence. If a sequence is defined within another type, and either that type, or some other type enclosing this sequence is a struct, then the sequence items can be of that struct type. The type of the items is then represented by this element. The offset value is the same as the offset in a TypeCode representing a recursive sequence, specified as:

> "The offset parameter specifies which enclosing struct or union is the target of the recursion, with the value one indicating the most immediate enclosing struct or union, and larger values indicating successive enclosing struct or unions." [CORBA]

So in this type:

```
struct foo {
    long value;
    sequence<foo> chain;  };
```

the offset of the recursive link defining the type of chain items would be1.

TcRecursiveLink ::= `<XMI.CorbaRecursiveType xmi.offset="` *offset-to-sequence* `"/>`

The distance operation calculates the distance between the sequence type and struct content type.

```
TcRecursiveLink(seqTc : TypeCode, contentTc : TypeCode) : Sequence(string)
TcRecursiveLink(seqTc, contentTc) =
  Sequence{ '<XMI.CorbaRecursiveType xmi.offset="',
            TcDistance(contentTc, seqTc, 0),
           '/>'
  }
```

9.4.9 Helpers

The following operations support the above productions and operations.

9.4.9.1 LinksInRoot

Given an association from the metamodel, this operations detects and returns all instances of the association (links) in which both the objects at the link ends are in the model identified by the document root. The allContents operation is defined in the OCL of the MOF specification.

```
LinksInRoot(assoc : Association) : Sequence(Link)
LinksInRoot(assoc) =
```

```
  self.refPkg.getAssociation(a)->iterate(ra; links : Sequence(Link) |
      ra.allLinks()->select(l |
          self.root.allContents->includes(l->at(1)) and
               root.allContents->includes(l->at(2))))
```

9.4.9.2 UnreferencedAssoc

This operation returns all the Associations in a metamodel defined by a Package, which have no References corresponding to either AssociationEnd. The operation considers both the Associations immediately contained by the Package, as well as those which may be enclosed in nested Packages.

```
UnreferencedAssoc(pkg : Package) : sequence(Association)
UnreferencedAssoc() =
  pkg.allContents->select(c |
     c.oclIsTypeOf(Association) and not
     c.oclIsTypeOf(Association).contents->exists(ae |
        pkg.allContents->select(c2 | c2.oclIsTypeOf(Reference))->collect(r |
           r.referencedEnd)->includes(ae)))
```

9.4.9.3 AllClassProxies

This operation returns all the class proxies (instances of subtypes of RefObject) which are enclosed by the provided RefPackage, including those which may be enclosed by nested RefPackages.

```
AllClassProxies(pkgProxy : RefPackage) : Sequence(RefObject)
AllClassProxies(pkgProxy) =
  pkgProxy.metaObject().oclAsType(Package).findElementsByTypeExtended
      (Class, false)->collect(c | pkgProxy.getClassRef(c))->union
          (pkgProxy.metaObject().oclAsType(Package).findElementsByTypeExtended
               (Package, false)->collect(p | AllClassProxies(p)))
```

9.4.9.4 AllUncontainedObjects

This operation returns all the objects in the extent of the RefPackage which do not participate in a composite aggregation on the contained end (not contained by any other object).

```
AllUncontainedObjects(pkgProxy : RefPackage) : Sequence(RefObject)
AllUncontainedObjects(pkgProxy) =
  AllClassProxies(pkgProxy)->collect(c | c.allObjects(false))->select(obj |
    CompAssocProxies(pkgProxy)->iterate(ap; answer : boolean = false |
     answer or
     (if End(1, ap.metaObject().oclAsType(Association)).aggregation =
                                                        composition then
        ap.query(End(2, ap.metaObject().oclAsType(Association)), obj)->isEmpty
      else
```

```
        ap.query(End(1, ap.metaObject().oclAsType(Association)), obj)->isEmpty
    endif))
```

9.4.9.5 CompAssocProxies

This operation returns all the Associations enclosed in a RefPackage (directly or indirectly) which are defined as composite aggregations.

```
CompAssocProxies(pkgProxy : RefPackage) : Sequence(RefAssociation)
CompAssocProxies(pkgProxy) =
  pkgProxy.metaObject().oclAsType(Package).findElementsByTypeExtended
      (Association, false)->select(a |
          End(1, a).aggregation = composite or
              End(2, a).aggregation = composite)->collect(ca |
                  pkgProxy.getAssociation(a))
```

9.4.9.6 End

This operation returns one of the AssociationEnds of the provided Association. The index indicates whether the requested end is the first end (when the index equals 1) or the second end.

```
End(index : integer, assoc : Association) : AssociationEnd
End(i, assoc) =
  assoc->findElementsByType(AssociationEnd, false)->at(i)
```

9.4.9.7 ExtractSequence

This operation returns an OCL sequence corresponding to the elements of a CORBA sequence, when the Any-typed value is sequence.

```
ExtractSequence(value : Any) : Sequence(Any)
ExtractSequence(value) =
  ORB.create_dyn_any(value).oclAsType(DynSequence).get_elements()
```

9.4.9.8 ExtractField

This operation returns the value of a field of a struct value, where the field is specified using a zero-based index into the fields of the struct.

```
ExtractField(value : Any, i : long) : Any
ExtractField(value) =
    ORB.create_dyn_any(value).oclAsType(DynStruct).get_members()->at(i).value()
```

9.4.9.9 ExtractUnionField

This operation extracts the field value (member value) from a value of type union.

```
ExtractUnionField(value : Any) : Any
ExtractUnionField(value) =
    ORB.create_dyn_any(value).oclAsType(DynUnion).member().to_any()
```

9.4.9.10 ExtractUnionDiscrim

This operation extracts the disrciminator value from a union-typed value.

```
ExtractUnionDiscrim(value : Any) : Any
ExtractUnionDiscrim(value) =
    ORB.create_dyn_any(value).oclAsType(DynUnion).discriminator().to_any()
```

9.4.9.11 GetUnionFieldType

This operation returns the type of the field of a specific value of a union type.

```
GetUnionFieldType(value : Any) : TypeCode
-- cannot be represented by OCL
-- need to seek to member field
-- for expression, seek to member field before applying the member_type
-- operation
UnionFieldType(value) =
   DeAlias(SeekToUnionField(ORB.create_dyn_any(value).oclAsType(DynUnion))
      .member_type().to_any()
```

9.4.9.12 DeAlias

Return a typecode which is not an alias typecode. If the input typecode is not an alias typecode, it returns the input typecode. Otherwise, it de-aliases the originating (content) type of the typecode.

```
DeAlias(tc : TypeCode) : TypeCode
DeAlias(tc) =
  if tc.kind() = tkalias then
    DeAlias(tc.content_type())
  else
    tc
  endif
```

9.4.9.13 IdOfObject

This helper returns the locally unique identifier for an object if it is in the inventory of objects. Otherwise, it returns the empty string.

```
IdOfObject(obj : RefObject) : Sequence(string)
  if Sequence{ 1..(self.objectInventory->size) }->select(i |
       self.objectInventory->at(i) = obj)->isEmpty then
    if InScope(obj) then
      NewObjectId(obj)
    else
      ''
    endif
  else
    self.objectIds.at(Sequence{ 1..(self.objectInventory->size) }->select(i |
       self.objectInventory->at(i) = obj)->first)
  endif
```

9.4.9.14 UrlOfObject

This helper returns an URL which links to a representation of the object. This URL will typically link to an element in another XMI document.

```
UrlOfObject(obj : Object) : Sequence(string)
  -- not specified in OCL
  -- returns a legal URL which links to a representation of the object
```

9.4.9.15 DotNotation

Returns a string representation of the sequence of strings defining the qualified name. elements of the originating sequence are separated with dot (period) characters.

```
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)
```

9.4.9.16 NewObjectId

This helper generates a new object identifier. These identifiers are required to be locally unique within the XMI document, but they can also be unique in a wider context; e.g. a UUID.

```
NewObjectId(obj : RefObject) : string
  -- not specified
  -- any string which is a legal value of the XML ID type, and which
  -- is not currently used in the XML document, nor allocated to an object
  -- in this production
  -- can also be a string representation of a UUID
```

Producer object Modifications:

```
self.objectInventory ← self.objectInventory->append(obj)
self.objectIds ← self.objectIds.append(result)
```

When a new id is created, it must be associated with the object within the state of the Producer, so the id may be retrieved if any subsequent references to the element representing that object are desired.

9.4.9.17 InScope

This operation determines whether the provided object is part of the model being written out as an XML document.

```
InScope(obj : RefObject) : boolean
InScope(obj) =
  if self.byContainment then
    self.root.allContents->includes(obj)
  else
    AllClassProxies(self.refPkg)->collect(c | c.allObjects(false))->includes(obj)
  endif
```

9.4.9.18 HexAsString

This helper converts a CORBA octet value into a two character hexadecimal string. The syntax for the string is:

[0-9A-Fa-f][0-9A-Fa-f]

```
HexAsString(value : octet) : string
  -- not specified
```

9.4.9.19 IntegralAsString

These six helper operations return the string representation of an integral value. The resulting string consists of one ore more decimal digits with an optional leading sign. The syntax for the string is:

[+-]? [0-9]+

```
IntegralAsString(value : short) : string
IntegralAsString(value : unsigned short) : string
IntegralAsString(value : long) : string
IntegralAsString(value : unsigned long) : string
IntegralAsString(value : long long) : string
IntegralAsString(value : unsigned long long) : string
  -- not specified
```

9.4.9.20 RealAsString

These three helper operations return a decimal string representation of a floating point value. All sig-

nificant digits in the value should be included. The syntax for the representation is:

[-+]? (([0-9]+ '.' [0-9]*) | ('.' [0-9]+)) ([eE] [+-]? [0-9]+)?

```
RealAsString(value : float) : string
RealAsString(value : double) : string
RealAsString(value : long double) : string
  -- not specified
```

9.4.9.21 BooleanAsString

This helper translates a boolean value into a string.

```
BooleanAsString(value : Any) : string
  BooleanAsString(value) =
  if value.type().kind() = tk_boolean then
    if value.extract_boolean() then
      'true'
    else
      'false'
    endif
  else
    -- should not happen
  endif
```

9.4.9.22 EnumAsString

This helper translates an enumeration value into a string.

```
EnumAsString(value : Any) : string
EnumValue(value) =
  value.type().element_name(value.create_input_stream().read_long())
```

9.4.9.23 EncodedString

This helper encodes a string by escaping any embedded XML markup characters.

```
EncodedString(str : string) : string
  -- each occurance of the less-than character
  -- (<) with the lt entity (&lt;), each
  -- occurance of the greater-than character
  -- (>) with the gt entity (&gt;), and
  -- each occurance of the ampersand character
  -- (&) with the amp entity (&amp;)
```

9.4.9.24 EncodedCharacter

This helper encodes a character by escaping it if it is a markup character.

```
EncodedCharacter(str : character) : character
  -- converts a character if it is one of (<>&)
  -- replace the less-than character
  -- (<) with the lt entity (&lt;),
  -- replace the greater-than character
  -- (>) with the gt entity (&gt;), and
  -- replace the ampersand character
  -- (&) with the amp entity (&amp;)
  -- otherwise, returns character unchanged
```

9.4.10 CORBA-Specific Helpers

The following operations support the CORBA-specific productions and operations.

9.4.10.1 CorrespondingDataType

Finds a DataType in the metamodel that corresponds to the given TypeCode, returning the fully qualified name of the DataType in dot notation. If no such DataType exists, an empty string is returned. If multiple DataTypes using the same TypeCode exist in the metamodel, any one is allowed.

```
CorrespondingDataType(tc : TypeCode) : string
DotNotation(self.metamodel.allElements>select(e |
    e.oclIsOfType(DataType))->select(dt |
        dt.typeCode = tc)->first.qualifiedName)
```

9.4.10.2 CorbaTypeName

This helper produces a textual name for a TcKind value. For primitive types, this name is sufficient to fully describe the corresponding type.

```
CorbaTypeName(kind : TcKind) : string)
CorbaTypeName(kind) =
  if kind = tk_objref then
    'objref'
  else
    if kind = tk_short then
      'short'
    else
      -- the other cases omitted (deduce from pattern above)
    endif
  endif
```

9.4.10.3 IdOfTypeDefinition

This operation returns the URL pointing to the type definition element corresponding to the typecode.

```
IdOfTypeDefinition(tc : TypeCode) : Sequence(string)
IdOfTypeDefinition(tc) =
if Sequence{ 1..(self.tcInventory->size) }->select(i |
       self.tcInventory->at(i) = tc)->isEmpty then
    NewTcId(tc)
  else
    self.tcIds.at(Sequence{ 1..(self.tcInventory->size) }->select(i |
       self.tcInventory->at(i) = tc)->first)
  endif
```

9.4.10.4 NewTcId

This helper finds all TypeCode values in a metaobject. This includes TypeCode values embedded in Anys.

```
NewTcId(tc : TypeCode) : string
  -- not specified
  -- any string which is a legal value of the XML ID type, and which
  -- is not currently used in the XML document, nor allocated to an object
  -- in this production
```

Producer object Modifications:

```
self.tcInventory ← self.tcInventory->append(tc)
self.tcIds ← self.tcIds->append(result)
self.typeDefinitions ←
   self.typeDefinitions.append(TypeCodeValue(tc), result)
```

When a new id is created, it must be associated with the object within the state of the Producer, so the id may be retrieved if any subsequent references to the element representing that object are desired.

9.4.10.5 TcDistance

This operation measures the distance from a TypeCode, provided in the first argument, to an enclosing struct TypeCode equal to the second argument.

```
TcDistance(outer : TypeCode, inner : TypeCode, len : integer) : integer
TcDistance(outer, inner, len) =
  if outer.kind() = tk_struct or outer.kind() = tk_union then
    Sequence{ 0..outer.member_count-1 }->collect(i |
        TcDistance(outer.member_type(i), inner, len+1))->
            iterate(e; mx : Integer = 0 | max(e, mx))
  else
    if outer = inner then
      len
    else
      0
    endif
  endif
```

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.

10.2 XMI and W3C DCD

IBM and Microsoft have collaborated and proposed a new W3C proposal based on XML - Document Content Definition (DCD). DCD is richer than DTD and has better data structuring and data typing capabilities, thus making it an attractive target for XMI in the future. The XMI submitters anticipate that when the DCD specification solidifies, mappings from XMI to DCD will be produced as an evolution of XMI. Another W3C initiative that could influence future XMI direction is XML-Schema work that is getting underway.

10.3 XMI and CDIF

EIA CDIF (Electronic Industry Associates - Case Data Interchange Format) was proposed as one of the initial submissions to the SMIF RFP. The CDIF submitters have collaborated with the XMI submitters to incorporate key aspects of CDIF such as the use of unique IDs into the XMI final submission. Preliminary work on providing a migration path from CDIF to XMI has begun and technical feasibility has been assessed.

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. This analysis has been made by CDIF and MOF experts between the times of initial and final submission. The experience of CDIF designers in metamodeling architectures as well as the experience in the unique ID for meta information has been worked into the current XMI proposal. The work on a migration path from CDIF Transfer Format to XMI is expected to be done in the by the OMG Object Analysis and Design Task Force.

The co-submitters and supporters of the CDIF proposal have helped improve the XMI submission and are now part of the final submission team for this XMI proposal

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 6.2.2, *Requirements for XMI DTDs*, Section 6.4, *XMI DTD and Document Structure*, and Section 6.5, *Necessary 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 6.4, *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 6.5, *Necessary 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 MOF Subset

- XMI support for MOF meta-models (at the M2 level only) beyond the following subset is optional:

1. Data types not contained explicitly within the metamodel.
2. Metamodels having different names for MOF reference ends as association ends.
3. Metamodels having association ends without references.
4. Metamodels containing static attributes.
5. Metamodels with nested classes.

11.3.2 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 6.2, Section 6.3, Section 6.5, Section 6.6, Section 7.3, and Section 7.4.
- Incomplete model DTD generation rules (Section 6.7) may be used to support transmission of incomplete models. Either all incomplete rules or no incomplete rules should be supported. The incomplete model DTD is a different DTD than the complete model DTD. Support for incomplete models is an optional addition to the mandatory support for complete models.
- DTDs may support the CORBA typecode mapping (6.5.17) or the general data type mapping (6.11).
- Contained elements may optionally have a role for their container with lower bound multiplicity of zero.

11.3.3 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 6.5 and Section 6.10. 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.
- Processing of XMI differencing elements (Section 6.9) is an optional compliance point. Either all differencing elements are produced and processed, or no differencing elements are produced and processed.
- Documents may support the incomplete model DTD (Section 6.7) or the complete model DTD.
- Documents may support the CORBA typecode mapping (6.5.17) or the general data type mapping (6.11).
- Contained elements may optionally have a role for their container with lower bound multiplicity of zero.

11.3.4 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

[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. DCD supercedes XML-Data.

[DCD] Document Content Description, an XML submission to the W3C for data types and element dclarations. http://w3c.org/TR/NOTE-dcd DCD, by Texuality, Microsoft, and IBM, supercedes 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).

[ISO8601] "Data elements and interchange formats -- Information interchange -- Representation of dates and times", ISO 8601:1988

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.

The following is the Open Group DCE standard on UUIDs.

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

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 listed 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
- Appendix M1 of the UML 1.1 specification
- 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, *aggregate [class]* may be referred to as simply *aggregate*.

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 instantiated.

abstraction — A group of essential characteristics of an *entity* that distinguish it from other entities. An *abstraction* defines a boundary relative to the perspective of the viewer.

abstract language — A system of expression for expressing information that is independent of any particular human readable *notation*. Constrast: *concrete language* or *notation*. (MOF)

actual parameter — Synonym: *argument*.

aggregate [class] — A *class* that represents the "whole" in an *aggregation* (whole-part) relationship. See: *aggregation*. (UML)

aggregation — A special form of *association* that specifies a whole-part relationship between the aggregate (whole) and a component part. See: *composition*.

analysis — A phase 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.

annotation — Synonym: *note*. (MOF)

any

A CORBA primitive data type. A strongly typed “universal union” type that can contain any value whose type is a CORBA data type. This data type is typically used in CORBA IDL when it is not possible to choose an appropriate type at the time the interface is defined. Use of CORBA anys entails *dynamic type checking*, and extra overheads in value transmission. See *strong typing*, *dynamic typing*, *TypeCode*. (CORBA)

architecture

The organizational structure of a *system*. An architecture can be recursively decomposed into parts that interact through *interface*s, *relationship*s that connect parts, and *constraint*s on the way that parts can be assembled.

argument

A specific value corresponding to a *parameter*. Synonym: *actual parameter*.

array

1. A CORBA constructed data type.
2. A *collection* (1) whose type fixes the number of elements. The *ordering* and uniqueness properties of an array are indeterminate. (MOF)

artifact

A piece of information that is used or produced by a software development process. An artifact can be a model, a description or a piece of software.

association

1. A semantic relationship two or more types describes a set of connections between their respective instances. (UML)
2. An *association* (1) between classes. (MOF)

Association

A model element that defines an *association* (2) in a MOF metamodel. (MOF)

association end

See: *association role*.

AssociationEnd

A model element that defines an *association end* in a MOF metamodel. (MOF)

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. (UML)

association role

The *role* that a *type* or *class* plays in an *association*. Synonym: *association end*.

attribute

1. An attribute of an object is an identifiable association between the object and some other entity or entities. (OMA)
2. An attribute is a named property of a type. (UML)
3. An attribute is a named property of a class. (MOF)

Attribute

A model element that defines an *attribute* in a MOF metamodel. (MOF)

bag

An *unordered collection* in which duplicate members are allowed. (MOF)

base type

The base type of a *collection* (1) is the *type* (1) of its elements.

behavior

The observable effects of an operation, including its results (MOF). Synonym: behavior (OMA)

binary association

An *association* between two classes. The degenerate case of an *n-ary association* where “n” is two.

boolean

1. A UML *enumeration* type whose values are true and false. (UML)
2. A *CORBA* primitive *data type* whose values are true and false. (CORBA)

builtin type — A type in a type system which is available as a predefined type in all instantiations of the type system; e.g. “short” and “string” are builtin types in CORBA IDL. Constrast: *primitive type*.

boolean expression — An *expression* that evaluates to a *boolean* value.

CDATA section — A part of an *XML Document* in which any markup (e.g. tags) is not interpreted, but is passed to the application as is. (W3C)

cardinality — The number of elements in a *collection*. Constrast: *multiplicity*.

class — 1. A *type* (3) that characterizes objects that share the same attributes, operations, methods, relationships, and semantics. (UML)
2. An implementation that can be instantiated to create multiple objects with the same behavior. Types classify objects according to a common interface; classes classify objects according to a common implementation. (OMA)

Class — A model element that defines an *class* (1) in a MOF metamodel. (MOF)

classifier — 1. A category of UML model elements that roughly correspond to types in programming languages. The category includes *association class*es, *class*es (1), *data type*s (2), *interface*s, *subsystem*s and *use case*s. (UML)
2. The category of MOF model elements analogous to classifier (1):

classifier level — In MOF metamodels and UML models, this label indicates that the labelled feature is common to all instances of its classifier. For example, a classifier level attribute of a class is common to all instances of the class. Synonym: *static*. Contrast: *instance level*. (UML, MOF)

class diagram — A UML *diagram* that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships. (UML)

class proxy — A MOF *metaobject* that carries the *classifier level* attributes and operations for an instance of a MOF class. (MOF)

client — A type, class, or component that requests a service from another type, class or component. (UML)

closure — The *transitive closure* of some object under some relationship or relationships.

collection — 1. A group of values or objects. The values in a collection are often refered to as members or elements of the collection.
2. A *collection* (1) in which the members are instances of the same base type. The type of a collection is defined by the *base type* and a *multiplicity*. See: *array*, *sequence*, *bag*, *set*, *list* and *unique list*. (MOF)

compile time — Indicates something that occurs during the compilation of a software module.

component — An executable software module with an identity and a well-defined interface.

composite [class] — A class that is related to one or more classes by a composition relationship. See: *composition*.

composite aggregation — Synonym: *composition*.

composition — A form of *aggregation* 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*. (UML)

concrete class — A *class* that can be directly instantiated. Contrast: *abstract class*.

concrete language — Synonym: *notation*.

constraint — A semantic condition or restriction. Certain constraints are predefined, others may be user defined. Constraints may be expressed in *natural language* or a *formal language*. (UML, MOF)

Constraint — A model element that defines a *constraint* on another element in a *MOF metamodel*. (MOF)

container — 1. An entity that exists to contain other entities. See *containment*.
2. An entity's container is the entity that contains it.

containment — A form of aggregation that is similar to *composition*. The fundamental properties of containment are:

- an entity can have at most one container at any given time, and
- an entity cannot directly or indirectly contain itself.

containment hierarchy — A containment hierarchy is a tree-shaped graph of entities, consisting of a root entity and all other entities that are directly or indirectly contained by it.

containment matrix — A set of constraints on a containment relationship (expressible as a matrix of boolean values) that determine what other kinds of entities a given kind of entity can contain. For example, the MOF Model definition includes such a matrix to specify which concrete subclasses of ModelElement can be contained by each concrete subclass of Namespace. (MOF)

CORBA — Acronym: The Common Object Request Broker Architecture.

CORBA IDL — Synonym: *IDL*.

data — A representation of information.

data type — A type whose values have no *identity*. The data types in a type system are typically into the primitive built-in types, and constructed types such as enumerations and so on.

DataType — A model element that defines a *data type* on another element in a *MOF metamodel*. (MOF)

dependency — 1. A *relationship* between two entities in which a change to an aspect of one entity affects the other (dependent) entity in some way.
2. A *dependency* (1) between two modeling elements such that a change to an element changes the meaning of the dependent element. (UML, MOF)

derived attribute — An pseudo-attribute whose value is not stored explicitly as part of an object, but is calculated from other state when required. Derived attributes can also be updated. (MOF)

derived association — A pseudo-association whose component links are not stored explicity, but are calculated from other state when queried. Derived associations can also be updated. (MOF)

derived element — 1. A model element whose value 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. (UML)
2. An element in a *metamodel* that is derived from other metamodel elements, and yet is visible in the interfaces produced by an object mapping. See *derived attribute*, *derived association*. (MOF)

design — The phase of the software development process whose primary purpose is to decide how the system will be implemented. During the design phase, 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. Contrast: *analysis time*.

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 element*s).

document element — See root element. (XML)

Document Type Definition — See DTD (XML)

domain — An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic typing — A category of *type safety* that can only be enforced by *dynamic type checking*. Type systems with dynamic typing are more expressive than those with static typing only. at the cost of run time overheads and potential type errors. Contrast: *static typing*.

dynamic type checking — A *type checking* activity that occurs at *run time*. Contrast: *static type checking*.

DTD — 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. (XML)

EBNF — Acronym: Extended Backus-Naur Form. A widely used notation for expressing *grammar*s.

element — 1. An atomic constituent of a *model*. Synonym: *model element*. (MOF, UML)
2. A logical unit of information in a XML document. An XML element consists of a *start tag*, an *element content* and a matching *end tag*. (XML)

element attributes The name-value pairs that can appear within the *start tag* of an *element* (2). (XML)

element content The elements or text that is contained between the *start tag* and *end tag* of an element. (XML)

element type A particular type of element, such as a paragraph in a document or a class in an XMI encoded metamodel. The element type is indicated by the name that occurs in its start-tag and end-tag. (XML)

empty string A *string* with zero characters.

end tag A tag that marks the end of an element, such as </Model>. See *start tag*. (XML)

entity 1. A "thing".
2. An item of interest in a system being modelled.

enumeration 1. A type that is defined as a finite list of named values. For example, Color = {Red, Green, Blue}. (UML)
2. A kind of constructed data type in the CORBA type system. (CORBA)

export 1. To transmit a description of an object to an external entity. (OMA)
2. In the context of packages, to make an element visible outside of its enclosing *namespace*. See: *visibility*, *import* (2). (UML)

expression A formula in some language that can be evaluated in some context to give a value. For example, the expression (7 + 5 * 3) evaluates to 22.

extent The set of objects that belong to a MOF package instance, class proxy or association instance. (MOF)

feature A (meta-)model element that defines part of another (meta-)model element. For example an UML class has attributes and operations as features. (UML, MOF)

formal language A *language* with a specified syntax and meaning.

formal parameter Synonym: *parameter*.

framework A micro-architecture that provides an extensible template for applications within a specific domain. (UML)

frozen Synonym: *immutable*. (MOF)

grammar A formal specification of the syntax of a *language*.

generalizable element A model element that may participate in a generalization relationship. See: *generalization*. (UML)

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: *specialization*.

generic interface Interfaces that are shared by all MOF metaobjects. See *Reflective*. Contrast: *specific interfaces*. (MOF)

HTML Acronym: Hyper Text Markup Language. A language for associating visual markup and hyperlinks with textual information that is one of the cornerstones of the World Wide Web. HTML is a particular application of *SGML*. (W3C)

identifier A value that denotes an *instance* with *identity*. See: *name*, *object reference*.

identity "Thingness". A *instance* has identity if it can be distinguished from other instances irrespective of its component values. For example, objects have identity but numbers do not.

IDL 1. Acronym: Interface Definition Language. The OMG language for specifying CORBA object interfaces. (OMA)
2. An interface specification in CORBA *IDL* (1) - colloquial.

IDL mapping 1. A mapping of the design expressed in a model onto CORBA IDL.
2. An *IDL mapping* (1) defined in the MOF standard that maps a *MOF metamodel* into CORBA IDL for metaobjects that represent metadata for the metamodel.

immutable The property of an entity or value that it will never change. For example, the number 42 is immutable. Synonym: *frozen*. Constrast: *read only*. (MOF)

implementation 1. An artifact that is the realization of an abstraction in more concrete terms. For example, a class is an implementation of a type, a method is an implementation of an operation. (UML)
2. A realization of a design object in engineering technology; e.g. IDL or program source code.
3. The process of producing an *implementation* (1)(2).

implementation inheritance The use of inheritance to produce one implementation artifact from another implementation artifact. Implementation inheritance presupposes interface inheritance.

import 1. To create an object based on a description of an object transmitted from an external entity. See *import* (1). (OMA)
2. In the context of *package*, a *dependency* that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *export* (2). (UML)
3. A relationship between packages in a MOF metamodel that makes the contents of the imported package visible within the importing *package*. (MOF)

Import A model element that in a MOF metamodel that specifies that one package imports another package. (MOF)

information The conjunction of *data* and structure. For example, facts.

inheritance The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See *generalization*. (UML, MOF)

instance 1. An instance of a *type* (1) is some value that satisfies the type predicate. (ODP)
2. An object created by instantiating a class. (OMA)
3. An entity to which a set of operations can be applied and which has a state that stores the effects of the operation. (UML)

instance level — In MOF metamodels and UML models, this label indicates that the labelled feature is common to all instances of its classifier. For example, a classifier level attribute of a class is common to all instances of the class. Contrast: *classifier level*. (UML, MOF)

instantiate — The act or process of making an *instance* of something. See: *reify*.

interface — A *type* (1) that describes the externally visible behavior common to a set of objects. An interface includes the *signature*s of any operations common to all of the objects.

interface inheritance — The inheritance of the interface of a more specific element. This does not imply inheritance of behavior.

introspection — A style of programming in which a program is able to examine parts of its own definition. Constrast: *reflection* (1).

invariant — A *constraint* on an entity or group of entities that must hold at all times.

link — A semantic connection between a tuple of objects. An instance of an association. See: *association*.

link role — An instance of an *association role*. See: *link*, *role*.

list — A *collection* in which the order of the contents is significant, and duplicates are allowed. An ordered collection. See: Set, Array, Unique list.

knowledge — The conjunction of *information* with some aspect of understanding.

language — A means of expression. See *abstract language*, *concrete language*, *natural language*.

markup — Information that is intermingled with the text of an XML document to indicate its logical and physical structure. (XML)

member — Synonym: *feature*.

meta- — A prefix that denotes a Describes relationship. For example, “metadata” describes “data”. (MOF)

metadata — 1. Data that describes other data. A constituent of a model. (MOF)
2. An inclusive term for metadata (1), meta-metadata and meta-meta-metadata. (XMI)

meta-level — The level of “meta-”ness of a concept in a metadata framework.

meta-metadata — Data that describes metadata. A constituent of a metamodel. (MOF)

meta-meta-metadata — Data that describes meta-metadata. A constituent of a meta-metamodel. (MOF)

meta-metamodel — A model that defines an *abstract language* for expressing *metamodel*s. The relationship between a *meta-metamodel* and a *metamodel* is analogous to the relationship between a *metamodel* and a *model*. See: *MOF Model, the*. (MOF)

metamodel — A model that defines an *abstract language* for expressing other *model*s. An instance of a *meta-metamodel*. See: *MOF metamodel*. (MOF)

metamodel elaboration — The process of generating a repository type from a published metamodel. Can includes the generation of interfaces and repository implementations for the metamodel being elaborated. (MOF)

metaobject

1. An object that represents *metadata* (2). (MOF)
2. Often, a MOF metaobject. (MOF)

metaobject protocol

A *reflection* (1) technology in which a program can alter the behavior of the instances of a class by send a message to its metaclass. This style of reflection is not part of the MOF specification.

Meta Object Facility, the

See: *MOF, the*.

method

The implementation of an operation. The algorithm or procedure that effects the results of an operation. (UML)

model

1. A semantically closed abstraction of a system. See: *system*. (UML)
2. A semantically closed collection of *metadata* described by a single *metamodel*. (MOF)

model aspect

A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel. (MOF)

model element

Synonym: *element*. (MOF, UML)

ModelElement

The abstract superclass of all model elements in a *MOF metamodel*. (MOF)

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: *component*.

MODL

Acronym: Meta Object Definition Language. A textual language developed by DSTC that can be used to define MOF metamodels. (MOF)

MOF, the

1. Acronym: Meta Object Facility. The OMG adopted standard for representing and managing metadata. (MOF)
2. A metadata service that implements the *MOF, the* (1) specification. (MOF)

MOF-based model

Synonym: *MOF model.*

MOF-based metamodel

Synonym: *MOF metamodel.*

MOF meta-metamodel

Synonym: *MOF Model, the.*

MOF metamodel

A metamodel whose meta-metamodel is the MOF Model. (MOF)

MOF model

A model (2) whose metamodel is a *MOF metamodel*. (MOF)

MOF Model, the

The MOF Model is the standard meta-metamodel that is used to describe all MOF metamodels. It is defined in the MOF specification. (MOF)

multiple inheritance

A kind of inheritance in which a type may have more than one supertype.

multiplicity

1. 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. (UML)
2. A specification of the allowable cardinalities of the values of an attribute, parameter or association end, along with its uniqueness and orderedness. In the MOF, the allowable cardinalities of a multiplicity must form a contiguous subrange of the non-negative integers. (MOF)

multi-valued

A ModelElement with multiplicity said to be multi-valued when the 'upper' bound of its multiplicity is greater than one. The term does not the number of values held by an attribute, parameter, etc., at any point in time, but rather to the number of values that it can have at one time. Contrast: *single-valued*. (MOF)

n-ary association

An association involving three or more classes. Each link of the association is an n-tuple of values from the respective classes.

name

1. A human readable identifier. See: *identifier*.
2. The *name* (1) of a model element. (MOF, UML)

namespace

1. A mapping from *name*s (1) to entities denoted by those names.
2. An element of a metamodel whose primary purpose is to act as a *namespace* (1) for element names. (MOF)

Namespace

The abstract class in the MOF model that is the supertype of those classes that act as *namespace*s (2). The Namespace class also provides element containment in the MOF Model. (MOF)

natural language

A language that has no specification. A language that has evolved for human to human communication; e.g. English, Sanskrit, Amercan Sign Language.

nested package

A package that is defined as contained by another package in a MOF metamodel. An instances of a nested package can only exist in the context of an instance of its enclosing package. (MOF)

node

1. A component in a network. A network consists of nodes connected by edges.
2. 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. (UML)

notation

A system of human readable (textual or graphical) symbols and constructs for expressing information.

note

A comment attached to an element or a collection of elements. A note has no semantics. (UML)

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. (MOF, UML)

object reference

An *identifier* for an *object*, typcally a CORBA object. (OMA)

OCL — Acronym: Object Constraint Language. A pure expression language that is a non-normative part of the UML specification (ad/97-08-08) that is designed for expressing constraints. (UML)

operation — A service that can be requested from an object to effect behavior. An operation has a *signature*, which may restrict the *actual parameter*s that are possible. (MOF, UML)

ordered collection — A *collection* that is ordered. See *ordering*. (MOF)

ordering — A property of *collection*s. A collection is ordered if the sequence in which the elements appear needs to be preserved. (MOF)

package — A mechanism for organizing the elements of a model or metamodel into groups. Packages may be nested within other packages. (MOF, UML)

Package — The class in the MOF Model that describes a *package* in a metamodel. (MOF)

package cluster — A package that groups together a number of packages so that a set of instances of those packages can form a single extent. A package composition mechanism. (MOF 1.x)

package consolidation — Synonym: *package cluster*. (MOF 1.x)

package importing — See: *import* (3). A package composition mechanism. (MOF)

package inheritance — A generalization relationship between packages. Analogous to interface *interface inheritance* for classes. A package composition mechanism. (MOF)

package nesting — Defining one package inside another. A package composition mechanism. See: *nested package*. (MOF)

parameter — 1. A place holder for a value that can be changed, passed or returned by a computation. A parameter typically consists of a parameter name, a type and attributes that specify the information passing semantics for actual parameters. Synonym: *formal parameter*. Constrast: *actual parameter*, *argument*.
2. A parameter (1) of an operation or exception. (CORBA, MOF)
3. A parameter (1) of an operation, message or event. (UML)

postcondition — An *constraint* that must be true at the completion of a computation.

precondition — An *constraint* that must be true at the start of a computation.

primitive type — A type from which other types may be constructed, but that is not constructed from other types. See *type system*.

product — The artifacts of development, such as models, code, documentation, work plans. (UML)

profile — A simplified subset of a language or a metamodel.

projection — 1. A primitive operation in relational algebra which produces a relation by “slicing” one or more columns from another relation.
2. The set of MOF class instances that is visible via the reference operations of a class instance. For a class X, a n-ary association $A(X,Y_1, ... Y_{n-1})$ and an instance $x \in X$ then the expression

PROJECT [Y_1, ... Y_{n-1}] (SELECT A WHERE X = x)

defines the set of links. In the binary case, the set is a set of instances. (MOF)
3. A mapping from a set to a subset. (UML)

property — 1. A characteristic of an entity.
2. A *property* (1) that is represented as a mapping from an entity and a property name to a value for the property. See *tagged value*. (UML)

pseudo-code — An informal description of an algorithm in a language whose meaning is not fully defined.

published (meta-)model — A (meta-)model which has been frozen, and made available for use. For example, a published metamodel can be used to instantiate repositories and can be safely reused in other metamodels.

quokka — A small scrub-wallaby found on Rottnest Island, Western Australia.

read only — Describes an object or attribute for which no explicit update operations are provided. (MOF)

reference — 1. An identifier.
2. A use of a model element. (UML, MOF)
3. A feature of a class that allows a client to navigate from one instance to another via *association* links. See *projection* (2). (MOF)

Reference — A model element that defines an *reference* in a MOF metamodel. (MOF)

reflection — 1. A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
2. In the MOF, reflection characterizes what happens when a client examines and updates metadata without compile time knowledge of its metamodel. (MOF)

reflective — Describes something that uses or supports *reflection*.

reflective interfaces — Synonym: *generic interface*. (MOF)

Reflective — The name of the CORBA IDL module containing the MOF's reflective interfaces. (MOF)

reify — To produce an *object* representation of some *information*.

relation — A collection of *relationship*s (1) with the same roles. A relation is typically pictured as a two dimensional table with the rows representing relationship tuples, and the columns representing the roles and their values.

relationship — 1. A semantic connection between 2 or more entities where each entity fills a distinct role. A relationship is typically expressed as a tuple.
2. Colloquially, a *relation*.
3. A *relationship* (1) between elements of a model. Examples include associations and generalizations (MOF, UML).

repository 1. A logical container for metadata. (MOF)
2. A distributed service that implements a *repository* (1). (MOF)

requirement A desired feature, *property* (1), or behavior of a system.

responsibility A contract or obligation of a type or class. (UML)

reuse The act or process of taking a concept or artifact defined in one context and using it again in another context.

role 1. A position in a *relationship* or column in a *relation*.
2. 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). (UML)

root element The single outermost element in an *XML Document*. Synonym: *document element*. (XML)

run time The period of time during which a computer program executes.

scope 1. A region of a specification in which a given identifier or entity may be used.
2. An attribute of some features in the UML metamodel and MOF Model that determines if the feature is *instance level* or *classifier level*. (MOF, UML)

sequence 1. A CORBA constructed data type. (CORBA)
2. A *collection* whose data type does not specify ordering or uniqueness semantics. Differs from an *array* in that the number of elements is not fixed. (MOF)

set An *unordered collection* in which a given entity may appear at most once.

SGML Acronym: 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.

signature The name and parameters of an operation. Parameters may include an optional returned parameter. (MOF)

single inheritance A form of generalization in which a type may have only one supertype.

single-valued A *ModelElement* with a *multiplicity* is called single-valued when its upper bound is equal to one. The term single-valued does not pertain to the number of values held by the corresponding feature of an instance at any point in time. For example, a single-valued attribute, with a multiplicity lower bound of zero may have no value. Contrast: *multi-valued*.

specialization The reverse of a *generalization* relationship.

specific interfaces An interface for metadata described by a given metamodel that is tailored to the abstract syntax of that metamodel. Contrast: *generic interface*.

specification A precise description that can or should be used to produce things.

Standard Generalized Markup Language See: *SGML*

start tag A tag that marks the beginning of an element, such as <Model>. Also see end-tag. (XMI)

state The state of an object is the group of values that constitute its properties at a given point in time.

static In C++ or Java, a static attribute or a static member function is shared by all instances of a class. Synonym: *classifier level*.

static type checking Contrast: *dynamic type checking*.

static typing Contrast: *dynamic typing*.

strong typing A characteristic of a computational system that type failures are guaranteed not to occur.

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.

subclass In a generalization relationship the specialization of another class, the superclass. See: *generalization*.

subtype In a generalization relationship the specialization of another type, the supertype. See: *generalization*.

subsystem A part of a system that it is meaningful to describe in isolation.

superclass In a generalization relationship the generalization of another class, the subclass. See: *generalization*.

supertype In a generalization relationship the generalization of another type, the subtype. See: *generalization*.

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. (UML)

tagged value A representation of a *property* as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined; others may be user defined. (UML, MOF)

technology mapping A mapping that transforms a design expressed as a model or metamodel into implementation artifacts; e.g. CORBA IDL or program source code.

top-level package A *package* that is not nested in another package. (MOF)

transitive closure 1. The transitive closure of the value v_0 in V under the mapping m : V $\rightarrow$ V is defined by the following equation:

$$TC(v_0, m) \cong \{ v \in V : (v = v_0) \vee (\exists v_i \in TC(v0, m) : m(v_i) = v) \}$$

In other words, the set of all V's that are "reachable" from v_0 via the mapping. (Math)
2. The transitive closure of an initial object under an association is the set of objects reachable from the initial object via extant links in the association. (MOF, XMI)

type — 1. A predicate characterizing a collection of entities. (RM-ODP)
2. A predicate defined over values that can be used to restrict a possible parameter or characterize a possible result. Synonym: *type* (1). (OMA)
3. A *stereotype* of *class* that is used to specify a domain of *instance*s (*object*s) together with the operations applicable to the objects. A *type* (3) may not contain methods. (UML)

type checking — A process that checks for programs or executions that could lead to *type failure*.

TypeCode — A CORBA primitive data type. The TypeCode type is used in CORBA to pass runtime descriptions of CORBA types. A CORBA *any* value contains a TypeCode to describe the embedded value's type. See *any*. (CORBA)

type error — An event that is triggered when *type checking* detects a situation which could lead to *type failure*.

type expression — An expression that evaluates to a reference to one or more types. (UML)

type failure — A type failure occurs when a computation erroneously uses a value thinking it has one type when it has a different (incompatible) type. The consequences of a type failure are often completely unpredictable.

type loophole — A construct or artifice that allows a program to breach *type safety*.

type safety — A desirable property of a program or computation that *type failure*s are guaranteed not occur.

type system — A language for expressing *type*s (1). A type system is typically defined from a small set of *primitive type* and type constructors. See *metamodel.*

typing — Synonym: *type checking*.

unique list — An *ordered collection* in which no entity may not appear more than once as a collection member; i.e. a list in which duplicate elements are not allowed. (MOF)

uniqueness — A property of *collection* types that determines whether a given element may appear more than once in the collection. (MOF)

unordered collection — A *collection* in which the order in which the collection members appear has no significance. See *ordering*. (MOF)

UML, the — Acronym: The Universal Modeling Language. (UML)

UUID — Acronym: Universally Unique IDentifier. An identifier that guaranteed to be unique across all computer systems and across time, provided certain assumptions hold.

valid XML document — An *XML Document* that conforms to its *DTD*. (XML)

value	1. An element of a type domain. (UML) 2. An entity that can be a possible actual parameter in a request. (OMA)
view	A *projection* (3) of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective. (UML)
visibility	An *attribute* of a model element whose value (public, protected, private, or implementation) determines the extent to which the model element may be seen, and hence used, outside of the *namespace* in which it is defined.
W3C, the	Acronym: the World Wide Web Consortium. The standards body that takes the lead in developing standards related to the Web; e.g. HTML, HTTP and XML. (XML)
well-formed XML document	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. (XML)
XLink	An XML construct for representing links to external documents. See Xpointer. (XML)
XMI	Acronym: XML-based Metadata Interchange. The proposed OMG specification for a metadata interchange format that is based on the W3C's *XML* specification. (XMI)
XML	Acronym: Extensible Markup Language. A *profile* of SGML. XML is the W3C standard for representing structured information; e.g. web metadata. (XML)
XML Declaration	A processing instruction at the start of an XML document, which asserts that the document is an *XML Document*. (XML)
XML Document	An XML document consists of an optional *XML Declaration*, followed by an optional *DTD*, followed by a *document element*. (XML)
XPointer	An XML construct for linking to an element, range of elements, or text region within the same XML document. (XML-Link 6)

Index

A

abstract class. See class, abstract
abstract language. See language
abstraction Glossary-220
actual parameter. See parameter
aggregate 4-30, Glossary-220
aggregate class. See aggregate
aggregation Glossary-220
analysis Glossary-220
analysis time Glossary-220
annotation Glossary-220
any Glossary-221
architecture Glossary-221
argument Glossary-221
array Glossary-221
artifact Glossary-221
Association 4-30, 4-41, Glossary-221
association Glossary-221
  binary Glossary-221
  class. See class, association
  derived Glossary-224
  end. See association end
  n-ary Glossary-229
  role Glossary-221
association end Glossary-221
AssociationEnd 4-30, 4-41, Glossary-221
Attribute 4-30, Glossary-221
attribute Glossary-221
  derived Glossary-224
  element. See element, attribute

B

bag Glossary-221
base type. See type, base
behavior Glossary-221
binary association. See association, binary
boolean Glossary-221
builtin type. See type, builtin

C

cardinality 4-30, Glossary-222
CDATA section Glossary-222
CDIF 3-23, 4-32
Class 4-30, Glossary-222
class Glossary-222
  abstract Glossary-220
  association class Glossary-221
  composite Glossary-222, Glossary-223
  proxy Glossary-222
class diagram 4-41, Glossary-222
class proxy. See class, proxy
classifier Glossary-222
classifier level. See scope, classifier level
client Glossary-222
closure. See transitive closure
collection Glossary-222
  ordered Glossary-230
  unordered Glossary-234

compile time Glossary-222
compliance point
  optional 4-41, 4-42
compliance points
  optional 4-42
component Glossary-222
composite aggregation. See composition.
composite. See class, composite
composition Glossary-223
concrete class. See class, composite
concrete language. See language, concrete
Constraint 4-30, 4-41, Glossary-223
constraint Glossary-223
container Glossary-223
containment Glossary-223
  hierarchy Glossary-223
  matrix Glossary-223
CORBA Glossary-223
  general language mapping requirements 3-21
CORBA Any 9-176, 9-182
CORBA IDL 4-30
CORBA IDL. See IDL
CORBA TypeCode 9-187

D
data Glossary-223
data type. See type, data
DataType 4-30, Glossary-223
DCD 4-37
DCE 6-54
dependency Glossary-223
derived association. See association derived
derived attribute. See attribute, derived
derived element. See element, derived
descendent 6-73
design Glossary-224
design time Glossary-224
development process. See process, development
diagram Glossary-224
document element. See element, root
Document Type Definition. See DTD
domain Glossary-224
DTD 4-27, 4-35, Glossary-224
  automatic generation of 4-39
  boilerplate 4-39
  responsibility for standardization of 4-43
dynamic type checking. See type checking, dynamic
dynamic typing. See typing, dynamic

E
EBNF 4-27, Glossary-224
element 4-34, Glossary-224
  attribute 4-35, Glossary-225
  content 4-34, Glossary-225
  derived Glossary-224
  document. See element, root
  ID attribute 4-35
  nesting of 4-34
  root Glossary-232
  type Glossary-225
element attribute. See element, attribute

element content. See element, content
element type. See element, type
elemet
  generalizable Glossary-225
end tag. See tag, end
entity Glossary-225
enumeration Glossary-225
export Glossary-225
expression Glossary-225
extensions 4-27
extent Glossary-225

F
feature Glossary-225
formal language. See language, formal
formal parameter. See parameter, formal
framework Glossary-225
frozen Glossary-225

G
generalizable element. See element, generalizable
generalization Glossary-225
generic interface. See interface, generic
grammar Glossary-225

H
HTML 4-32, 4-33, Glossary-226

I
identifier Glossary-226
identity Glossary-226
IDL Glossary-226
  mapping. See mapping, IDL
IDL mapping. See IDL, mapping
IDrefs 6-73
immutable Glossary-226
implementation Glossary-226
implementation inheritance. See inheritance, implementation
Import Glossary-226
import Glossary-226
information Glossary-226
inheritance 4-30, Glossary-226
  implementation Glossary-226
  interface Glossary-227
  multiple Glossary-228
  single Glossary-232
instance Glossary-226
instance level. See scope, instance level
instantiate Glossary-227
interface
  generic Glossary-225
  inheritance. See inheritance,interface
  reflective Glossary-231
  specific Glossary-232
introspection Glossary-227
invariant Glossary-227

K
knowledge Glossary-227

L
language Glossary-227
  abstract Glossary-220
  concrete Glossary-223
  formal Glossary-225
  natural 4-30, Glossary-229
link Glossary-227
  role. See role, link
Linking 6-72, 6-73
list Glossary-227
  unique Glossary-234

M
mapping
  IDL Glossary-226
  technology Glossary-233
markup 4-32, Glossary-227
member Glossary-227
meta- 4-29, Glossary-227
Meta Object Facility, the. See MOF, the
metadata Glossary-227
  characteristics of 4-28
  complete encoding of 4-40
  definition 4-28
  differential 4-27
  examples of 4-28
  interchange 4-27
meta-level Glossary-227
  number of 4-29
meta-metadata Glossary-227
  metamodel 4-29
meta-meta-metadata Glossary-227
meta-metamodel Glossary-227
metamodel Glossary-227
  elaboration Glossary-227
  interchange 4-27
  knowledge of 4-40
  published Glossary-231
metamodel elaboration. See metamodel, elaboration
metaobject Glossary-228
  protocol Glossary-228
metaobject protocol. See metaobject, protocol
method Glossary-228
model Glossary-228
  aspect Glossary-228
  definition 4-28
  element Glossary-228
  fragments 4-41
  interchange 4-27
  interchange of ill-formed 4-41
  published Glossary-231
  versions of 4-42
ModelElement Glossary-228
modeling time Glossary-228
MODL 3-24, Glossary-228
module Glossary-228
MOF 4-28
  meta-metamodel Glossary-228
  metamodel Glossary-228
  model Glossary-228
MOF metamodel

XMI requires validity of 4-41
MOF model
definition 4-28
MOF Model, the Glossary-228
MOF metamodel 4-29
UML 4-30
MOF, the Glossary-228
MOF-based metamodel. See MOF metamodel
MOF-based model. See MOF model
multiple inheritance. See inheritance, multiple
multiplicity 4-30, Glossary-229
multi-valued Glossary-229

N

name Glossary-229
name space Glossary-229
Namespace Glossary-229
namespace Glossary-229
n-ary association. See association, n-ary
natural language. See language, natural
navigability 4-30
nested package. See package, nested
node Glossary-229
notation Glossary-229
note Glossary-229

O

object Glossary-229
reference. See object reference
Object Constraint Language (OCL) 9-167
object containment 9-170
object reference Glossary-229
ObjectContents 9-175
Object-Element 9-174
OCL 3-24, 4-30, Glossary-230
Operation 4-30
operation Glossary-230
ordered collection. See collection, ordered
ordering Glossary-230

P

Package 4-30, Glossary-230
import 4-30
inheritance 4-30
nested 4-30
package Glossary-230
cluster Glossary-230
consolidation Glossary-230
importing Glossary-230
inheritance Glossary-230
nested Glossary-229
nesting Glossary-230
typ-level Glossary-233
Parameter 4-30
parameter Glossary-230
actual Glossary-220
formal Glossary-225
postcondition Glossary-230
precondition Glossary-230
primitive type. See type, primitive
process

development Glossary-224
Producer 9-168, 9-188, 9-190, 9-193, 9-203, 9-204, 9-207
product Glossary-230
Production OCL Operations
NewTcId 9-207
TcDistance 9-207
Production Rules
AnyValue 9-187
AttributeAsElement 9-176
CharacterValue 9-185
ClassAttributes 9-173
CompositeAsElement 9-181
ContentsFromRoot 9-170
EmbeddedObject 9-175
EnumAsElement 9-186
EnumAttribute 9-179
IntegralValue 9-186
MvAttributeContents 9-178
ObjectAsElement 9-174
ObjectContents 9-174
ObjectReference 9-177
ObjRefOrDataValue 9-182
OtherExtentLinks 9-173
OtherLinks 9-171
RealValue 9-187
ReferenceAsElement 9-179
ReferencingElement 9-180
RequiredTypeDefinitions 9-188
SequenceValue 9-184
StringValue 9-185
StructValue 9-183
SvAttributeContents 9-177
TcAlias 9-192
TcArray 9-193
TcEnum 9-194
TcExcept 9-195
TcFixed 9-197
TcObjRef 9-194
TcRecursiveLink 9-199
TcSequence 9-193
TcSimple 9-197
TcString 9-196
TcStruct 9-192
TcUnion 9-195
TypeCodeState 9-190
TypeCodeValue 9-190
TypeId 9-189
TypeRef 9-189
UnionValue 9-184
production rules 9-167
profile Glossary-230
projection Glossary-230
property Glossary-231
prototype 2-19, 2-20
pseudo-code Glossary-231

R

RDF 4-37
read only Glossary-231
Reference Glossary-231
reference Glossary-231

object. See object reference
reflection Glossary-231
Reflective Glossary-231
reflective Glossary-231
reflective interfaces. See interface, reflective
reify Glossary-231
relation Glossary-231
relationship Glossary-231
repository Glossary-232
requirement Glossary-232
responsibility Glossary-232
reuse 4-30, Glossary-232
role Glossary-232
association. See association, role
link Glossary-227
root element. See element, root
run time Glossary-232

S

scope Glossary-232
classifier level Glossary-222
instance level Glossary-227
scrub-wallaby Glossary-231
sequence Glossary-232
set Glossary-232
SGML 4-33, Glossary-232
signature Glossary-232
single inheritance. See inheritance, single
single-valued Glossary-232
specialization Glossary-232
specific interfaces. See interface, specific
specification Glossary-232
start tag. See tag, start
state Glossary-233
static type checking. See type checking, static
static typing. See typing, static
stereotype 4-42, Glossary-233
string Glossary-233
empty Glossary-225
strong typing. See typing, strong
subsystem Glossary-233
subtype Glossary-233
superclass Glossary-233
supertype Glossary-233
supplier Glossary-233
syntax
abstract 4-28
system Glossary-233

T

tag 4-34
balanced pairs of 4-34
end Glossary-225
name 4-34
start Glossary-232
tagged value 4-42, Glossary-233
Tag-Value 6-68
tatic Glossary-233
technology mapping. See mapping, technology
timestamp 6-57
top-level package. See package, top-level

transitive closure Glossary-233
Transmitting Incomplete Metadata 6-69
Transmitting Metadata Differences 6-75
type Glossary-234
  base Glossary-221
  builtin Glossary-222
  checking. See type checking
  data Glossary-223
    encoding of 4-39
  element. See element, type
  error Glossary-234
  expression Glossary-234
  failure Glossary-234
  loophole Glossary-234
  primitive Glossary-230
  safety Glossary-234
  system Glossary-234
type checking Glossary-234
  dynamic Glossary-224
  static Glossary-233
TypeCode Glossary-234
typeing
  static Glossary-233
typing Glossary-234
  dynamic Glossary-224
  static Glossary-233

U

UML 4-32, 4-42
  metamodel for 4-43
UML, the Glossary-234
unique list. See list, unique
uniqueness 4-30, Glossary-234
unordered collection. See collection, unordered
UOL 3-24
URI 4-36
usage scenarios 5-45
UUID 6-54, Glossary-234
UUIDrefs 6-73

V

Value 6-68
value Glossary-235
vendor extensions
  support for 4-42
verified 6-57
view Glossary-235
visibility Glossary-235

W

W3C, the 4-27, 4-33, Glossary-235

X

XLink 9-177, 9-182, Glossary-235
  XPointer 4-37
XLinks 6-73
XMI 4-27, 9-171, Glossary-235
  applicability of 4-38
  data interchange using 4-43
  design goals for 4-38

document production rules 4-27
DTD production rules 4-27
generation of import/export tools for 4-39
use for data interchange 4-37
xmi
extender 6-57
extenderID 6-58
id 6-53
idref 6-55
label 6-54
name 6-59
position 6-60, 6-61
uuid 6-54
uuidref 6-55
version 6-56, 6-59
XMI element 6-56
XMI. 9-171
XMI.add 6-60
XMI.any 9-169, 9-182, 9-187, 9-188, 9-189
XMI.content 6-57, 9-169, 9-170, 9-172
XMI.contents 9-170
XMI.CorbaRecursiveType 9-199
XMI.CorbaTcAlias 9-192
XMI.CorbaTcAny 9-197, 9-198
XMI.CorbaTcArray 9-193, 9-194
XMI.CorbaTcBoolean 9-197, 9-198
XMI.CorbaTcChar 9-197, 9-198
XMI.CorbaTcDouble 9-197
XMI.CorbaTcEnum 9-194
XMI.CorbaTcEnumLabel 9-194
XMI.CorbaTcExcept 9-195
XMI.CorbaTcField 9-192, 9-195, 9-196
XMI.CorbaTcFixed 9-197
XMI.CorbaTcFloat 9-197
XMI.CorbaTcLong 9-197
XMI.CorbaTcLongDouble 9-197, 9-198
XMI.CorbaTcLongLong 9-197, 9-198
XMI.CorbaTcNull 9-197, 9-198
XMI.CorbaTcObjRef 9-194
XMI.CorbaTcOctet 9-197, 9-198
XMI.CorbaTcPrincipal 9-197, 9-198
XMI.CorbaTcSequence 9-193
XMI.CorbaTcShort 9-197
XMI.CorbaTcString 9-196
XMI.CorbaTcStruct 9-192
XMI.CorbaTcTypeCode 9-197, 9-198
XMI.CorbaTcUlong 9-197
XMI.CorbaTcUlongLong 9-198
XMI.CorbaTcUnion 9-195
XMI.CorbaTcUnionMbr 9-195
XMI.CorbaTcUshort 9-197
XMI.CorbaTcVoid 9-197, 9-198
XMI.CorbaTcWchar 9-197, 9-198
XMI.CorbaTcWstring 9-196
XMI.CorbaTypeCode 9-182, 9-188, 9-190
XMI.delete 6-60
XMI.difference 6-60
XMI.documentation 6-58
XMI.element.att 6-53
XMI.enum 9-186
XMI.extension 6-57

XMI.extensions 6-57
XMI.field 9-183, 9-184
XMI.header 6-57
XMI.link.att 6-54
XMI.metametamodel 6-59
XMI.metamodel 6-59
XMI.model 6-58
XMI.octetStream 9-184
XMI.reference 6-61, 9-177
XMI.replace 6-61
XMI.seqItem 9-178, 9-184
XMI.TypeDefinitions 9-188
XMI.unionDiscrim 9-184, 9-185
XMIDataType 6-68
XMIEnumSet 6-68
XML 4-27, 4-33, Glossary-235
  APIs 4-33, 4-38
  benefits of using 4-33
  conformance with style of 4-39
  display of 4-33
  links 4-37
  low cost of entry 4-33
  namespaces 4-37
  overview of 4-34
  uptake of 4-34
XML Declaration Glossary-235
XML Document Glossary-235
XML document
  semantic correctness of 4-36
  valid 4-36, 4-42, Glossary-234
  well-formed 4-36, Glossary-235
XML-Data 4-37, 4-39
XPointer 6-73, Glossary-235
XSL 4-33, 4-37